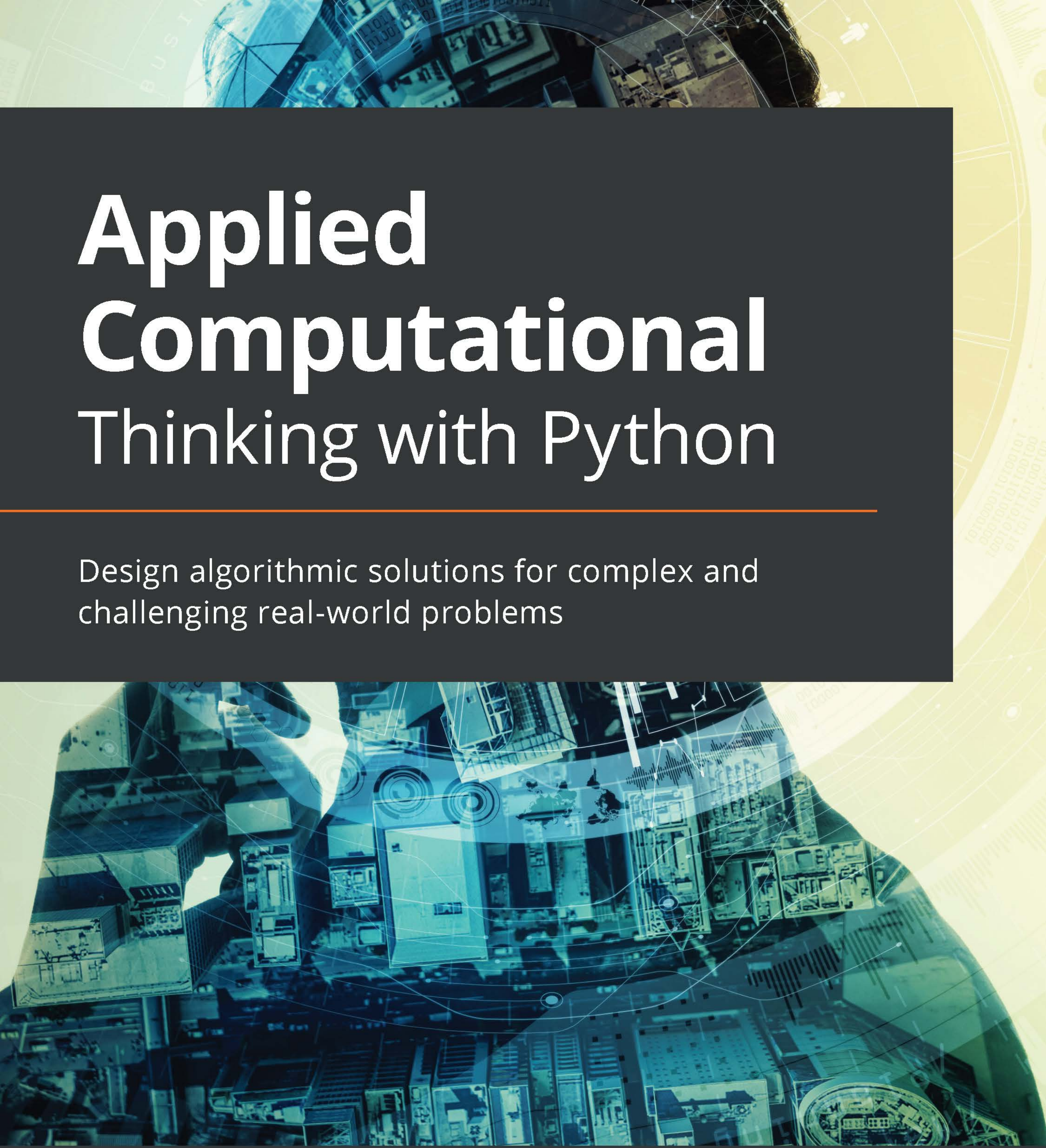
Applied Computational Thinking with Python

Design algorithmic solutions for complex and challenging real-world problems



Sofía De Jesús | Dayrene Martinez

Applied Computational Thinking with Python

Design algorithmic solutions for complex and challenging real-world problems

Sofía De Jesús

Dayrene Martinez

Packt

BIRMINGHAM—MUMBAI

Applied Computational Thinking with Python

Copyright © 2020 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author(s), nor Packt Publishing or its dealers and distributors, will be held liable for any damages caused or alleged to have been caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

Commissioning Editor: Richa Tripathi
Acquisition Editor: Alok Dhuri
Senior Editor: Rohit Singh
Content Development Editor: Rosal Colaco
Technical Editor: Pradeep Sahu
Copy Editor: Safis Editing
Project Coordinator: Deeksha Thakkar
Proofreader: Safis Editing
Indexer: Rekha Nair
Production Designer: Shankar Kalbhor

First published: November 2020

Production reference: 1261120

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham
B3 2PB, UK.

ISBN 978-1-83921-943-6

`www.packt.com`

To my parents, Elia and Miguel, and to my siblings, Rebeca, Graciela, and Miguelito, without whom I wouldn't be who I am or have been able to have the experiences that got me here. I love you all. And to Kristen McKeen, Will Murchison, Teresa Caro, and Joel Socarrás, friends without whom I couldn't survive this life.

- Sofía De Jesús

To my loving mother, Xiomara Garcia, for the everlasting support and love, and to my siblings, Karlyn and Junior Martinez, for being great role models—you guys inspire me every day. And to Samantha Tolentino Benitez, Noemi Reyes, Rosemary Vergara, Pedro Peralta, Charles Ozuna, and Gabriel Alatorre for being the best listeners when I needed it most. I am grateful for all of you.

- Dayrene Martinez

Packt.com

Subscribe to our online digital library for full access to over 7,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

Why subscribe?

- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Fully searchable for easy access to vital information
- Copy and paste, print, and bookmark content

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at packt.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at customercare@packtpub.com for more details.

At www.packt.com, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.

Contributors

About the authors

Sofía De Jesús is a computational thinking teacher with a degree from the University of Puerto Rico with a focus on math. She has a master's degree from the University of Dayton. Her experience in education and development spans two decades. She has worked with clients to develop solutions in multiple programming languages. As a teacher, Sofía helps students incorporate the philosophy of computational thinking in courses such as game design, circuits, Python, web design, and robotics. She likes to play video games and spend time with her 11 year-old Yorkie, King Kong. Sofía also enjoys creating materials, small furniture, and jewelry using CNC machines and laser cutters. She enjoys spending as much time in Puerto Rico as work and life permits.

I started this book before the world changed in 2020. If it wasn't for my family, my co-author, Dayrene, Gregg Moses, Emily Joyce, Princess Harrison, Brooke Marcy, and so many others who stepped up and held me up, this book would not have been possible. Thank you.

Dayrene Martinez is a systems engineer in the defense industry where she develops neural networks embedded in real-time software to aid in decision making for aerospace vehicles. She received a B.S. in electrical engineering from the New Jersey Institute of Technology (NJIT). Dayrene is passionate about STEM education and spends her free time in a voluntary capacity inspiring the next generation of engineers. She is an active supporter and keynote speaker for the non-profit organization Latinas in STEM. She teaches circuitry, robotic design, and software engineering to middle and high school students.

I would like to thank my co-author, Sofía, for helping me accomplish one of my lifetime goals—publishing a book. I would also like to thank Colin Pouchet and Scott Terhorst for your encouragement and guidance at work. I would also like to thank Greg Wagner, my boss, for stimulating my interest in machine learning by giving me challenging work, and Jaz Carvajal, without whom I would not even be an engineer. Thank you for being the best and most supportive mentor.

About the reviewer

Dr. Gowrishankar S. is currently working as associate professor in the department of computer science and engineering at the Dr. Ambedkar Institute of Technology, Bengaluru, Karnataka, India. His current research interests are mainly focused on data science, including its technical aspects, as well as its applications and implications. Specifically, he is interested in the applications of machine learning, data mining, and big data analytics in healthcare.

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit `authors.packtpub.com` and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Table of Contents

Preface

Applied Computational Thinking with Python provides a hands-on approach to implementation and associated methodologies that will have you up and running and productive in no time. Developers working with Python will be able to put their knowledge to work with this practical guide using the computational thinking method for problem solving.

This book will help you to develop logical processing and algorithmic thinking while solving real-world problems across a wide range of domains. It's an essential skill that you should possess to keep ahead of the curve in this modern era of information technology. Developers can apply their knowledge of computational thinking to practice solving problems in multiple areas, including economics, mathematics, and artificial intelligence.

The book begins by helping you get to grips with decomposition, pattern recognition, pattern generalization and abstraction, and algorithm design, along with teaching you how to apply these elements practically while designing solutions for challenging problems.

We'll find out how to use decomposition for solving problems through visual representation. We'll also employ pattern generalization and abstraction for designing solutions and building the analytical skills required to assess algorithmic solutions. We'll also use computational thinking with Python for statistical analyses.

We will understand the input and output requirements in terms of designing algorithmic solutions, and use computational thinking to solve data processing problems. We'll identify errors in logical processing to refine your solution design and apply computational thinking in various domains, such as cryptography, economics, and machine learning.

You'll then learn about the various techniques involved in problem analysis, logical reasoning, algorithm design, classification and clusters, data analysis, and modeling, and understand how computational thinking elements can be used together with these aspects to design solutions.

We'll also develop logical reasoning and problem-solving skills that will help you tackle complex problems. We'll also explore core computer science concepts and important computational thinking elements using practical examples and find out how to identify the algorithmic solution that is best-suited to your problem.

Toward the end, you will discover how to identify pitfalls in the solution design process and how to choose the right functionalities to create the best possible algorithmic solutions.

By the end of this algorithm book, you will have gained the confidence to apply computational thinking techniques to software development.

Who this book is for

This book is for students, developers, and professionals looking to develop problem-solving skills and tactics involved in writing or debugging software programs and applications. Familiarity with Python programming is required.

What this book covers

Chapter 1, *Fundamentals of Computer Science*, helps you learn about the fundamental elements of computer science, including theory, design, computational processes and systems, and computers. The focus of this chapter will be on the software elements of computer science.

Chapter 2, *Elements of Computational Thinking*, explains each of the elements of computational thinking—decomposition, pattern recognition, pattern generalization and abstraction, and algorithm design—and how the process of computational thinking is not linear. Rather, a developer can go back through some of these elements at all stages of the algorithm design process until a solution for a particular problem is reached. This chapter will include some short, relevant problems that are used to demonstrate the use of the elements of computational thinking to arrive at the algorithm.

Chapter 3, *Understanding Algorithms and Algorithmic Thinking*, provides you with an introduction to algorithms and their definition. You will also review some algorithms to help you develop the analysis skills necessary when assessing algorithms.

Chapter 4, *Understanding Logical Reasoning*, explores logical reasoning processes such as conditional statements, algorithmic reasoning, and Boolean logic. Throughout the chapter, you will learn basic and intermediate logic processing skills through real and relevant problem analysis.

Chapter 5, *Exploring Problem Analysis*, explores topics in problem analysis, focusing on problem definition, decomposition, and analysis. In order to practice and further understand the first element of computational thinking, decomposition, you will be presented with real and relevant problems. You will then be able to define and decompose the problem into parts, such as identifying input and output, and additional relevant information needed to begin planning solutions for the problems presented.

Chapter 6, *Designing Solutions and Solution Processes*, gives you an opportunity to design solutions to multiple problems using previously learned content in the computational thinking process and begin to incorporate logical processing to create a visual representation of the decision process for the solution. Visual representations include diagrams, flow charts, and other helpful processes.

Chapter 7, *Identifying Challenges within Solutions*, provides the opportunity to practice identifying some of the common errors and/or better possible solutions for an existing problem. While most problems can be solved with a multitude of algorithms that address the needs of the problem, some solutions are best suited for that particular problem. The goal of this chapter is to introduce you to the identification of some pitfalls in the solution design process.

Chapter 8, *Introduction to Python*, introduces you to basic Python commands and functionalities while applying them to problems. Using the elements of computational thinking, you will be able to design solutions by incorporating the concepts learned earlier.

Chapter 9, *Understanding Input and Output to Design a Solution Algorithm*, helps you to assess problems to identify the input and output needed in order to design and implement a solution algorithm for the problems.

Chapter 10, *Control Flow*, helps you to learn more about conditional statements and how to work with `for` and `while` loops when solving problems using computational thinking and the Python programming language. You will apply the logical processing learned previously to create Python algorithms when solving problems.

Chapter 11, *Using Computational Thinking and Python in Simple Challenges*, helps you apply the knowledge acquired previously to complete the computational thinking process when solving challenges in multiple disciplines, using examples that are real and relevant, in order to design the best possible algorithms for each scenario.

Chapter 12, *Using Python in Experimental and Data Analysis Problems*, explains how to utilize Python capabilities to solve problems that involve experimental data and data processing. The computational thinking elements will be used to solve real and relevant problems using advanced functionalities.

Chapter 13, *Using Classification and Clusters*, covers classification and clusters. You will use Pandas, Scikit-Learn, and other packages to create training and testing models, and learn some basic definitions for use in large data analysis, such as machine learning and neural network models.

Chapter 14, *Using Computational Thinking and Python in Statistical Analysis*, dives into further topics relating to statistical analysis, such as importing data, indexing, and preprocessing data. You will then use data visualizations to make decisions on what variables to explore for further analysis.

Chapter 15, *Applied Computational Thinking Problems*, helps you to work through multiple problems that combine topics from each of the previous chapters in order to solve a problem and design an algorithm in Python. You will use the computational thinking elements to determine what functionalities are necessary in order to design models and create solutions for problems in linguistics, cryptography, and more.

Chapter 16, *Advanced Applied Computational Thinking Problems*, works through additional applied problems in a variety of areas, including geometric tessellations, creating models of housing data, creating electric fields, analyzing genetic data, analyzing stocks, creating a **convolutional neural network** (**CNN**), and more. You will use the computational thinking elements to solve problems and create different figures and visual representations for your problems and datasets.

To get the most out of this book

You will need to have **Python 3.9** *installed on your computer in order to run the codes. All code examples have been tested on a Windows operating system using Python 3.9 and should run in subsequent releases. All code has also been tested using the Anaconda virtual environment.*

Software/hardware covered in the book	OS requirements
Python (3.9)	Windows, macOS X, and Linux (any)

Additional libraries and packages used in this book include the following:

- **NumPy**
- **Scikit-Learn**
- **TensorFlow/Keras**
- **Matplotlib**
- **Seaborn**

- **Cairos**
- **NLTK**
- **Pandas**

If you would like to run the code using a Spyder environment or Jupyter notebook, you can install Anaconda, an environment manager for Python and R programming languages.

If you are using the digital version of this book, we advise you to type the code yourself or access the code via the GitHub repository (link available in the next section). Doing so will help you avoid any potential errors related to the copying and pasting of code.

Download the example code files

You can download the example code files for this book from GitHub at `https://github.com/PacktPublishing/Applied-Computational-Thinking-with-Python`. In case there's an update to the code, it will be updated on the existing GitHub repository.

We also have other code bundles from our rich catalog of books and videos available at `https://github.com/PacktPublishing/`. Check them out!

Download the color images

We also provide a PDF file that has color images of the screenshots/diagrams used in this book. You can download it here: `https://static.packt-cdn.com/downloads/9781839219436_ColorImages.pdf`.

Conventions used

There are a number of text conventions used throughout this book.

`Code in text`: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: "Finally, we ask the program to analyze the text using the `Readability` function. Notice that we saved that to `r`."

A block of code is set as follows:

```
def encrypt(message, key):
    encryptedM = ''
    for letts in message:
        if letts in LETTERS:
            num = LETTERS.find(letts)
            num += key
            encryptedM +=  LETTERS[num]

    return encryptedM
```

Any command-line input or output is written as follows:

```
There once was a citizen in the town of Narnia, whose name was
Malena. Malena loved to hang with their trusty dog, King Kong.
You could always see them strolling through the market in the
morning, wearing their favorite blue attire.
```

Bold: Indicates a new term, an important word, or words that you see on screen. For example, words in menus or dialog boxes appear in the text like this. Here is an example: "As you can see from the preceding screenshot, the **deaths** column continues to rise, as does the number of cases, which we will take a look at a little later in this problem."

Tips or important notes

Appear like this.

Get in touch

Feedback from our readers is always welcome.

General feedback: If you have questions about any aspect of this book, mention the book title in the subject of your message and email us at `customercare@packtpub.com`.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you would report this to us. Please visit `www.packtpub.com/support/errata`, selecting your book, clicking on the Errata Submission Form link, and entering the details.

Piracy: If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at `copyright@packt.com` with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in, and you are interested in either writing or contributing to a book, please visit `authors.packtpub.com`.

Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions, we at Packt can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about Packt, please visit `packt.com`.

Section 1: Introduction to Computational Thinking

In this world that we live in, we interact with code without realizing it constantly throughout our day. When we search for something online, when we use our smartphone, when we use our fitness trackers or smartwatches, and so much more, there are algorithms involved. Even our cars have computers that we interact with, some of us daily. We're going to look at what is behind programming and designing algorithms, starting with some of the basic fundamentals of computer science, looking at some applications of computational thinking and algorithm designs with the Python programming language.

In this section, you will gain a clear understanding of computer science, the elements of computational thinking – namely, decomposition, pattern recognition, pattern generalization, and abstraction – and algorithm design.

This section comprises the following chapters:

- *Chapter 1, Fundamentals of Computer Science*
- *Chapter 2, Elements of Computational Thinking*
- *Chapter 3, Understanding Algorithms and Algorithmic Thinking*
- *Chapter 4, Understanding Logical Reasoning*

- *Chapter 5, Exploring Problem Analysis*
- *Chapter 6, Designing Solutions and Solution Processes*
- *Chapter 7, Identifying Challenges within Solutions*

1 Fundamentals of Computer Science

The world of computer science is a broad and complex one. Not only is it constantly changing and evolving, but the components we consider part of computer science are also adapting and adjusting. The computational thinking process allows us to tackle any problem presented with purpose and focus. No matter what the problem is, we can break it down, find patterns that will help us find solutions, generalize our solutions, and design algorithms that can help us provide solutions to the problems.

Throughout this book, we will be looking at the computational thinking process carefully, tackling problems in multiple areas and using the Python programming language and associated libraries and packages to create algorithms that help us solve these problems. Before moving into the problems, however, we will explore some of the important computer science concepts that will help us navigate the rest of the book.

In this chapter, we will explore the following topics:

- Introduction to computer science
- Theoretical computer science
- Systems software
- Computing
- Data types and structures

Technical requirements

You will need the latest version of Python to run the codes in this chapter. You will find the full source code used in this chapter here: `https://github.com/PacktPublishing/Applied-Computational-Thinking-with-Python/tree/master/Chapter01`

Introduction to computer science

When looking for a definition of computer science, you will encounter multiple variations, but all state that computer science encompasses all aspects of computers and computing concepts, including hardware and software. In computer science, hardware design is learned in courses offered in engineering or computer engineering, for the most part. The software side of computer science includes operating systems and applications, among other programming areas. For the purposes of this book, we will be concentrating on the software side of computer science.

In this chapter, we'll look at some of the basic definitions, theories, and systems that are important as we delve deeper into the computational thinking world. Once we have identified key areas and defined the concepts, we will be ready to move on to the applications and real-world challenges we face in an ever-changing tech world while also exploring the elements of computational thinking and the Python programming capabilities that can help us tackle these challenges.

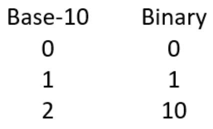
The wide range of topics available in computer science can be both daunting and exciting and it is ever evolving. Some of these topics include game design, operating systems, applications for mobile or desktop devices, the programming of robots, and much more. Constant and consistent breakthroughs in computers and computing provide new and exciting opportunities, much of which are unknown to us. Having a basic understanding of the systems behind computer science can help us interact with technology and tackle problems more efficiently.

Learning about computers and the binary system

All computers store information as **binary** data. The binary system reads all information as a switch, which can be on or off, 0 or 1. The binary system is a base-2 system. You'll need a basic understanding of binary numbers and the binary system to progress in computer science.

The binary system translates all data so that it can be stored as strings using only two numbers: 0 and 1. Data is stored in computers using bits. A **bit** (which stands for **binary digit**) is the smallest unit of data you can find in a computer, that is, a 0 or a 1.

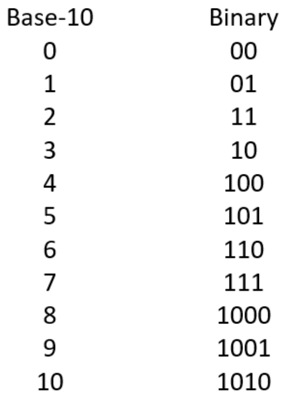
When counting in the binary system, the first two numbers are 0 (or 00) and 1 (or 01), much like in the base-10 number system we use in everyday life. If we were to continue counting in binary, our next number would be 10. Let's compare the first three numbers in the base-10 system and the binary system before we learn how to convert from one to the other:



Base-10	Binary
0	0
1	1
2	10

Figure 1.1 – Base-10 and binary comparison

The next number in the base-10 system would be 3. In the binary system, the next number would be 11, which is read as *one one*. The first 10 numbers in the base-10 and binary systems are shown as follows:



Base-10	Binary
0	00
1	01
2	11
3	10
4	100
5	101
6	110
7	111
8	1000
9	1001
10	1010

Figure 1.2 – Base-10 and binary comparison (cont'd)

As mentioned, the binary system is a base-2 system. This means that each digit of the base-10 system is paired with a power of 2, so we use those powers to convert between numbers. Understanding how to convert from base-2 to base-10 and vice versa can help us have a better understanding of the relationship between numbers in the different systems.

Converting from binary to base-10

We will start with an example to convert from a binary number to a base-10 number. Take the number 101101. To convert the number, each digit is multiplied by the corresponding base-2 power. The binary number given has 6 digits, so the powers of 2 we will use are 5, 4, 3, 2, 1, and 0. This means the number is converted as follows:

$$1 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$$
$$= 32 + 0 + 8 + 4 + 0 + 1 = 45$$

The binary number 101101 is equivalent to 45 in the base-10 system. In everyday life, we write numbers in base-10, so we understand the number 45 as written. However, our computers convert this information into binary to be able to process it, so the number becomes the binary number 101101 so that it can be easily read by the computer.

Converting from base-10 to binary

Again, let's start with an example to demonstrate the process of converting from a base-10 number to a binary number. Take the number 591. To convert the base-10 number to binary, we have to divide the number by 2 iteratively. If the result has no remainder, we insert a 0 (if it is the first number) or insert a 0 to the left of the existing numbers.

If the result has a remainder of 1, we insert a 1 (if it is the first number) or insert a 1 to the left of the existing numbers.

When we divide 591 by 2, the result is 295 with a remainder of 1. That means our right-most number, which is our first number, is 1.

Now divide 295 by 2. The result is 147 with a remainder of 1. So, we insert a 1 to the left of the 1. Our number is now 11.

Now divide 147 by 2. The result is 73 with a remainder of 1. Our result is now 111. Now we'll carry out further divisions:

- $73 \div 2 = 36$ with a remainder of 1. Our number is now 1111.
- $36 \div 2 = 18$ with no remainder. Our number is now 01111.
- $18 \div 2 = 9$ with no remainder. Our number is now 001111.
- $9 \div 2 = 4$ with a remainder of 1. Our number is now 1001111.
- $4 \div 2 = 2$ with no remainder. Our number is now 01001111.
- $2 \div 2 = 1$ with no remainder. Our number is now 001001111.
- $1 \div 2 = 0$ with a remainder of 1. Our number is now 1001001111.

The number 591 in base-10 is equivalent to the number 1001001111 in the binary system.

Another way to convert the number is to use a table for the divisions:

Starting Base-10	Divided by 2	Remainder
591	295	1
295	147	1
147	73	1
73	36	1
36	18	0
18	9	0
9	4	1
4	2	0
2	1	0
1	0	1

Table 1.1 – Conversion of the base-10 number 591 to binary

Using the table, take the numbers from the right-most column and write them starting with the last row from bottom to top. The result is 1001001111.

Learning how to convert numbers is only a small piece of converting data to binary, but it is an important piece. All information, including letters and symbols, must be converted to binary in order to be read by a computer. **ASCII** (which stands for **American Standard Code for Information Exchange**) is a protocol that has been adopted universally to convert information. That said, some of the protocol is obsolete, so other protocols use ASCII as a base to expand on its capabilities. Unicode is a widely used 16-bit character set that is based on ASCII.

As discussed, in this section, we learned that information must be encoded or converted in order for a computer to read it. Multiple systems and protocols exist, but for now, we will move on to computer science theory. However, revisiting binary, ASCII, and Unicode as you work through problems can be helpful.

Understanding theoretical computer science

While you don't need to be a master mathematician to love computer science, these two subjects are intrinsically tied. Computer science, particularly programming, uses algorithms, which are algebraic in nature. We will explore algorithms in depth later on, but again, the important point here is that they are mathematical. The logical processes stem from the philosophical nature and history of mathematics. Now, if mathematical topics are not to your liking, don't despair. The logical processes needed to become a programmer and developer can be used without having to learn higher mathematics. Knowing higher mathematics just simplifies some concepts for those who have that background.

Theoretical computer science includes multiple theories and topics. Some of the topics and theories are listed as follows, but keep in mind that other topics are also included in theoretical computer science that may not be discussed in this book. A short description and explanation for each of the theories or terms listed as follows are included for your review:

- Algorithms
- Coding theory
- Computational biology
- Data structures
- Cryptography
- Information theory
- Machine learning
- Automata theory
- Formal language theory
- Symbolic computation
- Computational geometry
- Computational number theory

We will look at the aforementioned theories in the following sections.

Algorithms

An algorithm is a set of instructions that a computer can read. Algorithms provide the rules or instructions in a way that means a computer can logically process the information provided as input and create an output. In most books, you are introduced to the algorithm and programming by creating the *Hello World!* program. I won't make this book the exception.

In Python, the code would require that we print the message to the screen. Because the Python language is easy to learn and to read, many, if not most, of the code strives to be logical. So, in order to print a message to the screen, we use the `print()` command. Here is the code we'd use:

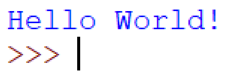
```
print("Hello world!")
```

Similarly, we could use the code given as follows:

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

Python reads both " and ' as the same thing when it comes to strings.

The result of the preceding code looks like the following screenshot when we run the algorithm:



```
Hello World!
>>> |
```

Figure 1.3 – The 'Hello World!' Python program

Don't worry, we'll discuss the Python programming language later on in *Chapter 2*, *Elements of Computational Thinking*, and more in depth in *Part 2*, *Applying Python and Computational Thinking*, starting in *Chapter 8*, *Introduction to Python*, as well.

While lengthy, the discussion on algorithms is critically important to this book and to your progression with Python. Consequently, we will be covering this in-depth exploration of algorithms in *Chapter 2*, *Elements of Computational Thinking*, and *Chapter 3*, *Understanding Algorithms and Algorithmic Thinking*, of this book, as algorithms are a key element of the computational thinking process.

Important Note:

Chapter 2, *Elements of Computational Thinking*, will focus on the computational thinking process itself, which has four elements: **decomposition**, **pattern recognition, pattern generalization and abstraction**, and **algorithm design**. As you can see, that last element is algorithm design, so we will need to get more acquainted with what an algorithm is and how we can create them so that you can then implement and design algorithms when solving problems with Python. *Chapter 3*, *Understanding Algorithms and Algorithmic Thinking*, will focus on a deeper understanding of algorithm definition as well as an introduction to the design process.

We'll look at coding theory next.

Coding theory

Coding theory is also sometimes known as algebraic coding theory. When working with code and coding theory, there are three areas that are studied: **data compression**, **error correction**, and **cryptography**. We will cover these in more detail in the following sections.

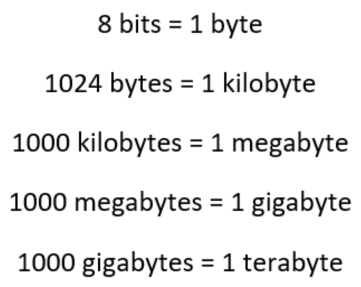
Data compression

The importance of data compression cannot be understated. Data compression allows us to store the maximum amount of information while taking up the least amount of space. In other words, data compression uses the fewest number of bits to store the data.

Important Note:

Remember that a **bit** is the smallest unit of data you can find in a computer, that is, a 0 or a 1, while a group of bits is called a **byte**. One byte usually has 8 bits. We use bytes as a unit of measurement for the size of the memory of a computer, storage device, such as a memory card or external drive, and more.

As our technology and storage capacities have grown and improved, our ability to store additional data has as well. Historically, computers had **kilobytes** or **megabytes** of storage when first introduced into households, but they currently have **gigabytes** and **terabytes** worth of storage. The conversions for each of the storage units are shown as follows:



8 bits = 1 byte

1024 bytes = 1 kilobyte

1000 kilobytes = 1 megabyte

1000 megabytes = 1 gigabyte

1000 gigabytes = 1 terabyte

Figure 1.4 – Byte conversions

If you look for information online, you may find that some sources state that there are 1,024 gigabytes in a terabyte. That is a binary conversion. In the decimal system, or base-10 system, there are 1,000 gigabytes per terabyte. To understand conversion better, it is important to understand the prefixes that apply to the base-10 system and the prefixes that apply to the binary system:

Base-10 prefixes	Value	Binary prefixes	Value
kilo	1000	kibi	1024
mega	1000^2	mebi	1024^2
giga	1000^3	gibi	1024^3
tera	1000^4	tebi	1024^4
peta	1000^5	pebi	1024^5
exa	1000^6	exbi	1024^6
zetta	1000^7	zebi	1024^7
yotta	1000^8	yobi	1024^8

Table 1.2 – Base-10 and binary prefixes with values

As mentioned, the goal is always to use the least amount of bits for the largest amount of data possible. Therefore, we compress, or reduce, the size of data in order to use less storage.

So, *why is data compression so important?* Let's go back in time to 2000. Back then, a laptop computer on sale for about $1,000 had about 64 MB of **RAM** (**Random Access Memory**) and 6 GB of hard drive memory. A photograph on our digital phones takes anywhere from 2 to 5 megabytes of memory when we use its actual size. That means our computers couldn't store many (and in some cases any) of the modern pictures we take now. Data compression advances allow us to store more memory, create better games and applications, and much more, as we can have better graphics and additional information or code without having to worry as much about the amount of memory they use.

Error correction

In computer science, errors are a fact of life. We make mistakes in our processes, our algorithms, our designs, and everything in between. Error correction, also known as error handling, is the process a computer goes through to automatically correct an error or multiple errors, which happens when digital data is incorrectly transmitted.

An **Error Correction Code** (**ECC**) can help us analyze data transmissions. ECC locates and corrects transmission errors. In computers, ECC is built into a storage space that can identify common internal data corruption problems. For example, ECC can help read broken codes, such as a missing piece of a **QR** (**Quick Response**) code. A type of ECC is a **hamming code**. A hamming code is a binary linear code that can detect up to two-bit errors.

Important Note:

Hamming codes are named after Richard Wesley Hamming, who discovered them in 1950. Hamming was a mathematician who worked with coding as related to telecommunications and computer engineering.

Another type of ECC is a **parity** bit. A parity bit checks the status of data and determines whether any data has been lost or overwritten. Error correction is important for all software developed, as any updates, changes, or upgrades can lead to corruption of the entire program or parts of the program or software.

Cryptography

Cryptography is used in computer science to hide code. In cryptography, information or data is written so that it is unreadable by anyone other than the intended recipient of the message. In simple terms, cryptography takes readable text or information and converts it into unreadable text or information.

When we think about cryptography now, we tend to think of **encryption** of data. Coders encrypt data by converting it into code that cannot be seen by unauthorized users. However, cryptography has been around for centuries, that is, it pre-dates computers. Historically, the first uses of cryptography were found around 1900 BC in a tomb in Egypt. Atypical or unusual hieroglyphs were mixed with common hieroglyphs at various parts of the tomb.

The reason for the unusual hieroglyphs is unknown, but the messages were hidden from others with their use. Later on, cryptography would be used to communicate in secret by governments and spies, in times of war and peace. Nowadays, cryptography is used to encrypt data, as our information exists in digital format, so protecting sensitive information, such as banking, demographic, or personal data is important.

We will be further exploring the topics of coding theory through some of the problems presented throughout this book.

Computational biology

Computational biology is the area of theoretical computer science that focuses on the study of biological data and bioinformatics. **Bioinformatics** is a science that allows us to collect biological data and analyze it. An example of bioinformatics is the collection and analysis of genetic codes. In the study of biology, large quantities of data are explored and recorded.

Studies can be wide-ranging in topics and interdisciplinary. For example, a genetic study may include data from an entire state, an entire race, or an entire country. Some areas within computational biology include molecules, cells, tissues, and organisms. Computational biology allows us to study the composition of these things, from the most basic level to the larger organism. Bioinformatics and computational biology provide a structure for experimental studies in these areas, create predictions and comparisons, and provide a way to develop and test theories.

Computational thinking and coding allow us to process that data and analyze it. In this book, problems presented will allow us to explore ways in which we can use Python in conjunction with computational thinking to find solutions to complex problems, including those in computational biology.

Data structures

In coding theory, we use data structures to collect and organize data. The goal is to prepare the data so that we can perform operations efficiently and effectively. Data structures can be primitive or abstract. Software has built-in data structures, which are the primitive data structures, or we can define them using our programming language. A primitive data structure is pre-defined. Some primitive data structures include integers, characters (**char**), and Boolean structures. Examples of abstract or user-defined data structures include arrays and two-dimensional arrays, stacks, trees and binary trees, linked lists, queues, and more.

User-defined data structures have different characteristics. For example, they can be linear or non-linear, homogeneous or non-homogeneous, and static or dynamic. If we need to arrange data in a linear sequence, we can use an array, which is a linear data structure. If our data is not linear, we can use non-linear data structures, such as graphs. When we have data that is of a similar type, we use homogeneous data structures.

Keep in mind that an array, for example, is both a linear and homogeneous data structure. Non-homogeneous or heterogeneous data structures have dissimilar data. An example of a non-homogeneous data structure a user can create is a class. The difference between a static and a dynamic data structure is that the size of a static structure is fixed, while a dynamic structure is flexible in size. To build a better understanding of data structures, we will explore them through problem solving using the computational thinking elements throughout this book. We will revisit data structures again very briefly at the end of this chapter, as they relate to data types, which are discussed then.

Information theory

Information theory is defined as a mathematical study that allows for the coding of information so that it can be transmitted through computer circuits or telecommunications channels. The information is transmitted through sequences that may contain symbols, impulses, and even radio signals.

In information theory, computer scientists study the quantification of information, data storage, and information communication. Information can be either analog or digital in information theory. **Analog data** refers to information represented by an analog signal. In turn, an analog signal is a continuous wave that changes over a given time period. A **digital signal** displays data as binary, that is, as a discrete wave. We represent analog waves as sine waves and digital waves as square waves. The following graph shows the sine curve as a function of value over time:

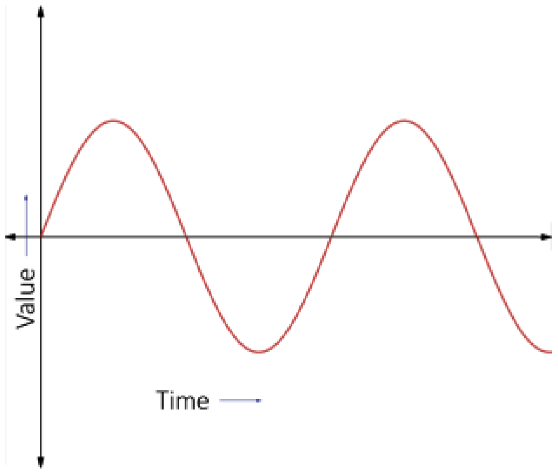


Figure 1.5 – Analog signal

An analog signal is described by the key elements of a sine wave: amplitude, period, frequency, and phase shift:

- The **amplitude** is the height of the curve from its center. A sine curve repeats infinitely.
- The **period** refers to the length of one cycle of the sine curve, that is, the length of the curve before it starts to repeat.
- The **frequency** and the period of the sine curve have an inverse relationship:

  $$\text{frequency} = \frac{1}{\text{period}}$$

  In relation to the inverse relationship, we can also say:

  $$\text{period} = \frac{1}{\text{frequency}}$$

- The **phase shift** of a sine curve is how much the curve shifts from 0. This is shown in the following graph:

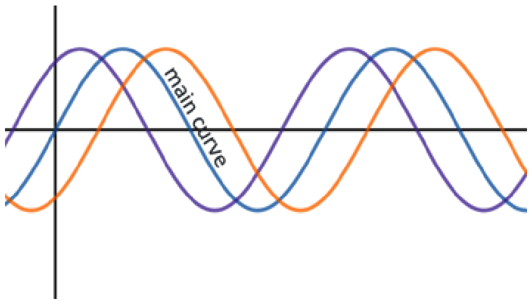


Figure 1.6 – Phase shift examples

In contrast, digital signal graphs look like bar graphs or histograms. They only have two data points, 0 or 1, so they look like boxy hills and valleys:

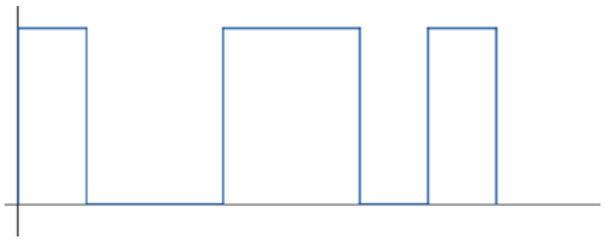


Figure 1.7 – Digital signal

Digital signals have finite sets of discrete data. A dataset is discrete in that it contains individual and distinct data points. For analog signals, the data is continuous and infinite. When working with computer science, both types of signals are important and useful. We will explore digital signals in some of the applications in later problems throughout the book, and specifically in the problems presented in *Chapter 16*, *Advanced Applied Computational Thinking Problems*.

Automata theory

Automata theory is one of the most fascinating topics in theoretical computer science. It refers to the study of machines and how calculations can be completed in the most reliable and efficient way. Automata theory involves the physical aspects of simple machines as well as logical processing. So, *what exactly is automata used for and how does it work?*

Automata are devices that use predetermined conditions to respond to outside input. When you look at your thermostat, you're working with an automata. You set the temperature you want and the thermostat reacts to an outside source to gather information and adjust the temperatures accordingly.

Another example of automata are surgical robots. These robots can improve the outcomes of surgeries for patients and are being improved upon constantly. Since the goal of automata theory is to make machines that are reliable and efficient, it is a critical piece in the development of artificial intelligence and smart robotic machines such as surgical robots.

Formal language theory

Formal language theory is often tied to automata theory in computer science. Formal language is the study of the syntax, grammar, vocabulary, and everything involving a formal language. In computer science, formal language refers to the logical processing and syntax of computer programming languages. With regard to automata, the machines process the formal language to perform the tasks or code provided for it.

Symbolic computation

Symbolic computation is a branch of computational mathematics that deals with computer algebra. The terms *symbolic computation* and *computer algebra* are sometimes used interchangeably. Some programming software and languages are focused on the symbolic computations of mathematics formulas. Programs using symbolic computation perform operations such as polynomial factorization, simplifying algebraic functions or expressions, finding the greatest common divisor of polynomials, and more.

In this book, we will use computer algebra and symbolic computation when solving some real-world problems presented. Python allows us to not only perform the mathematical computations that may be required for problems, but also explore graphical representations or models that result from those computations. As we explore solutions to real-world problems, we will need to use various libraries or extensions to the Python programming language. More on that throughout *Part 2*, *Applying Python and Computational Thinking*, of this book, where we will explore the Python programming language in greater detail.

Computational geometry

Like symbolic computation, **computational geometry** lives in the branch of computer science that deals with computational mathematics. The algorithms we study in computational geometry are those that can be expressed with geometry. The analysis of the data is done with geometric figures, geometric analysis, data structures that follow geometric patterns, and more. The input and output of problems that require computational geometry are geometric.

When thinking of geometry, we often revert to the figures we mostly associate with that branch of mathematics, such as polygons, triangles, and circles. That said, when we look at computational geometry, some of the algorithms are those that can be expressed by points, lines, other geometric figures, or those that follow a geometric pattern. Triangulation falls under this branch of computer science.

Triangulation of data is important for applications such as optical 3D measuring systems. We triangulate GPS signals to locate a phone, for example, which is used in law enforcement.

There are many uses of triangulation in modern times, some of which we'll explore through real and relevant problems presented in this book.

Computational number theory

Number theory is the branch of mathematics that studies integers and their properties. **Computational number theory** then is the study of algorithms used to solve problems in number theory. Part of the study of number theory is primality testing.

Algorithms created to determine whether input or output is prime have been used for many purposes. One of the most critically important uses and applications of primality testing and number theory is for encryption purposes. As our lives have moved to saving everything electronically, our most personal information, such as banking information, family information, and even social security numbers, live in some code or algorithm. It is important to encrypt such information so others cannot use or access it. Computational number theory and cryptography are intrinsically tied, as you will be able to explore later.

Some of the theories presented are meant to help you understand how intertwined computer science theories are, their applications, and their relevance to what we do each day.

In this section, we learned about theoretical computer science. We also learned about its various theories.

Throughout this book, we will be using computational thinking (discussed further in *Chapter 2*, *Elements of Computational Thinking*) to help us tackle problems, from the most basic applications to some complex analyses, by defining and designing adequate algorithms that use these theories. Theoretical computer science is used to study a system's software, which we will explore next.

Learning about a system's software

System's software is used to perform multiple functions and communicate between the **operating system** (**OS**) of a computer, peripherals such as a keyboard and mouse, and firmware, which is permanently saved to a device and is needed for its operation, among other functions. These are part of the two main types of software: **system software** and **application software**.

System software allows a computer to communicate between the hardware and the applications. Think of a smartphone. The phone is composed in its most basic form of the hardware, which includes the battery, cameras, memory, screen, and all the physical components and peripherals. The OS allows those components to be used by applications.

Take the camera application of a phone. The system's software lets the application communicate with the phone to use the camera to take a picture, edit it, save it, and share it. A computer's OS also allows the hardware to communicate with programs. A design program will use the mouse or other peripheral that can be used to draw, create, use a touchscreen if available, and more.

If we do not know our system's software, we cannot create applications that can communicate effectively with our hardware, creating errors that can range from critical, or rendering a peripheral useless, to minor, where some components may work, say taking a picture, but others may not, such as saving or sharing the picture. The system's software is created in a way that provides us with the easiest, most efficient way to communicate between the hardware and applications.

Operating systems

The OS performs multiple tasks. If you recall, error handling is part of an OS that checks for the most common possible errors in order to fix them without creating a larger problem or rendering an application worthless. Error handling is one of the operating system's most important tasks. In addition, the OS is responsible for the security of your computer or device. If you have a smartphone, you know that many updates to the OS are done in order to fix a security problem or to prevent a security breach. The OS is responsible for only allowing an authorized user to interact with the content that is stored in the device.

In addition to security and error handling, an OS is responsible for allocating memory for files and organizing them. When we save and delete a file or program, the memory that had been used is now free. However, there may be something saved immediately before and immediately after. The OS allocates and reallocates memory in order to maintain the best performance possible by the device. Memory management not only refers to user-saved files, but also to the RAM.

The file management of a device is also run by the OS. The OS will allocate the information as a filesystem, breaking the information into directories that are easily accessed by the user and by the device. The filesystem is responsible for keeping track of where files are, both from the OS and from the user, the settings for access to the device, which are evolving constantly, and how to access the files and understand the status of the files. Access to devices has changed in recent years.

While computers typically use a username and password, many devices can now be accessed through a fingerprint, a numerical or alpha-numerical passcode, facial recognition, images, paths, and more. As any of these topics evolve, the OS evolves as well and needs to be updated or recreated. The operating system is also responsible for allowing communication between the applications and the device.

Application software

Application software refers to software applications that perform a particular task. Think of the applications, or apps, that you can access from a mobile device. There are hundreds of types of applications, such as static games that live on the device, games that allow you to play with others remotely, news applications, e-book readers, fitness training apps, alarms, clocks, music, and so much more! Applications always perform some form of task, be it for personal use, business use, or educational use.

Application software has multiple functions. You may find suites for productivity, such as **Microsoft** (**Office**) and **Google** products. When we need to do research on the internet, we use applications called browsers, which allow us to access the information and index the information so that we can access it. These browsers include **Google Chrome**, **Safari**, **Firefox**, **Edge**, **Opera**, and others. Browsers are used by both mobile devices and computers. Keep in mind that the purpose of an app is to perform a specific task for the end user.

Important Note:

As an aside, applications have grown exponentially since computers became household tools and phones started being used for other things rather than just for calling others. Early computers were used for just that: computing, or calculating mathematical analyses and tasks. That's one of the reasons it is so important to have an understanding of the development and history of computer science. Since we cannot completely predict future uses of computer science and system software, the more we know about them, the more we will be able to create and adapt when technological advances happen.

In this section, we learned about the system's software. We also learned about OS software and application software. For the purposes of this book, some applications will be more important as we sort through some of the problems presented, such as databases, productivity software, enterprise resource planning, and educational software.

In the next section, we'll learn about computing.

Understanding computing

In computer science, **computing** refers to the activities that computers perform in order to communicate, manage, and process information. Computing is usually divided into four main areas: **algorithms**, **architecture**, **programming languages**, and **theory**.

Since, we've discussed theory and algorithms in previous sections, we will now focus on defining architecture and programming languages.

Architecture

Computer architecture refers to the set of instructions that interact with computer systems. In more basic terms, the architecture includes the instructions that allow software and hardware to interact. Computer architecture has three main subcategories: **Instruction Set Architecture** (**ISA**), **Microarchitecture**, and **System Design**.

Instruction Set Architecture (ISA)

The ISA is the boundary that exists between the hardware and the software. It is classified in multiple ways, but two common ones are **complex instruction set computer** (**CISC**) and **reduced instruction set computer** (**RISC**). These are defined as follows:

- **CISC**: This is a computer that has explicit instructions for many tasks, such as simple mathematical operations, and loading something from memory. CISC includes everything that is not included in RISC.
- **RISC**: This is a computer with an architecture that has reduced **cycles per instruction** (**CPI**).

CISC tries to complete instructions with fewer steps, while RISC only uses simple instructions. CISC is multi-step, while RISC is single-step, performing one task at a time. The CISC process includes the instructions, the microcode conversion, microinstructions, and execution. By contrast, RISC includes instructions and execution.

In CISC, **microcode** conversion refers to the interpretation of the language at a lower level. It takes into consideration the hardware resources to create microinstructions. **Microinstructions** are single instructions in microcode. After microcode creates the microinstructions, the microinstructions can be executed. The following diagram shows the process for both RISC and CISC:

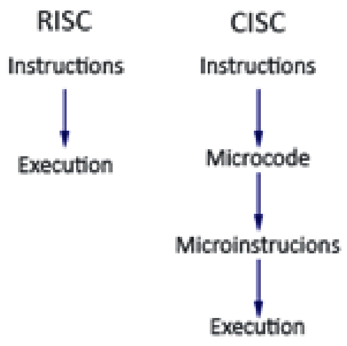


Figure 1.8 – Difference between RISC and CISC

Both RISC and CISC are necessary for computer programmers. There are advantages and disadvantages to having a single-step process (RISC) versus a multi-step process (CISC). RISC reduces the cycles per instruction, doing one thing at a time. CISC reduces the instructions in a program, but at the cost of cycles per instruction. Depending on what our needs are, we can choose the best path to take.

Programming languages

Programming languages are the way we write instructions for computers and other devices. Different languages are used depending on what is needed, ease of use, and much more. Examples of programming languages include the following:

- **Ruby** and **Python**: Ruby is a programming language mostly used for web applications. Ruby is stable and easy to use; however, many developers choose to use Python over Ruby because Python is faster. Although Ruby has not been as popular and had some performance issues, the language is very much alive in 2019 and continues to grow. Python, on the other hand, is widely used for multiple purposes, such as web applications, user interface applications, and websites, among others. We will explore Python in greater depth later on in this book.
- **C**: The C languages are a critically important part of computer science, as C was the first language used and is still the most widely used language. C has been around since 1972, when Dennis Ritchie invented it, but it has been used by others since 1978, when it was first published. While other languages have grown in popularity since, C is still used in 2019. Some of its uses include operating systems, hardware drivers, and applications, among others. C is a base-level language, which means it requires almost no abstraction.
- **C++**: C++ was developed by Bjarne Stroustrup as an extension of C in 1985. The goal of the language was to add object-oriented capabilities. The language is still widely used both in conjunction with the C language in operating systems and for other software. C++ is an intermediate-level programming language.
- **C#**: C# (C sharp) is a high-level programming language. Much like C++, it has object-oriented capabilities and is an extension of the C programming language. One of the main differences between C++ and C# is that C++ uses machine code while C# uses bytecode. Machine code can be executed directly by a computer. Bytecode has to be compiled, so it is considered a low-level code that needs to be interpreted.
- **Swift**: The Swift programming language was developed by **Apple Inc.** in 2014. As programming languages go, Swift is one of the newest. Apple released it as an open source programming language with **version 2.2**, which was released in 2015. The language is considered to be a general-purpose and compiled programming language.

- **Scratch**: Scratch was developed as a visual programming, block-coding language in 2002 by **MIT Media Lab**. As a block programming language, it is used extensively in schools to teach students of all ages how to code. Scratch is now adapted for multiple uses, including some robotic applications, such as Vex Code, incorporating machine learning and artificial intelligence, and much more. It is compatible with popular classroom peripherals such as the **Makey Makey**, which is a circuit that interacts with the computer and can be fully controlled with a Scratch program. While it is popular for educational purposes, the power of the programming language cannot be understated and the language itself and its functionalities continue to grow.
- **Java** and **JavaScript**: JavaScript is a scripting language that is used only within browsers. It is used in the making of websites and web applications. Java, on the other hand, is a general-purpose programming language. JavaScript helps us make websites animated or add interactive functionalities to them. Contrastingly, Java is compiled into bytecode and is widely used in the development of Android devices and applications.
- **PHP**: PHP is otherwise known as **Hypertext Preprocessor**. Much like Java, it is a general-purpose programming language. It is widely available, as it is open source. PHP is used in website design and applications and is considered to be easy to learn, yet has many advanced features. PHP can also be used to write desktop applications.
- **SQL**: SQL, or **structured query language**, is a programming language used to interact with data. SQL is domain-specific. It has been around for almost as long as C, making its first appearance in 1974. The main importance of SQL is that it can interact with databases, where other languages are not able to do so.

In computational thinking, we use many different programming languages, depending on what our goals are, what information we have or need, and what our application or software requirements are. Choosing a language is dependent on not just our knowledge of the language, but the possible functionalities of the language.

We will get to work more extensively with Python in this book because of its open source nature, ease of use, and the large number of applications it can be used for. However, Python is not the only option. Knowing about other languages is important, especially for developers.

In this section, we learned about computing and a few of its areas, namely, architecture and programming languages. We also learned about the ISA and its types, along with an introduction to various programming languages. In the next section, we'll look at data types and structures.

Learning about data types and structures

In computer science, data types and structures are two distinct things:

- A **data type** is a basic classification. Some data types include integers, float, and strings.
- **Data structures** use multiple types of data types. They can organize the information into the memory and determine how we access the information.

Let's look at these in more detail in the following sections.

Data types

As mentioned, data types are basic classifications. They are variables that are used throughout a program and can only exist with one classification. There are different classes of data type. We will focus on **primitive** and **abstract** data types for now, but we will revisit this topic as we move through problems and design solutions.

Primitive data types include **byte**, **short**, **int**, **long**, **float**, **double**, **Boolean**, and **char**:

- A **byte** can store numbers from -128 to 127. While these numbers can be stored as integers, or **int**, a byte uses less storage, so if we know the number is between those values, we can use a byte data type instead.
- A **short** is a number between -32,768 and 32,767.
- An integer, **int**, is used to store numbers between -2,147,483,648 and 2,147,483,647.
- **Long** is used to store numbers from -9,223,372,036,854,775,808 and 9,223,372,036,854,775,807.
- A **float** allows us to save a decimal number.
- Decimal numbers can also be saved as **double**, which has more precision than a float.
- **Boolean** values are data types that are either `True` or `False`. So, a variable can be saved such that when its value is printed, the result will be saved as true or false.
- **Char** is used to save a variable as a single character.

We'll look at data structures in the next section.

Data structures

As mentioned under the *Coding theory* section earlier in this chapter, data structures are used to collect and organize data in the most efficient and effective way possible. Data structures can be primitive, such as the built-in data structures in software, or abstract. Primitive data structures can also be defined using programming languages, but they are pre-defined. Some of the primitive data structures include the data types listed in the previous section, such as **chars** and **Boolean** structures.

Abstract data types (**ADTs**) include the information for the structure and design of data types. Abstract data structures include arrays and two-dimensional arrays, stacks, trees and binary trees, linked lists, queues, and more, as mentioned in the *Coding theory* section earlier in this chapter. Lists can contain multiple instances of the same data values. These lists are countable, so we can find how many elements are in the list, reorder them, remove items, add items, and so on. Lists are widely used as linked lists, arrays, or dynamic arrays:

- A **linked list** means that each data element in the list is connected, or points, to the next one, regardless of where they are stored within the memory.
- An **array** is ordered. The elements are read in order to be able to make sense. Think of an array as reading this sentence. You don't read the sentence as "*array an think reading as this of sentence.*" We read the sentence in order, from left to right, not in a jumbled order.
- **Dynamic arrays** can be resized, which is important when choosing a data type.

A **stack** ADT is a collection of elements and has two operations – push and pop. A push is used to add an element to the collection while a pop removes the most recent element.

A **queue** ADT is a linear data structure. As with a stack, we can add or remove elements. However, in a queue ADT, the point of deletion and the point of insertion are done at two different ends.

As mentioned before, the data structures are concrete implementations of data types. How we add or remove elements from a collection, for example, is the data structure.

This can all be slightly confusing, but we will be learning more about them through examples in later chapters. For now, understanding the definitions and simple examples is enough.

Summary

We have learned some fundamentals of computer science in this chapter. We looked at how to convert from binary to base-10. We also explored topics and theories in theoretical computer science. We learned about computing and data types and structures. These sections will allow us to understand the computational thinking process and how to tackle all types of problems presented, starting in *Chapter 2*, *Elements of Computational Thinking*, of this book.

As we delve deeper into the computational thinking world and process, we will need to revisit some of the content of this chapter as we look at problems, search for the best way to solve them, and make decisions about how to write the algorithms.

Problems may have an infinite number of ways to be solved using algorithms. Understanding how processes work and which data structures are most suitable for our problems is imperative in creating the best solutions. Identifying the data types needed for the algorithms and how computers read data will only help us in writing the most effective and efficient algorithms.

In the next chapter, we will learn about the computational thinking process and how to break down problems in order to design our algorithmic solutions.

2
Elements of Computational Thinking

The previous chapter provided some general information about the fundamentals of computer science. In this chapter, we will focus more closely on understanding computational thinking and the elements that make up computational thinking: decomposition, pattern recognition, pattern generalization or abstraction, and algorithm design.

The importance of the computational thinking process cannot be understated. Through the process, we analyze problems to design the best possible solutions based on a set of conditions. Although many algorithms can be written to answer the same questions, using the computational thinking process helps us to determine the optimal path to take to design a clear algorithm that can be generalized.

In this chapter, we will cover the following topics:

- Understanding computational thinking
- Decomposing problems
- Recognizing patterns

- Generalizing patterns
- Designing algorithms
- Additional problems

In order to learn more about computational thinking, we will be looking at the elements through the lens of a problem presented. Keep in mind that we will be doing a deeper dive into the Python programming language in *Chapter 8*, *Introduction to Python*, but you can go back and forth to that chapter as needed while you dive into this and all the other chapters of the book.

Technical requirements

You will need the latest version of Python to run the codes in this chapter. You will find the full source code used in this chapter here: `https://github.com/PacktPublishing/Applied-Computational-Thinking-with-Python/tree/master/Chapter02`

Understanding computational thinking

In its most basic definition, computational thinking is a problem-solving process. Much like with design thinking, the scientific method, and other similar methods, there are a number of steps we go through to find solutions. For example, the scientific method has seven steps. Please keep in mind that there are multiple interpretations of the scientific method out there and some differ in terms of the number of steps. For the purposes of this discussion, we will use these seven steps:

1. **Question**
2. **Hypothesis**
3. **Materials**
4. **Experiment**
5. **Results**
6. **Conclusion**
7. **Communication of findings**

The establishment of the scientific method is a highly debated topic, but most researchers agree that it dates back to the 10th century.

The scientific method made it possible to observe the natural world, create hypotheses to test observations, and develop tests and results through an established process. The method itself has some basis in philosophers such as Plato and Aristotle, who promoted empirical research. However, their methodology was not as developed as what we call the scientific method today.

The computational thinking elements are similar to the scientific method. Computational thinking uses fewer steps to tackle problems associated with programming, where the scientific method is used for experiments. With computational thinking, we generalize the algorithms, while in the scientific method, we are able to reproduce results and generalize the conclusions from samples to populations.

In modern times, we have developed other methodologies depending on the fields of study we pursue and technologies we have developed. Two examples of that are the design thinking process and computational thinking.

Design thinking has five steps or stages:

1. Empathize
2. Define
3. Ideate
4. Prototype
5. Test

We use these aforementioned stages to understand the needs of a client, class, problem, situation, or other circumstance that we need to address. Empathizing with the needs of the user helps us identify and define the problem. The ideation and prototype stages are where we create possible solutions. Testing the possible solutions is the next required step in finding the best possible one. After all the stages, we can go back through the cycle if we need to, since the goal of the design thinking process is not perfection, so additional work can always be carried out. The goal of design thinking is to provide a solution that is effective and plausible for a defined problem. This is not the only viable solution, nor is it perfect.

In computational thinking, we use a similar process that has four elements:

1. **Decomposition**
2. **Pattern recognition**
3. **Abstraction**
4. **Algorithm design**

As with the design thinking process, the problem is not clearly defined in computational thinking. These problems are sometimes referred to as *ill-defined*. We are presented with a set of circumstances and we define or decompose that problem before we start ideating, or creating possible solutions based on patterns we can see. When we think about the computational thinking process, what we are really doing is trying to figure out how we can get a computer to follow a set of steps in order to solve the problem we have been presented with.

Let's take a look at a simple computational thinking problem.

Problem 1 - Conditions

Let's imagine that a raffle at a radio station has contestants select one of two possible winning structures: $250 or the height, in quarters, of the contestant in cash.

A computational thinking problem can be as vague as *Problem 1*, where no question is even being asked. You are given a set of conditions and it is your job to determine what the problem is and find solutions for that problem you yourself have defined. If you think about it, there is no perfect answer for this problem, but there are ways for you to create conditions that work to determine which option is indeed best, depending on the height of the contestant.

To **decompose** this problem, we need to look at what is stated and take into consideration what is not stated. We need rules.

Simply stated, a winner will choose a monetary payout: either $250 in cash or the equivalent of their height in quarters. Those things are stated. But what isn't stated is also important:

- *What is the timeline for the raffle? How many winners are there?*
- *Do we want to track how much we have spent after each contestant has chosen?*
- *Do we want to use a baseline for comparison purposes?*

There are other things that may come to mind, but for now, let's stick to these questions. We are going to assume that the raffle doesn't have a set start or end date and that the radio station may choose multiple winners in a given day – or none at all. These are some of the considerations we will look at when figuring out patterns, generalizing them, and designing the algorithms.

Given all the information about payouts, we still do not have a way to figure out when the payout is greater. *Is it best to choose the $250? Or is it best to choose the height in quarters? Can we create an algorithm that tells us which option is best somehow?* Yes, we can create an algorithm that addresses the entire problem.

The **pattern** for this problem will always be the same: the amount is set for the cash value and the height of a quarter is set, so we can always use math to figure out what the height in quarters converts to money-wise based on someone's height.

We can clearly state the winnings based on each choice if we know a few things. This includes *the choice of cash or choice of height in quarters*. If height in quarters is chosen, we need the following:

- The contestant's height
- The thickness of the quarter

What happens next is part of both **pattern** and **abstraction**. We do not know the choice until each contestant decides, but we can find out what each quarter's thickness is ahead of time. It will be needed later for our algorithm. Each quarter is approximately 0.069 inches, or 1.75 millimeters, thick.

Important Note:
For the purpose of this book, we will be using customary measurements, for example, feet and inches. However, the algorithms for metric measurements will also be provided.

Looking at our problem, we can state the winnings in two ways. The following expressions included for the height in quarters winnings are **mathematical algorithms**. They show the steps needed in order to determine the total winnings given the height of the contestant.

Note that in order to use the customary algorithms, height would need to be given in customary units. In order to use the metric algorithm, height would need to be given in metric units. If a contestant chooses the cash, then the total winnings are simply $250. If the contestant chooses the height in quarters, then the algorithms for both customary and metric units are as follows:

- Total winnings (customary): $(h \div 0.069) \times \$0.25$
- Total winnings (metric): $(h \div 1.75) \times \$0.25$

I like a gamble that is not high stakes. So, I'm going to say that I want to test this out and use my own height. So instead of taking $250, I choose to find out what my height would be in quarters. I am 5'10" tall. Let's figure out how many inches that is. Since we have 12 inches in a foot, the algorithm for the total height is as shown:

$$5 \times 12 = 60 \text{ inches}$$

But I said I am 5'10", so we will need to add those 10 extra inches:

$$60 + 10 = 70 \text{ inches}$$

Now, let's use the mathematical algorithm we defined earlier, $(h \div 0.069) \times \$0.25$, in order to find out how much I'd win:

$$(70 \div 0.069) \times \$0.25 \approx \$253.62$$

I used the ≈ symbol instead of = because ≈ means this is an approximation. Since I rounded off, I wanted to make sure I showed it was the best approximation, not the exact number.

Maybe you are done with the problem now, but in computational thinking, we have to go through abstraction and design an algorithm that will apply to all instances. We can create a very simple program that uses simple input from a user, or we can create a more complex program that provides not just the basic total, but maybe the sums, a graphic, or whatever else we find relevant to our scenario and that applies to all cases.

We will be designing those algorithms more once we have learned about each part of the computational thinking process in more depth. We will even go back to this problem to show how to create that algorithm for the computer to run for us. We can create an algorithm that lets us use someone's height to make a decision about which winnings to use.

Or, as mentioned earlier, we could write a baseline using $250 as the winnings for every contestant and then input what each contestant has chosen in order to see whether we are below or above the $250 baseline. We can aggregate those results, which means to continue adding them to see where we end up once the radio station stops the raffle. We could even have a graphic that shows us where we are over time, if contestants were choosing differently the longer the radio station ran the raffle, and so on.

In this section, we learned about the elements of computational thinking. We also looked at a computational thinking problem.

For now, however, let's continue to look at the elements of computational thinking – that is, decomposition, pattern recognition, pattern generalization and abstraction, and algorithm design, in more depth.

Decomposing problems

Decomposition is the process of breaking down data. It can include a number of processes or steps necessary in order to solve the problem. By decomposing the problem, we can identify the components, or smaller parts, of the problem before we generalize the pattern.

Through decomposition, we can identify and solve one case in order to then generalize those steps to all possible instances of the problem. In order to really understand decomposition, we will need to go back to our problem stated earlier, which, simply stated, is asking the question: *Will my height result in more money if I take my height in quarters or should I take a $250 payout?* We can state that we want to know one instance and do that problem mathematically one time, such as solving the problem for my own height only. However, we may need the information for other instances. We could create a program that just identifies which option, $250 or your height in quarters, would be best. Or we could take into consideration some of the following scenarios, which would mean a different algorithm:

- We could check the option given the height but also add each item to a list in order to track all decisions.
- We could also need the array and the sum of the elements in that list to track spending throughout the contest.
- We could also compare the sum to a baseline, using $250 as a base for each of the individuals.
- We could also use all of the elements, such as the list, the sum, the comparison, and a visual graphic display to better understand and track the results.

As you can see, the algorithm will depend on what it is exactly we want to track or answer for this problem. Our algorithm could be a simple yes or no type of problem, where we'd just check which option is best, or it could be a much more robust algorithm, with data and visual representations for the data tracked. Now let's take a look at how we work to find patterns in our problems.

Recognizing patterns

Pattern recognition is the process of finding similarities, or patterns, once we go through the decomposition of problems. In *Problem 1*, we were shown a problem where a contestant would win $250 or choose to take their height in quarters. This will be the same for every contestant. The only difference is that the total value changes depending on the height of the person.

In this section, let's take a look at a different problem in order to better understand pattern recognition.

Problem 2 - Mathematical algorithms and generalization

Imagine you are preparing a party for a soccer team. This is a community team, so there are always between 12 and 15 children that stop by. You want to place an order for the food you will need. You know it will cost you \$12 per child from the catering company you will be using. Now, let's break down the problem:

- **Decomposition**: I know we have between 12 and 15 children. We also know there is a cost of \$12 per child. Our problem can be thought of as a question: *How can we estimate the cost?*
- **Pattern recognition**: You know the number of children, k, is between 12 and 15. You know it is going to cost \$12. If I had 5 children, for example, the cost would be $\$12 \times 5 = \$60$.
- **Pattern generalization**: The number of children is not known, so we will use the variable k for that unknown value. That way, we can find out the total cost no matter how many children we have. We are generalizing from one case, 5 children, to all cases, k children.
- **Algorithm design**: We will write the mathematical algorithm for now. We will be working more with programmable algorithms coming up in *Chapter 3*, *Understanding Algorithms and Algorithmic Thinking*. We will be revisiting these mathematical algorithms there as well. The total cost will be given by the equation $\$12k = T$, where T is the total cost and k is the number of children.

As you can see from the preceding problem, pattern recognition is important in order to find a generalized pattern and write our algorithm. Now, let's look more closely at pattern generalization.

Generalizing patterns

Once we have recognized our pattern, we need to go through **pattern generalization** and **abstraction**. That is, we want to make sure that the solution we come up with can be used for multiple instances of the problem we have identified. Pattern generalization can be something as simple as writing a basic linear mathematical algorithm, like we did for the cost of a party, where the cost per child was \$12. So, the cost for any number k of children would be given by $12k$. But pattern generalization can be much more than that.

If we go back to *Problem 1*, where you could choose $250 or you could choose your height in quarters, our pattern **generalization** would allow us to check for anyone's height against the $250 in order to determine whether you would get more money by choosing the cash option or by choosing the quarters.

Abstraction lets us focus on the things we need and discard things we do not need in order to create the best algorithm for our problem. Now, depending on what we decide we need, we can add or remove some conditions.

For example, if I am a contestant, I only really want to know what option gives me more money. I do not care about total wins, who's choosing $250, who's choosing height, and so on. But if I'm the radio station, I may want to know the sum, the comparison to the baseline, and much more. I would have to choose that baseline and maybe even graphically show what has happened over time. That is all part of the abstraction process. When you are solving a computational thinking problem, you are also determining what matters and what does not matter to your solution and algorithm.

In the simplest form of this problem, if you are a contestant, you want to know what your best possible case for winnings is. If you choose $250 but your height makes it so that your height in quarters is more than $250, you would want to know. If you are working at the radio station, you may want to track more than just each winning individually. Abstraction allows you to adapt to all cases, from doing one mathematical problem to creating an algorithm that could track all the choices from all the contestants. Let's look now at how we can create those algorithms.

Designing algorithms

As previously mentioned throughout this chapter, an *algorithm* is a set of instructions. When we are writing a computer program, the algorithm is a set of instructions given to the computer so it can provide a solution to a posted problem. We have been sticking to mathematical algorithms so far only because we haven't done a deeper dive into Python yet. However, we will now translate some of the algorithms into Python programming.

First, let's take a look at *Problem 1*. Here, we had a situation where you can win $250 or your height in quarters. Assuming it is you who's competing, you would want to know which option gives you the most in winnings.

Let's take a look again at our mathematical algorithms from earlier in this chapter:

- Total winnings (customary): $(h \div 0.069) \times \$0.25$
- Total winnings (metric): $(h \div 1.75) \times \$0.25$

Remember, if you are using your height in customary units, you'll use the first algorithm. If you are using metric units, you'll want to adapt the program accordingly.

When we are programming, we need to define our variables. In this case, `h` is the variable we are using for height. But think about it; your height may not change if you're an adult, but for the sake of argument, we will assume it won't always be the same. So, we will need whoever wants to know what the best option is, $250 or their height in quarters, to *input* their height so that the program will provide them with the answer.

Input is something the user can enter. So, when we define our variable, we are going to ask for input. A good practice in Python and any other language is not to just ask for the input with no guidance. That is, we want to tell the user the question they are answering. For example, I can write the following code to ask a user for their height input:

```
h = input("Enter your height in inches: ")
```

The preceding code will ask the user to enter some input. It also asks that the user enter the information in inches. If you were using metric units, you would state that instead.

We also saved the information as the variable `h`. But we haven't done anything with that variable yet.

We can just do the basic math and print out the value we get based on height:

```
h = input("Enter your height in inches: ")
total = (int(h)/0.069)*0.25
print(total)
```

Notice in the preceding snippet that we used `int(h)` in the definition of the `total` variable. We converted the `h` value to an integer so we could perform a mathematical operation using that variable. When we asked for the input, the variable was saved as a string, which is why we need to convert it to be able to use it.

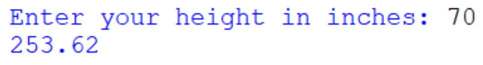
Running the previous code with my height, which is 70 inches, yields the following result:

```
253.62318840579707
```

It would look much better if we rounded that answer, and Python has a way for us to do that easily. If we adjust the print code shown as follows, our answer will result in `253.62`:

```
h=input("Enter your height in inches: ")
total = (int(h)/0.069)*0.25
print(round(total,2))
```

When I run this program, here's what the window looks like:



```
Enter your height in inches: 70
253.62
```

Figure 2.1 – Python program output

But sometimes we want the code to do more. Let's remove that `print` command and create some conditions. In the next few lines, we will use the value provided to make some comparisons. For example, we can ask the computer to check some things for us. There are three possibilities:

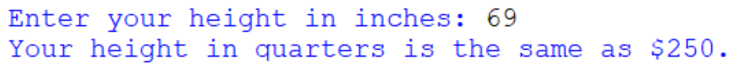
- Our height could yield exactly the same as $250.
- Our height could yield less than $250.
- Our height could yield more than $250.

Now, I will ask the computer to tell me what to do based on those conditions. We will need an **if-elif**, **else** statement for this. These are conditions that we will test in order to receive better output. We will test whether the total is the same as $250. Else, if the total is less than $250, we will want the computer to do something (that is our `elif` statement). Finally, in all other cases, we will use the `else` command:

```
h=input("Enter your height in inches: ")
total = (int(h)/0.069)*0.25
total = round(total,2)
if total == 250:
    print("Your height in quarters is the same as $250.")
elif total > 250:
    total = str(total)
    print("Your height in quarters is more than $250. It is $"
+ total)
else:
    print("You're short, so choose the $250.")
```

Let's see what some test cases look like.

First, let's use a height of **69** inches:

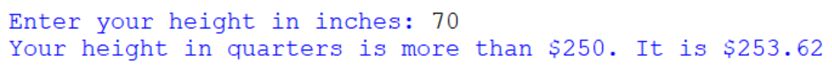


```
Enter your height in inches: 69
Your height in quarters is the same as $250.
```

Figure 2.2 – Case 1: Height yields $250

So, anyone who is 5'9" cannot go wrong with either choice, since they'll always end up winning $250.

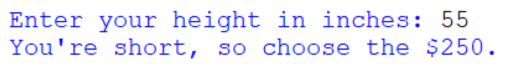
Now, let's look at my height, **70** inches:



```
Enter your height in inches: 70
Your height in quarters is more than $250. It is $253.62
```

Figure 2.3 – Case 2: Height yields more than $250

Finally, let's look at a height less than 69 inches, so let's use **55** inches:



```
Enter your height in inches: 55
You're short, so choose the $250.
```

Figure 2.4 – Case 3: Height yields less than $250

Notice that we can adjust the code to say what we want it to. I chose to use full sentences to establish the results, but you could adapt and adjust the code as needed to suit your needs or preferences. Some of the code may be challenging at this point, but we will be doing a deeper dive with the Python programming language in *Chapter 8*, *Introduction to Python*.

As you can see, we have three algorithms that provide us with the same kind of information. One is more robust than the other two, but how complex or simple our algorithm is depends on what we need from it. If you were holding this raffle again later on, you might have forgotten what the algorithm was, how you wrote it, or what everything meant. However, with the last code, you get a lot of information just by running it, which is more helpful than the first two. You can also add all that information as comments within the code too, but we will talk about those in *Chapter 8*, *Introduction to Python*.

Also keep in mind that we ran this as if we were the contestants. While that is helpful, you may want to consider what changes you would make if you were the radio station. You could write some code that saves all the instances that are run so that you can then check and add all the winnings. You could even calculate that sum through code. Since that code is a bit more complex, we will be touching on it more throughout the book and, more specifically, in *Chapter 8*, *Introduction to Python*.

Now, let's take a look at a few more problems and respective algorithms in order to get more comfortable with the computational thinking process.

Additional problems

Throughout this section, we are going to take a look at additional problems. For *Problem 2*, we will go right into the algorithm, as we went through the other steps in the problem earlier in this chapter. The next problem will have the entire computational thinking process as well.

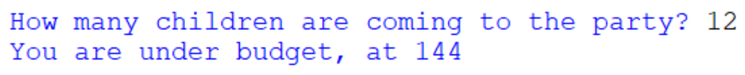
Problem 2 - Children's soccer party

Earlier in this chapter, we were planning a party for a soccer team, where there was a cost of $12 per child. We stated that the number of children was unknown, so we will use the variable k to designate the unknown quantity. We also stated that we had a mathematical algorithm, $T = 12k$, that gave us the total cost, T, of k children. Let's add a condition here. If we had a budget of $200, we'd want to know if we are over, under, or right on that budget.

We can use Python to write an algorithm for this situation as well:

```
k = int(input("How many children are coming to the party? "))
T = 12 * k
if T == 200:
    print("You are right on budget, at " + str(T))
elif T <= 200:
    print("You are under budget, at " + str(T))
else:
    print("You are over budget, at " + str(T))
```

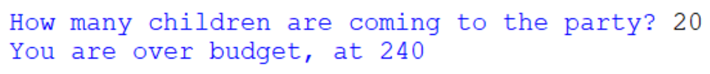
Let's test some cases in order to verify that our code is working:



```
How many children are coming to the party? 12
You are under budget, at 144
```

Figure 2.5 – Case 1: 12 children attend the party

Great! We are under budget if 12 children attend. So, *what if 20 children attend?* Let's have a look:



```
How many children are coming to the party? 20
You are over budget, at 240
```

Figure 2.6 – Case 2: 20 children attend the party

We do not have enough money for 20 children!

As you can see, the program provides us with some information about the total and whether we are over or under budget. As with any algorithm, this isn't the only way we could write the program in order to get this information. Try your hand at different algorithms to solve this simple problem or add a few conditions of your own and code them. Practice and adding conditions will allow you to get more comfortable designing and writing algorithms.

Problem 3 - Savings and interest

Now we have a new problem. A bank pays compound interest at a rate of x% per month. *What will be the payout after a number of years if you deposit any given amount?*

Let's **decompose** this problem. First of all, we know interest is compounded monthly. Let's talk about compound interest. The interest on an investment is the percentage that is paid out in a time period. Compound interest means that the interest pays out on the initial amount plus interest each time. Compound interest is a **pattern**. In fact, a formula exists for it.

The thing I do not know is what percentage the bank is paying out, or the amount deposited, or the number of years it'll be deposited for. So, we will need to write a program that addresses all of the possibilities. This is **pattern generalization**. What we do know is that the interest is compounded monthly. There is actually a mathematical formula for this:

$$A = P\left(1 + \frac{r}{n}\right)^{nt}$$

Let's talk about the terms from the preceding equation:

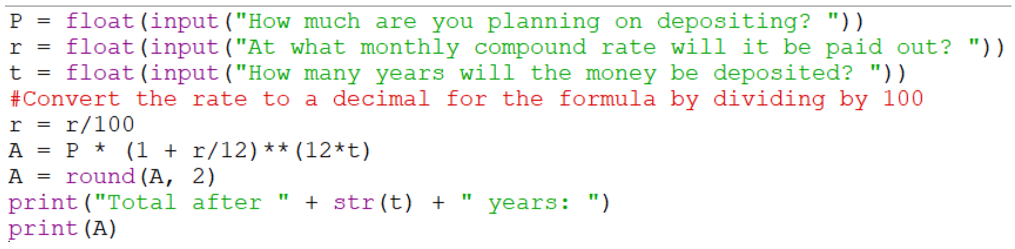
- A is the total amount.
- P is the principal amount, that is, the initial deposit.
- r is the interest rate (keep in mind that for 3%, the interest is written as 0.03, for example).
- n is the number of times interest is compounded per year.
- t is the number of years the deposit goes untouched.

Because there is a mathematical algorithm, we can now create a program for this using the formula. However, we will need to make sure whoever runs the program knows what it is we are asking for with regard to all the inputs. We are asking for a few things:

- *What amount is being deposited?*
- *At what rate is the bank paying?*
- *For how many years will the money be deposited?*

We do know the *n* in the formula. That *n* is 12 because this is monthly compound interest. That means it will compound 12 times each year. So, *n* = 12.

Now it is time to write the program for this:



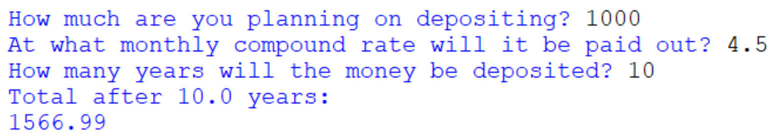
```
P = float(input("How much are you planning on depositing? "))
r = float(input("At what monthly compound rate will it be paid out? "))
t = float(input("How many years will the money be deposited? "))
#Convert the rate to a decimal for the formula by dividing by 100
r = r/100
A = P * (1 + r/12)**(12*t)
A = round(A, 2)
print("Total after " + str(t) + " years: ")
print(A)
```

Figure 2.7 – Sample program for compound interest

The preceding screenshot shows us the Python program for compound interest. Notice the comment, which is preceded by the # symbol. It states that we needed to convert the rate to use the formula. We would have otherwise obtained an incorrect total. In addition, we used float here because we need to use decimals. The integers, or `int`, would not give us the information we needed. Also, we rounded the total to two decimal places. That is because we use two decimal places when we talk about money. The text for the algorithm shown in *Figure 2.7* is included as follows:

```
P = float(input("How much are you planning on depositing? "))
r = float(input("At what monthly compound rate will it be paid
out? "))
t = float(input("How many years will the money be deposited?
"))
#Convert the rate to a decimal for the formula by dividing by
100
r = r/100
A = P * (1 + r/12)**(12*t)
A = round(A, 2)
print("Total after " + str(t) + " years: ")
print(A)
```

Using the code from the compound interest algorithm, we can run any possible instance for compound interest if we have the initial amount, the rate, and the number of years we will be depositing the amount for. The output of the program given an initial deposit of $1,000 at a rate of 4.5% for 10 years is provided as follows:

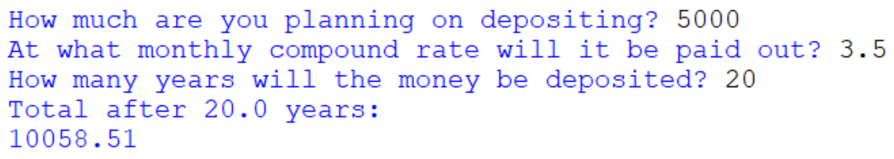


```
How much are you planning on depositing? 1000
At what monthly compound rate will it be paid out? 4.5
How many years will the money be deposited? 10
Total after 10.0 years:
1566.99
```

Figure 2.8 – Sample 1: Output for compound interest

As you can see, the total after 10 years for a deposit of $1,000 at 4.5% compounded monthly is $1,566.99.

Let's test the program one more time. This time, we will deposit $**5,000** at a **3.5**% rate compounded monthly for **20** years:



```
How much are you planning on depositing? 5000
At what monthly compound rate will it be paid out? 3.5
How many years will the money be deposited? 20
Total after 20.0 years:
10058.51
```

Figure 2.9 – Sample 2: Output for compound interest

Our total after 20 years would be $10,058.51. *That means our money doubled!*

Having this calculator program allows us to only calculate for interest compounded monthly. We can create a new program to calculate interest compounded at any rate: monthly, annually, bi-monthly, and so on. You can try playing with the code in order to create your calculators. These are helpful if you want to know what to expect when investing money or depositing in savings accounts. For example, you can determine how long it would take for your deposits to reach a certain amount. Say you wanted $50,000 for a college education for your children. In this case, you could figure out how much you would need to deposit in order to have that amount in 18 years, when they'd most likely be ready to go to college.

Summary

In this chapter, we have gone through the computational thinking process. We learned about the four major elements of computational thinking: decomposition, pattern recognition, pattern generalization, and algorithm design. We also learned that problems in computational thinking are not always clearly defined. It is up to us to interpret and decompose the information so we can find the patterns. Once we find the patterns and define what we'd like to generalize, we can then design algorithms in order to solve our problems.

We also learned that algorithms and solutions are not unique. There are multiple ways to write an algorithm for each problem we encounter. The computational thinking process allows us to explore problems in multiple ways and create solutions that align with our own interpretations and needs.

In the next chapter, we will be learning about algorithms and algorithmic thinking more in depth, as they are the product of all computational thinking problems. In order to better understand algorithms, we will take a look at Boolean operators and learn how to identify and address errors in logic processing. We will write algorithms for problems presented, as well as analyze given algorithms for possible errors in processing.

3
Understanding Algorithms and Algorithmic Thinking

In this chapter, we will focus more closely on understanding algorithms and algorithmic thinking. While this is the last step in the computational thinking process, it is critical that we understand how algorithmic thinking helps us plan and understand our problems better. That is, the more we practice algorithmic design and algorithmic thinking, the easier it is to understand, decompose, and recognize patterns when problems are presented to us.

In this chapter, we will cover the following topics:

- Defining algorithms in depth
- Designing algorithms
- Analyzing algorithms

After reading this chapter, you'll understand algorithms better. So, we'll begin by analyzing the definition of algorithms again, which we covered previously in *Chapter 2*, *Elements of Computational Thinking*, as well as how to design mathematical and computational algorithms.

Technical requirements

You will need the latest version of Python to run the codes in this chapter. You will find the full source code used in this chapter here: `https://github.com/PacktPublishing/Applied-Computational-Thinking-with-Python/tree/master/Chapter03`

Defining algorithms in depth

As we mentioned in *Chapter 2*, *Elements of Computational Thinking*, an **algorithm** is simply a set of instructions. We use instructions in everyday life, sometimes consciously, sometimes unconsciously. Think about the routines you follow in the morning, for example. The alarm clock sounds. *What do you do next? Do you go prepare coffee? Shower? Brush your teeth first?*

Most of us follow the same steps every single morning. You could say we've programmed ourselves to follow those steps. Now think of a time your schedule changed and your routine was different. I know I've had to stop and regroup many times because my *program* no longer works. I can't wake up at 6 a.m. for a 5 a.m. flight, for example.

Algorithms for computers are similar in that we need to reprogram the set of instructions if a set of conditions has changed. The programs can only go as far as we have stated parameters for them. Most programs cannot adjust or adapt to any new information that is not previously coded into it. That said, **machine learning** and **artificial learning** are evolving. We're not talking about those kinds of programs, but even in those instances, we'd still need to adjust those programs to do what we need them to.

To design algorithms, we need to make sure that they meet some specific characteristics:

- **They are clear and unambiguous.**
- **They have inputs that are well defined.**
- **They have outputs that are well defined.**
- **They have finiteness.**
- **They are feasible.**
- **They are language-independent.**

Let's look at each of the characteristics in the preceding list and define them.

Algorithms should be clear and unambiguous

An algorithm is clear and unambiguous when every one of the steps can easily be understood, is easily defined, and has inputs and outputs that are also clear and well defined. There should also be only one meaning for each component of the algorithm.

Algorithms should have inputs and outputs that are well defined

The **inputs** for an algorithm can be user-provided, meaning that the user of the program enters the data. Input can also mean something that is defined within the program. This means that I may include a variable with a set value already provided.

For example, if I need a user to tell me the number of tickets they are purchasing, I can write the algorithm to ask for that input. I can also give that input as a defined variable with a given value already. An algorithm does not always require an input – zero-input algorithms do exist – but when the algorithm requires input, defining that input is important. An example of an input is asking for a user's name in a program. Think about modern video games. Many of them will prompt the user for a name with phrases such as, *"Hello traveler. What is your name?"*

As a user, I'd enter `Sofia` when given that prompt, which gives me the following:

```
"Hello Sofia. Welcome to the adventure!"
```

As you can see, the game will then produce an output and uses my name in that output.

This final line is the **output** of the program. I can write a simple program to ask that question in Python as well:

ch3_nameprompt.py

```
name = input("Hello traveler. What is your name? ")
salutation = "Hello %s. Welcome to the adventure!" % name
print(salutation)
```

Notice that we used `%s` and `%` symbols here. The syntax here is what we call an f-string. We use the `%s` syntax to let the program know where we want to insert the information and then we call that information by using the `%` symbol. In this case, we saved the input to the `name` variable, then called it in the `salutation` variable.

When run, the program looks like this:

```
Hello traveler. What is your name? Sofia
Hello Sofia. Welcome to the adventure!
```

This simple algorithm allowed us to save the name as a variable. That variable was used only once in the output of this simple code. However, in a game, that name variable may be used in multiple instances, such as during conversations with characters within the game, and so on.

The **output** of a program is the information that leaves a system, that is, the product of your program. Given some information or code, the output is what is produced from the instructions in the program.

Algorithms should have finiteness

An algorithm has to have **finiteness**. This means that an algorithm must end. Let's look at a situation where an algorithm would not end. **I don't recommend writing this or running it!** Nonetheless, let's look at the steps we would take to create this algorithm:

1. Define a variable, `i`, and set it as equal to `0`:

   ```
   i = 0
   ```

2. Increase the value by `1`. There are a few different ways we can do that:

   ```
   i = i + 1
   i += 1
   ```

   Both of the preceding lines of code will increase the value of `i` by `1`.

3. *Add an error!* We're about to create an error in finiteness. Again, I'm only doing this to prove a point, but this is an error you want to avoid:

   ```
   i = 0
   while i >= 0:
       i += 1
       print(i)
   ```

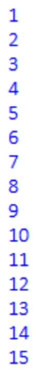
In this algorithm, I'm telling the program to continue to increase `i` by `1` so as long as it is greater than `0`, then the computer is supposed to print the value. This will just continue to go on forever and ever, without stopping, because the condition will always hold true as given. So, the output for the program will begin at 1, but will continue printing the next item in the sequence as 2, 3, 4, 5, and so on. The program simply has no way to end.

Now, a similar program may be done given a few different conditions. Let's say we want to print all the values of our addition, but only so long as `i` is less than 15:

ch3_finiteness.py

```
i = 0
while i < 15:
    i += 1
    print(i)
```

The preceding program is a terminating program. It now only works for all values while `i` is less than `15` (not including 15). We will get an output shown as follows:



```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
```

Figure 3.1 – Output for finiteness code

I know I said this program did not include 15. It doesn't. Since this happens while `i` is less than 15, the last value it will evaluate for is 14. However, it says that while the value is less than 15, we increase it by 1 (`i += 1`). So, when `i` is 14, the printed value is 14 + 1, or 15. Finiteness allows the program to terminate.

Algorithms have to be feasible

An algorithm also has to be **feasible**. To be feasible, an algorithm needs to be possible with the available content and resources. When writing algorithms, we have constraints, or conditions we may write into the steps. If there is no way to meet all the constraints, then the algorithm isn't feasible. Think of the two conditions, given as follows:

- It is 3:00 p.m.
- It is 5:00 p.m.

If we set both of these constraints on a variable, for example, it would not be possible. It cannot be both 3:00 p.m. and 5:00 p.m. at the same time. This is what we call **infeasible**. While the algorithm can continue, we're still creating a problem by making these two things true at the same time. Some constraints will never be met, so the algorithm is considered infeasible. There has to be a way for the algorithm to meet all constraints in order to be feasible. In addition, if an algorithm is written to depend on future technology, for example, it is also considered infeasible.

Algorithms are language-independent

Finally, an algorithm must be **language-independent**. The set of instructions in an algorithm should be written as simply as possible. A good algorithm will be such that it can be written in any language easily and produce the same output.

In this section, we learned about algorithms and the characteristics needed to design them. Keeping in mind the characteristics of a good algorithm will allow us to avoid errors and create working algorithms for whatever problems we are presented with, let's now take a look at how to design some algorithms.

Designing algorithms

When designing algorithms, order matters. There are hierarchies that matter when we are working with programming languages. That includes when we are working with Python. Think about this as the order of operations in mathematics. If you recall, we use the mnemonic **PEMDAS** to remember the order of operations in mathematics. **PEMDAS** stands for **Parentheses**, **Exponents**, **Multiplication/Division**, and **Addition/Subtraction**.

I write **Multiplication/Division** together like this because multiplication and division hold the same weight. That is, multiplication does not necessarily need to happen before division. If I have a division first and then a multiplication from left to right, then the division happens first. The same is true for addition and subtraction. Neither has more weight than the other, so we perform them in order of appearance from left to right.

Let's write a mathematical algorithm for a problem. We'll look at an algorithm in a food setting. And yes, I know I write about food and food algorithms a lot. I love food almost as much as I love code.

Problem 1 – An office lunch

An office is ordering catering for employees. Employees were given two lunch options: sandwiches or salads. Each sandwich meal costs \$8.50, while each salad meal costs \$7.95.

Office lunch mathematical algorithm

The number of employees who choose each option is unknown. Let's use some variables to help us in designing the mathematical algorithm. Let's use s for the number of sandwiches and b for the number of salad bowls. And I know what you're thinking, those two variables aren't very helpful if you come back to this problem a while from now. But we'll talk about that in a second. For now, let's just write what our total cost, c, will look like:

$$c = 8.5s + 7.95b$$

This is a simple mathematical problem that requires two unknown variable inputs, s and b, in order to get our total, c. Now let's look at a different version of the same lunch scenario.

Office lunch Python algorithm

Now let's think about a few more considerations when writing the program. As we design a Python algorithm for this problem, we'll need to think about two perspectives: the programmer and the user.

Sometimes we're both the programmer/developer and the end user for our programs, but many times, we'll write or develop content for someone else to use. It is important that we keep those considerations in mind because it may affect how we write our program and define our variables. In addition, if we're writing a program as part of a company, others may need to go and edit our programs at some point.

That means we need to write the program in a way that others will be able to understand. Our variables should be easily understood, so writing a simple one-letter variable may make it harder for another programmer or user to understand. Let's look at a program for *Problem 1*. Recall that in that problem, we're trying to determine the final cost for an office lunch for employees given two possible options:

- \$8.50 for a sandwich meal
- \$7.95 for a salad meal

Let's create the program for this problem using Python. Let's clarify some variables first. We'll want to use full words or a series of words separated by _ to define these variables. Before we start, you may want to recall that for **Python variables**, some rules need to be followed so as not to cause an error:

- Variables must start with a letter or an underscore (_).
- Variables can only contain letters, numbers, and underscores.
- Variables cannot start with a number.
- Variables are case sensitive (*alpha* is not the same variable as *Alpha* or *ALPHA*).

For *Problem 1*, we need three variables:

- The total cost of the lunch
- The number of sandwich meal lunches
- The number of salad meal lunches

Now we need to name them:

- `total_cost` = the total cost for all lunches
- `number_of_sandwiches` = the total number of sandwich meals ordered
- `number_of_salads` = the total number of salad meals ordered

The important thing here is that those variables are easily read and easily understood. I should make a note that I am partial to lowercase variables when programming. I do have some exceptions for when I like to use capital letters, but you'll see many examples with only lowercase letters and underscores. I found a long time ago that even when capital letters made sense to me at the time I was writing a program, I'd later forget which letters were capitalized, which was just an added headache that could be avoided if I just used lowercase letters instead.

In addition, some programmers eliminate the underscores and use variables such as `numberofsandwiches` or simply `sandwiches`, for example. Both of those are acceptable, of course, and the simple `sandwiches` will make it easier to write some of the code. There are both pros and cons to doing this, however. If someone else is looking at the program, readability will be important. Like I said, I am partial to clear, lowercase variables and the use of underscores, but it is up to every programmer to make that choice themselves.

Now that I have defined my variables, I can begin to write my program. *What will I need to ask the user for?* I need **inputs** from the user for both the number of sandwiches and the number of salads. What I want as an **output**, or what the user will want as an output, is the total cost of the lunch. To ask for input from the user in Python, we need to use the `input` command. However, we also need to remember that since we are using this number in an algorithm that uses a `float` number (decimals are float characters), we need to convert the number provided to `integer` or `float`. Employees will not be able to order half a salad, so we can safely save them as integers, or `int`. As a reminder, comments in Python start with a # symbol. Write the code in IDLE, shown as follows:

ch3_officelunch.py

```
#Ask the user for the number of sandwich meals ordered and save
as variable.
number_of_sandwiches = int(input("How many sandwich lunches
were ordered? "))

#Ask the user for the number of salad meals ordered and save as
variable.
number_of_salads = int(input("How many salad lunches were
ordered? "))

#Create total_cost variable and save the algorithm for total
the new variables.
total_cost = 8.50 * number_of_sandwiches + 7.95 * number_of_
salads

#Print the total cost. Don't forget to convert the total_cost
to string.
print("The total cost for the employee lunch is $" + str(total_
cost) + ".")
```

When running the code, the user can enter the number of each of the options for the office lunch. The code first asks the user for the number of sandwiches like so:

```
How many sandwich lunches were ordered?
```

The code will then ask for the number of salad lunches and provide a total cost. The following sample takes an input of 12 sandwich lunches and 23 salad lunches, which would be a total cost of $284.85:

```
How many sandwich lunches were ordered? 12
How many salad lunches were ordered? 23
The total cost for the employee lunch is $284.85.
```

Now let's take a look at a similar problem, but from a different perspective.

Problem 2 – A catering company

Let's say you start a simple catering company. You begin only selling two options, a sandwich meal for $8.50 and a salad meal for $7.95. You can create a program that stores these options using a Python dictionary.

You can find additional information on the Python programming language and dictionaries in *Chapter 8*, *Introduction to Python*, but we'll define a Python dictionary here as well. A dictionary is used when we want items that are unordered, can be changed, and are indexed. Here's an example of a dictionary for our catering company in Python:

ch3_cateringdict.py

```
catering_menu = {
    "sandwiches": 8.50,
    "salads": 7.95
    }
print(catering_menu)
```

Now, dictionaries are common and very useful for various reasons: primarily, that they are easy to read and they provide a way to change data as required.

When printed, the dictionary code looks like this:

```
{'salads': 7.95, 'sandwiches': 8.5}
```

Now that you have a dictionary, let's talk about its usefulness to your catering company. Let's say that there is a cost increase for your salad ingredients that you want to account for by changing the price of the salads. You can do so in a few different ways. You can change it in the original program, since it is so short, or you can just tell the program what you want to change based on the key. This is important because you may have two items for sale now, *but what happens when your menu options become much wider? Would you want to search for each item every time you change a price?* Python makes it easy to identify what you want to change and then change it.

To do so, you can use the following code:

```
catering_menu["salads"] = 9.50
```

Your new code in Python looks like this:

ch3_cateringdict2.py

```
catering_menu = {
    "sandwiches": 8.50,
    "salads": 7.95
    }
catering_menu["salads"] = 9.50
print(catering_menu)
```

When printed, the new value for the salads will be shown:

```
{'salads': 9.5, 'sandwiches': 8.5}
```

But, what happens if you want to add a menu item? Say you want to add a soup option for $3.75. In this case, you can add the menu option to your dictionary by using a simple line of code, as follows:

```
catering_menu["soup"] = 3.75
```

When you put it all together, the initial code and the changes would look like the following code block. Notice that you have the initial dictionary, then the two changes below that. When you print the dictionary, it will include all changes along with the addition of the soup option:

ch3_cateringdict3.py

```
catering_menu = {
    "sandwiches": 8.50,
    "salads": 7.95
    }
catering_menu["salads"] = 9.50
catering_menu["soup"] = 3.75
print(catering_menu)
```

Now that you have added the `soup` item, you can print your dictionary to see your full menu:

```
{'soup': 3.75, 'salads': 9.5, 'sandwiches': 8.5}
```

We can use the information within the dictionary to create more robust programs, such as an online menu, an ordering menu option, and much more. In this section, we learned about designing an algorithm with the help of two problems.

We will take a look at more development using Python through additional problems in the following chapters of this book, especially in *Section 3*, *Data Processing, Analysis, and Applications Using Computational Thinking and Python*. For now, we'll move on to analyzing some algorithms.

Analyzing algorithms

As mentioned previously in this chapter, when we design algorithms, they should meet the following characteristics:

- They are clear and unambiguous.
- They have inputs that are well defined.
- They have outputs that are well defined.
- They have finiteness.

- They are feasible.
- They are language-independent.

In addition to those characteristics, when we are looking at algorithms and analyzing them, we want to make sure we ask ourselves some questions:

- *Does the algorithm do what we want?*
- *Does the output make sense?*
- *Is there another way to get the same information in a clearer way?*

There are many more questions we can ask ourselves when analyzing algorithms, but for now, let's take a look at some algorithmic solutions and analyze them based on the aforementioned characteristics and questions.

Algorithm analysis 1 – States and capitals

A student has created an algorithm that includes a list of US states and the capitals for each of those states, but only those states she has already studied are included. Her algorithm is shown as follows:

ch3_statecapitals1.py

```
Ohio = "Columbus"
Alabama = "Montgomery"
Arkansas = "Little Rock"

print(Ohio)
```

The program is simple, yet not easy to use, nor helpful when run. *Does it contain the information needed?* Yes. *Can we organize it in a different way so we can call the information in other ways?* Yes.

Think about states and capitals as key pairs. We can use a dictionary to store the information. You may recall from earlier in this chapter that a dictionary can be adjusted and adapted easily, adding a new key with a simple line of code. Let's first convert the information in the previous code into a dictionary:

ch3_statecapitals2.py

```
state_capitals = {
    "Ohio" : "Columbus",
    "Alabama" : "Montgomery",
    "Arkansas" : "Little Rock"
    }

print(state_capitals["Ohio"])
```

Notice that we can now access the information for the state capital by simply giving the state name. The output for this code is simply `Columbus`. *But what if you just want to run the program and ask for the user to input a state of their choosing?* We can also write that in a line of code with the existing dictionary. Take a look at the following code:

ch3_statecapitals3.py

```
state_capitals = {
    "Ohio" : "Columbus",
    "Alabama" : "Montgomery",
    "Arkansas" : "Little Rock"
    }

state = input("What state's capital are you looking for today?
")
capital = state_capitals[state]


print("The capital of " + state + " is " + capital + ".")
```

In this code, the user enters the state for which they want to find the capital. This is helpful, as you can just run the code each time without having to go into it to change the line of code to be printed, which we had to do with the algorithm in the `ch3_statecapitals2.py` file. The code, when run, looks like this:

```
What state's capital are you looking for today? Alabama
The capital of Alabama is Montgomery.
```

Now let's look at the need for the algorithm in the first place. The student wants to continue to add states to the program. With this program, since it is dictionary-based, she can simply add a line of code when she needs to add another state. For example, if she wanted to add the state of **Iowa**, whose capital is Des Moines, she'd need to use the following code:

```
state_capitals["Iowa"] = "Des Moines"
```

Take a look at the following code block. Note the placement of the code within the program. It is important that we place that new code before the new variables, otherwise, if you try to run the program and input `Iowa`, the code will return an error rather than providing the capital of Iowa.

In algorithms, logic is extremely important. We cannot use a value we have not defined in variables that have already been used. That is, if the variables `state` and `capital` are used before identifying the new value for `Iowa`, then the code ends with an error when the input is `Iowa`. However, if we add the key pair values before we run those two variables, the code runs as expected:

ch3_statecapitals4.py

```
state_capitals = {
    "Ohio" : "Columbus",
    "Alabama" : "Montgomery",
    "Arkansas" : "Little Rock"
    }
state_capitals["Iowa"] = "Des Moines"

state = input("What state's capital are you looking for today?
")
capital = state_capitals[state]


print("The capital of " + state + " is " + capital + ".")
```

As you can see, we can adapt and adjust the code to better suit our needs. Now let's take a look at a few algorithms to determine whether they would run; that is, whether they would produce an error or run appropriately.

Algorithm analysis 2 – Terminating or not terminating?

As we discussed earlier in this chapter, algorithms should be terminating. That is, they must have a way to end, or they can cause many errors. Let's look at an algorithm and analyze it to determine whether it will terminate or not:

```
x = 0
while x >= 3:
x += 1
print(x)
```

First, let's take a look at the value of the x variable. The x variable starts the program with a value of 0. The while loop, which states the conditions under which the value of x will change, states that when the x value is greater than 3, it is incremented by a value of 1.

This algorithm terminates because it will print the original value for the variable, 0. However, this algorithm doesn't really perform any actions, as the condition will never be met. Also, notice that the print command is not indented. If it were indented, no output would be given for this algorithm, as the print command would never be called since the variable will never meet the conditions of the while loop.

Now let's take a look at the following algorithm:

```
j = 0
while j >= 0:
     j -= 1
     print(j)
```

In this case, the variable condition is met because j has to be greater than or equal to 0 for the program to run. Once the condition is met, the value of the variable is decremented by 1, so the print command will produce an output of -1. The code will not run a second time because the value of the variable is no longer greater than or equal to 0. This algorithm is terminating, produces an output, and is feasible.

Finally, let's take a look at the following algorithm with a changed condition:

```
j = 0
while j <= 0:
     j -= 1
     print(j)
```

In this case, the algorithm isn't terminating. Because we changed the while loop to be less than or equal to 0, this algorithm will now continue to run forever.

Analyzing algorithms can be very complex. We have only started to touch on some of the components of algorithms. As we delve deeper into other computational thinking problems throughout this book, we will need to keep in mind the characteristics of a good algorithm in order to analyze our own code effectively. It is also important that we continue to take into consideration the elements of the computational thinking process: **decomposition**, **pattern recognition**, **pattern generalization**, and **algorithm design**.

When we are designing the algorithm and testing it, using the characteristics of good algorithms will allow us to observe errors, adjust our algorithm for ease of use, provide better inputs and outputs, and ensure that we are not creating infeasible and non-terminating algorithms.

Summary

In this chapter, we discussed the definition of an algorithm, which is a set of steps that allows a computer to complete a process and provide some output. We went through the characteristics of algorithms.

We designed algorithms based on problem scenarios and then analyzed algorithms to determine whether they met the characteristics needed to run properly. Understanding the characteristics of algorithms and how algorithms work will allow us to create algorithms with far fewer errors than if we were unaware of these characteristics. Notice that I said *fewer* errors.

When working with code, errors are a fact of life. We will inevitably make mistakes and we will accidentally introduce bugs or make some code infinite. Understanding the characteristics of a good algorithm allows us to reduce those errors, even if we don't fully eliminate them from our daily lives.

In the next chapter, we will be learning more about logical reasoning. Throughout the chapter, we will discuss the definition of logic, learn about inductive and deductive reasoning, add to our knowledge of operators and Boolean logic, and learn more about logic errors. We will be using elements of computational thinking and the characteristics of algorithms to further our knowledge of logical reasoning.

4
Understanding Logical Reasoning

In this chapter, we will explore logical reasoning processes such as conditional statements, algorithmic reasoning, and Boolean logic. We will explore inductive and deductive reasoning before delving deeper into some logic operators. We will also learn about logic errors, how to identify them, and how to correct them.

In addition, we will be looking at writing algorithms using logic to solve computational thinking problems.

In this chapter, we will cover the following topics:

- Understanding the importance of logical reasoning
- Using Boolean logic and operators
- Identifying logic errors

When solving computational thinking problems, logical reasoning is necessary. We all know that programming code has steps that are followed linearly. Imagine we have 10 lines of code. If we do not apply logical reasoning, the code reads one line at a time—reads the first line, reads the second line, and so on until the last one. Using logical reasoning allows us to compare things before moving on, return to a previous line of code, and much more. Throughout this chapter, we will learn about logical reasoning in order to create algorithms that address problems in efficient ways using logic operators.

In order to understand logical reasoning, we will begin by defining logic in general, then how to use logic when designing and writing algorithms.

Technical requirements

You will need the latest version of Python to run the codes in this chapter. You will find the full source code used in this chapter here: `https://github.com/PacktPublishing/Applied-Computational-Thinking-with-Python/tree/master/Chapter04`

Understanding the importance of logical reasoning

As we worked through *Chapter 3*, *Understanding Algorithms and Algorithmic Thinking*, we learned how to design some basic algorithms while solving computational thinking problems.

In this section, we will understand how important logical reasoning is and how to apply the types of logical reasoning with the help of examples.

As we write algorithms, we'll need to use logical reasoning to create these algorithms.

In simple terms, **logical reasoning** is the set of steps followed to reach a conclusion. In computational thinking, when we design algorithms, the systematic set of steps we follow are part of the algorithm. The way a computer reads those algorithms depends on how we write that algorithm. There are two types of logical reasoning arguments, which are as follows:

- **Inductive reasoning**
- **Deductive reasoning**

Before we define those in more depth, let's look at why logical reasoning is so important and why order matters when we create algorithms.

In order to analyze a problem and provide an algorithm that helps us tackle the problem, we need to understand what logical reasoning is first. Logic can be daunting for some, but we use it every day subconsciously.

Let's look at a simple example. Say you take a shower every morning and go to work. *Well, would you get dressed for work before the shower?* No, because that would make absolutely no sense. Logically, you'd have to shower first before you put on your clothes for work. Now, I've skipped a ton of steps here, but those steps are logical steps. Other examples of logic include following recipes, using an umbrella if it's raining (or not), and so on.

Throughout this chapter, we'll weave in and out of logical reasoning and designing algorithms using **logical operators**. A logical operator allows a program to make decisions. We use those too in everyday life without realizing it. For example, if it's sunny and warm, we may want to go biking, but not if it's sunny and cold. The *and* here is a logical operator.

We take a lot of things into consideration when we're making decisions. In computational thinking, especially in algorithm design, we need to consider those things and provide a way for the program to test those conditions. We will delve deeper into logical operators later in this chapter. For now, let's look more closely at the types of logical reasoning and how to apply them.

Applying inductive reasoning

When we talk about inductive reasoning, we're really working backward. Inductive reasoning starts from a conclusion, which may be true or not, and works backward to create the code using the existing data. Let's look at a simple problem first.

Solving an inductive reasoning sample problem

We have a budget of \$150 for buying supplies: art pencils and erasers. The art pencils are \$1.75 each and the erasers are \$1.50 each.

Remember, in computational thinking, we decompose the problem first, then we identify the pattern, then we generalize that pattern, and then we create the algorithm. So, let's recognize that pattern.

Let's look at what we know so far and name some variables:

- The total budget is \$150.
- The cost of pencils is \$1.75 each.
- The cost of erasers is \$1.50 each.
- Let's denote the number of pencils by p.
- Let's denote the number of erasers by n.

Remember that when we get to that algorithm, we may want to rename those variables. But for now, since we're going to look at mathematical algorithms first, we'll keep the simple variables.

We can do this in one inequality. *Why an inequality and not an equation?* Because our total may not be exactly $150. But it can't be *more* than $150 because that's all the money we have.

Because this is a simple problem, we're identifying and generalizing that pattern in one move.

So, the number of pencils times the cost plus the number of erasers times the cost is less than or equal to $150:

$$1.75p + 1.50n \leq 150$$

Now let's talk about the algorithm design. Maybe this is something I buy regularly because I run art classes. I'm going to go off that scenario. Maybe my employer gives me at most $150, but depending on what I used before, I may need more pencils than erasers and vice versa. So, I need a program that I can use and reuse at the beginning of every term. *Was this part of my problem?* No, this was an ill-defined problem. So, I'm adapting the problem based on a set of particular needs. In short, I'm defining the problem I want to solve.

Important Note:
As a side note for an inductive and deductive reasoning dilemma, it is important to understand that conditional statements, such as the `if/then` statements we use often in programming, are usually associated with deductive reasoning. We can go into a debate about whether or not they can be inductive, but the truth is, inductive reasoning problems will use deductive reasoning. We will look in depth at deductive reasoning statements in the next section of this chapter.

So, I want the program to ask me how many pencils I want or I want it to ask me how many erasers I want. It all depends! Let's look at what the program should do for us. The following steps show us this:

1. Ask whether your input will be pencils or erasers.
2. Choose an inequality to use based on the input provided.
3. Identify how many of the pencils or erasers are possible (given the input).
4. Give a total cost for the number of pencils and erasers.

Please note that, as always, there are a lot of ways to arrive at the same answers in Python. While some of these programs are longer than what I would normally present, since we're learning both computational thinking and Python programming, it's important to show steps that are easy to understand.

For this particular program, we're going to need to import the math functions so that we can round down. *Why do we need to do that?* Well, we can't buy parts of erasers and pencils, only whole pencils and whole erasers. So, if the program says we can buy 19.5 pencils, that really means we can only purchase 19 pencils.

The `math.floor()` function allows us to round that number down to 19 with the simple function. We will explore more of the `math` functions as we go along in this book. Before we leave this quick topic, you should know that the `math` module in Python has built-in functions that align with the C language functions.

Let's go back to the problem. Take a look at the following written program:

ch4_inductiveP1.py

```
#We need the math module, so don't forget to import it.
import math
#Ask the user if they will input pencils or erasers first.
item1 = input("Will you be entering pencils or erasers? ")

if item1 == "pencils":
    pencils = int(input("How many pencils will you purchase? "))
    if pencils * 1.75 < 150:
        pencilstotal = pencils * 1.75
        total = 150 - pencilstotal
        total = total / 1.50
        erasers = math.floor(total)
        total2 = pencilstotal + erasers * 1.50
        print("You will be able to purchase " + str(pencils) + " pencils and " + str(erasers) + " erasers for a total cost of $" + str(total2) + ".")
    else:
        print("That's too many pencils.")
elif item1 == "erasers":
    erasers = int(input("How many erasers will you purchase?
"))
    if erasers * 1.50 < 150:
        eraserstotal = erasers * 1.50
        total = 150 - eraserstotal
        total = total / 1.75
        pencils = math.floor(total)
        total2 = pencils * 1.75 + eraserstotal
        print("You will be able to purchase " + str(pencils) +
" pencils and " + str(erasers) + " erasers for a total cost of
$" + str(total2) + ".")
#If the input given is too large based on the budget, this line
of code alerts the user.
    else:
        print("That's too many erasers.")
#If the input is incorrect, the program will print a statement
to alert the person that they need to use pencils and erasers
as input first.
else:
    print("Please run the program again and enter erasers or
pencils as your input.")
```

Remember that the preceding program will run the lines of code in order (sequentially). So, if a user inputs `erasers` first, then the first `if` statement and the nested `if` statement are ignored. If the user enters `pencils` first, then the algorithm runs normally from the first `if` statement and goes through the remaining conditions. Here's what the program does, in order:

1. Asks the user to input whether they are buying pencils or erasers.
2. If the user enters pencils, then the program asks how many pencils they'll purchase. Then, it calculates the number of erasers they can afford to buy.
3. If the user enters a number of pencils that is too large, they'll get a message that they can't afford that amount.
4. If the user enters erasers, then the program asks how many erasers they'll purchase, then calculates the number of pencils the user can afford to buy.
5. If the user enters a number of erasers that is too large, they'll get a message that they can't afford that amount.
6. If the user enters neither pencils nor erasers, they'll get a message to run the program again and enter one of those two options.

The preceding is an oversimplified inductive reasoning problem. Some inductive reasoning problems will ask that you look at data, make some probable conclusions, and then write a program to test those conclusions. In the process of learning logical reasoning, we are essentially training ourselves to look at decisions and how to process them in a way that a program can return the output we are looking for.

It is important to note here that there are multiple ways to look at problems and prepare solutions. While I prefer decision trees and flow-charts, other programmers and developers work more mathematically. Yet others like to write down what the program needs to do in simple sentences and/or paragraphs. The point of this process is to allow us to create a program that produces the necessary output and is easy to follow logically by both the programmers and developers and the computer running it.

Now, let's take a look at deductive reasoning.

Applying deductive reasoning

We're now at the section of this chapter that focuses on deductive reasoning. Even when I was a mathematics student, I found deductive reasoning fascinating. I quickly learned that mathematics taught us how to follow arguments logically in geometry and I fell in love with all things logic and truth tables.

Logic is taught using proofs and inductive and deductive reasoning. Truth tables help us analyze conditions. In truth tables, some things are assumed. For example, a statement is either true or false. The other statement is true or false. A combination of those statements depends on whether or not the statements are true or false.

Alright, that's a bit complicated. Before I move on to explain deductive reasoning, let's look at a quick truth table and the logic process it contains.

Truth tables were critical when I first started coding. They helped me understand the coding processes and how to work with conditions. Not every programmer or coder uses these tables, but I find them helpful, even if not used explicitly in the decision-making process. Let's look at one now.

Let's say we have a statement or condition `p` and that condition is `True`. Let's say that we have another statement or condition `q` and that it is also `True`. In truth tables, we use the symbol ¬ to denote *NOT*. So, `¬ p` ¬ is `False` and `¬ q` is also `False`. That's because if `p` is `True`, then *NOT* `p` is *NOT* `True`, in other words, `False`. The symbol ∧ is used for *AND*, so `p` *AND* `q` is written as `p ∧ q`. The symbol ∨ is used for *OR*, so `p` *OR* `q` is written as `p ∨ q`. In table format, our truth table looks as follows:

p	q	$\neg p$	$\neg q$	$p \wedge q$	$p \vee q$	$\neg p \wedge q$
True	True	False	False	True	True	False
True	False	False	True	False	True	False
False	True	True	False	False	True	True
False	False	True	True	False	False	True

Figure 4.1 – Truth table

Analyzing a truth table and understanding all the possible conditions can take time, but the process is similar to what we go through in logical reasoning when writing algorithms for problems. Now, let's take a closer look at deductive reasoning.

Let's first define what deductive reasoning is. **Deductive reasoning** is the process of going from a statement or hypothesis to a conclusion. Because deductive reasoning is what we use in algorithmic design, for the most part, we will need to define some terms associated with it.

Let's start with conditional statements.

Learning about conditional statements

Conditional statements are if/then statements. Here are a few logical arguments using conditional statements:

- If it rains, then I'll use an umbrella.
- If I drink water, then I won't be thirsty.
- If my dog needs to go out, then he stands by the door.
- If a quadrilateral has four right angles, then it is a rectangle.

All the preceding statements are examples of conditional statements. The first part of the statement is called the **hypothesis**. The second part of the statement is the **conclusion**. In the statement *If it rains, then I'll use an umbrella*, the hypothesis is *it rains* and the conclusion is *use an umbrella*. We do not include *if* or *then* in the hypotheses and conclusions.

In Python, as you saw in the example in the *Applying inductive reasoning* section, we use `if/then` statements when writing algorithms often. Here are some of the logical statements we use in Python:

- `if`: When using `if` statements, we ask whether a condition is met, then do something based on that true or false condition.
- `if-else`: When using `if-else` statements, we test one condition and do something, but if that condition is not met, then we do something else.
- `if-elif-else`: When using `if-elif-else` statements, we have one condition; if that's not met, we test another condition—that is, the **else if** (**elif**) condition—otherwise, we do something else.

All of the preceding statements can be nested. I can test one condition, then another, then another. I can have multiple `elif` statements between `if` and `else`, and so on. Let's look at some examples.

if statements

Let's look at a program that only uses one `if` statement:

ch4_ifStatement.py

```
number = int(input("What's your favorite number? "))
if number < 100:
    print("That's not a very large number.")
```

Now, the preceding code is a simple program, and one that only checks one condition. We could add conditions to test whether `number` is equal to `100`. We can add another one if `number` is larger than `100`, and so on. In this case, we only cared if `number` was below `100`.

If we input the number `53`, we'll get the following output:

```
What's your favorite number? 53
That's not a very large number.
```

If we input the number `100`, we won't get any message at all and the program will end:

```
What's your favorite number? 100
```

As you see, the program doesn't have anything to add. The condition wasn't met, so it ended. That's why `if-else` statements can come in handy.

if-else statements

Let's look at the previous algorithm and add an `else` statement. The previous program only checked whether the number provided was less than `100`. If we add an `else` statement, we can print something else on the screen for all numbers greater than or equal to `100`. Take a look at the following program:

ch4_if-elseStatement.py

```
number = int(input("What's your favorite number? "))
if number < 100:
    print("That's not a very large number.")
else:
    print("I guess you like large numbers.")
```

The preceding program now prints out a message regardless of what number the user gives as input. Let's test out `100` again:

```
What's your favorite number? 100
I guess you like large numbers.
```

As you can see, `100` is included in the large numbers category because our condition is that the numbers are less than `100`. That means `100` is not included in the condition. Testing conditions is how we arrive at conclusions in Python. We write algorithms that gather information from the program itself or user input, then it tests conditions to make decisions.

The following chart diagram shows the flow chart for `if-else` decision-making. We'll look at additional flowcharts when looking at `if-elif-else` statements and nested statements:

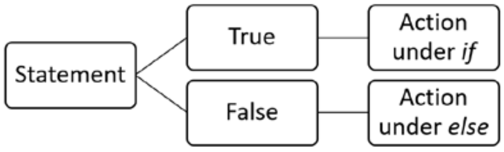


Figure 4.2 – if-else statement decision flowchart

As you can see in the preceding diagram, this is a binary decision. The statement will be tested to check whether it's **True** or **False**. If **True**, an action happens; otherwise, another action happens. For our number program, if the number was under `100`, one message is printed; otherwise, another message, `I guess you like large numbers`, is printed on the screen. Now, let's add multiple conditions.

if-elif-else statements

An `if-elif-else` statement is a simplification of a multi-condition statement—that is, you can have multiple `elif` statements. As mentioned earlier, **elif** stands for **else if**. Let's change our program up a bit. We'll allow the user to input a number between `1` and `20`. Here's what we'll program the algorithm to do:

1. Ask for a number between `1` and `20`.
2. Test whether the number is between `1` and `10` and print a message.
3. Test whether the number is between `11` and `20` and print a message.
4. Print an error message.

Let's take a look at how we'd program this. We need to remember a few things before we write this algorithm. To check numbers between `1` and `10` easily, we need to check that the number is less than `10`. That means `10` is not included.

Our `elif` statement would then check for numbers under `21`, since it will only include numbers we haven't tested yet. That is, if the user inputs `12`, the first condition isn't met, so it moves to the second condition. Yes, that would include all numbers under `21`, but keep in mind that if the number had been less than `10`, it will have already met a condition and the program would have printed the right message.

Finally, if the condition isn't met, we need to let users know they wrote a number that's not between `1` and `20`. The following program demonstrates this:

ch4_if-elif-elseStatement.py

```
number = int(input("Pick a number between 1 and 20. "))
if number < 10:
    print("That's less than 10.")
elif number < 21:
    print("That's between 10 and 20.")
else:
    print("That number isn't between 0 and 20. Run the program 
and try again.")
```

Let's try to test this with a number under `10`. If we run the program with the number `8`, we see the following output:

```
Pick a number between 1 and 20. 8
That's less than 10.
```

If we run the program with the number `10`, we see the following output:

```
Pick a number between 1 and 20. 10
That's between 10 and 20.
```

Finally, if we run the program with the number `21`, this is what we see:

```
Pick a number between 1 and 20. 21
That number isn't between 0 and 20. Run the program and try
again.
```

As you can see, each condition provides us with the answer for that given condition. Here's the flowchart for `if-elif-else` statements:

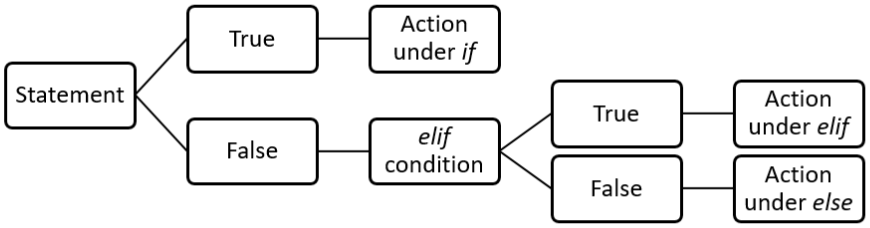


Figure 4.3 – if-elif-else statements decision flowchart

As you can see in the preceding diagram, `elif` just presents a new test. If **True**, we follow the action from the algorithm. If **False**, we move on to the `else` statement. That said, we can have multiple `elif` conditions. That means we can continue to test conditions one after the other unless and until we reach an `else` statement.

Understanding nested statements

Another type of logic statement we use in Python has to do with nested statements. In nested conditions, the `if` statement that is nested is only followed if the previous `if` statement is **True**. This is easier to understand with an example. Let's go back to our `if-elif-else` statement and add some nested conditions. We had previously asked the user to give a number between `1` and `20`. Now, let's say we want to subdivide the conditions further using the following code:

ch4_nestedStatements.py

```
number = int(input("Pick a number between 1 and 20. "))
if number < 10:
    if number < 6:
        print("Why such a small number?")
    else:
        print("Well, less than 10 but greater than 5. I'll take 
it.")
elif number < 21:
    if number < 16:
        print("You like values that are greater than 10, but 
not too much greater. I guess that's fine.")
    else:
        print("I like larger numbers myself too.")
else:
#Sometimes we make mistakes when providing input in programs. 
If you choose a number that's not between 0 and 20, the program 
will print this message.
    print("That number isn't between 0 and 20. Run the program 
and try again.")
```

In the preceding code snippet, the code has a message for any time we enter numbers that do not meet the guidelines. For example, the input requested is between `1` and `20`. *But what happens if the user types 0 or 21, or another number not in that range?* Then, the `print()` statement provides a message that asks the user to run the program again.

In this case, you can see that we have `if` statements, `elif` statements, nested `if` and `else` statements, and so on. Let's see a few test cases that test a few conditions to see what our program says:

- When we input `4`, the following output is seen:

```
Pick a number between 1 and 20. 4
Why such a small number?
```

- When we input `6`, we see the following:

```
Pick a number between 1 and 20. 6
Well, less than 10 but greater than 5. I'll take it.
```

- When we input `11`, we get this:

```
Pick a number between 1 and 20. 11
You like values that are greater than 10, but not too much
greater. I guess that's fine.
```

- When we input `18`, we get the following output:

```
Pick a number between 1 and 20. 18
I like larger numbers myself too.
```

As you can see from the preceding test cases, we have more outputs based on the conditions given in the program. While this was a simple number program, we can use similar logic when we are solving more complex problems.

Let's say you run an online store. The selections a user makes for items are going to be used in similar algorithms, albeit much more complex ones. The algorithm tests the conditions, such as items selected, quantities selected, and so on to apply totals, coupons, and much more. That's why logic and logical reasoning is so important in programming.

Now, as mentioned previously, the logical processing we use can be different for individual programmers. However, regardless of preference, logical reasoning and logical processing are absolutely necessary when we are writing algorithms. Rather than diving into the writing, we process problems, look at the decision-making and which steps need to happen, and then we design the algorithm. That logical process is critical to creating effective algorithms. We will continue to look at logical reasoning throughout this book as we analyze problems, even if we don't explicitly state so.

In this section, you learned about logical reasoning and its two types—inductive and deductive reasoning. We also learned about the conditional statements that will come in handy while coding.

Some of the algorithms we write can be simplified using Boolean logic and operators, which is what we'll take a look at in the next section.

Using Boolean logic and operators

Boolean logic refers to the operators, namely, `and`, `or`, and `not` in Python. You'll recall seeing this in the brief discussion on truth tables earlier in this chapter. As we'll see next, we use the same logical processing when writing the algorithms, even if the tables are not explicitly stated or used. When solving computational thinking problems, we sometimes have to meet multiple conditions at once. Let's look at this using just language for now.

Let's sort some fruit. If the fruit is round and orange, green, or yellow, it will be sorted into **group 1**. If the fruit is not round, but is orange, green, or yellow, it will be sorted into **group 2**. If the fruit doesn't match those requirements, it goes into **group 3**. Let's simplify these groups:

- **Group 1**: Round AND (orange OR green OR yellow)
- **Group 2**: Not round AND (orange OR green OR yellow)
- **Group 3**: All other fruit

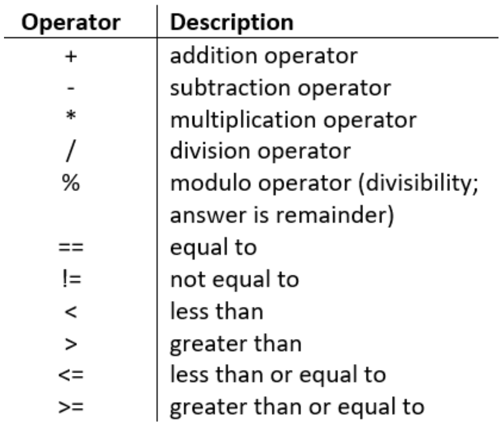
I know I stated the round condition first. But if you take a look at *groups 1* and *2*, the fruits need to be tested for those colors for both conditions—that is, if that condition is not met for color, it doesn't matter whether the fruit is round or not, it goes in *group 3*. So, here's what I'd write for an algorithm:

1. Test whether fruit is orange, green, or yellow.
2. If yes, test whether round, and sort into *group 1* or *2*.
3. If no, sort into *group 3*.

So, if we had a mandarin orange, that falls under *group 1*. If we had a banana, it would be in *group 2*. If we had strawberries, they would be in *group 3*.

Now, if we were going to write this, we'd need to make sure we've added the characteristics of the fruits so that we can test them against something. We will be looking at something like that in further chapters of this book, but for now, to simplify some of the learning, we'll create a similar algorithm but with numbers.

Before we move on too much, let's take a quick look at the basic operators in Python:



Operator	Description
+	addition operator
-	subtraction operator
*	multiplication operator
/	division operator
%	modulo operator (divisibility; answer is remainder)
==	equal to
!=	not equal to
<	less than
>	greater than
<=	less than or equal to
>=	greater than or equal to

Figure 4.4 – Basic Python operators

We'll go deeper into these operators when we get to *Section 2*, *Applying Python and Computational Thinking*, and take a deeper look at the Python programming language. However, we will need to use some of these for the next algorithm. First, let's look at the `and` operator.

The and operator

To understand the `and` operator better, it's best to look at a mathematical algorithm. Let's ask for a number and test whether that number is larger than `100` and a multiple of `2`. To test whether a number is a multiple of `2`, we use the **modulo operator** (**mod**). The symbol for `mod` is `%` in Python.

So, looking at the code, if `number % 2 == 0`, then the number is divisible by `2`. If `number % 2 == 1`, then it is not divisible by `2`. We use the equal (`==`) operator or not equal (`!=`) operator to complete these conditions:

ch4_andOperator.py

```
number = int(input("Give a number between 1 and 200. "))
if number > 99 and number % 2 == 0:
    print("That's a large, even number.")
elif number > 99 and number % 2 != 0:
    print("That's a large, odd number.")
elif number < 100 and number % 2 == 0:
    print("That's a small, even number.")
else:
    print("That's a small, odd number.")
```

Now, I know we've talked about different ways to write algorithms. *Did I need to use an AND operator for this one?* Probably not. I could have just written it as nested statements, `if-elif-else` statements, and so on. Some test cases and the results of the algorithm are shown as follows:

- When we input `104`, we see the following output:

```
Give a number between 1 and 200. 104
That's a large, even number.
```

- When we input `80`, we see the following output:

```
Give a number between 1 and 200. 80
That's a small, even number.
```

- When we input `31`, we get the following output:

```
Give a number between 1 and 200. 31
That's a small, odd number.
```

As you can see from the previous test cases, the program tests our cases and provides the printed messages based on the conditions met. Now, let's take a look at the `or` operator.

The or operator

As we saw in the fruit example earlier in this chapter, we checked whether the color of the fruit was orange, green, or yellow. That's how an `or` operator works. We check for something or the other. This time, we're going to look at some `True` and `False` statements. Let's say that variable `A` is `True` and variable `B` is `False`. If we were to use an `or` operator to check the result of `A` or `B`, then our answer would be `True`.

Why is that? Because no matter what, the result will be either `True` or `False`, which is a `True` statement. *Confused?* Logic can be confusing. Let's go ahead and test `A and B` as well as `A or B` in the following program to help you visualize this:

ch4_orOperator.py

```
A = True
B = False
C = A and B
D = A or B
if C == True:
    print("A and B is True.")
else:
    print("A and B is False.")
if D == True:
    print("A or B is True.")
else:
    print("A or B is False.")
```

Now, I added some conditions so that we'd get printouts and you could see that the logic I stated was right, but we didn't need to do all of this. We could have just printed `C` and `D`.

When we run this program, this is the result:

```
A and B is False.
A or B is True.
```

As you can see, `A and B` is `False` because one of the statements is `False`, which means the whole thing is `False`. `A or B` is `True` because one of them is `True`, so the condition is `True`. Now, let's look at the last operator (for now), the **not** operator.

The not operator

The `not` operator lets us test the opposite of things. So, if `A` is set as `True`, then `not A` is `False`. It's as simple as that. Let's look at a few examples through the following code:

ch4_notOperator.py

```
A = True
B = False
print(not A)
print(not B)
print(not (A and B))
print(not (A or B))
```

From the previous code, we've talked about the first printed statement here. Since `A` is `True`, `not A` is `False`. For the second `print` statement, we expect that result to be `True` because `B` is `False`. Now, we did the `A and B` and `A or B` statements previously. We know that `A and B` is `False`, so `not (A and B)` is `True`. We also know `A or B` is `True`, so `not (A or B)` is `False`.

Let's look at what the program prints:

- It prints the following for `not A`:

```
False
```

- Similarly, for `not B`, it prints the following:

```
True
```

- Also, for `not (A and B)`, it prints the following:

```
True
```

- Lastly, for `not (A or B)`, it prints the following:

```
False
```

In this section, you have learned about a few of the Boolean operators. With Boolean operators, we can write algorithms that test cases for us and provide outputs based on those cases. As mentioned, a program will run based on the instructions we write in the algorithm.

By writing our algorithms using these operators, we can ensure that conditions are applied only in the circumstances we want them to apply. Rather than having a program run on incorrect conditions, we can include statements and prompts to help produce the right outcomes. For example, if an input for distance is accidentally entered as negative, a Boolean statement could have checked conditions and provided the person with feedback within the program, then run again. Using Boolean operators provides clear logical processes and allows better and clearer algorithms.

Now that we've taken a look at the basic operators, it's important that we also look at errors. Identifying logic errors can help us avoid pitfalls in our algorithms.

Identifying logic errors

Before we talk too much about logic errors, let's talk about why it's important to keep them in mind. In Python, not all errors lead to a failed or crashed program. Some logic errors will allow a program to run entirely without crashing at all or alerting the user of an error. Those errors are hard to identify.

Here are some logic errors that can get us in trouble, but keep in mind that there are many ways to incorporate logic errors into our programs accidentally:

- Using the wrong variable in an equation or statement
- Using the wrong operator to test conditions
- Using the wrong indentation when checking for conditions

The one I am the guiltiest of is switching my variables, but I do also make mistakes in indentation often. Usually, those get identified more often when I try to run the program, because the program may fail to run in some instances.

Let's take a look at a simple algorithm that contains an error in a formula. In this first algorithm, the goal is to get the total cost after buying a number of orders of fries from a restaurant at a cost of $1.50 each:

ch4_Error1.py

```
number = int(input("Type the number of fries you are ordering: "))
cost = 1.50
total = number * number
print("Your total cost is $" + str(total) + ".")
```

If we run the preceding program, the program will run without problems/errors and show the following output for `12` orders of fries:

```
Type the number of fries you are ordering: 12
Your total cost is $144.
```

Now, if we're paying attention, we'll realize that the cost for 12 orders of fries is too high at $144. That's because there is an error in our algorithm. The algorithm should contain the `total = cost * number` formula, as follows:

ch4_Error1_fixed.py

```
number = int(input("Type the number of fries you are ordering: "))
cost = 1.50
total = cost * number
print("Your total cost is $" + str(total) + ".")
```

Now that we've fixed that formula, the output is correct:

```
Type the number of fries you are ordering: 12
Your total cost is $18.0.
```

As you can see, `$18.0` seems a lot more reasonable for 12 orders of fries at $1.50 each.

Errors in formulas can be difficult to find, especially if the program runs without alerting to the error. If we have a large algorithm with multiple formulas, finding those errors can become cumbersome and a lengthy process. The best recommendation for this is to test your algorithm at every step of the process you can. That way, finding errors becomes a simpler task.

Let's now take a look at an error when testing conditions. Much like errors in formula, errors in condition testing may be hard to spot, as the program may just run anyway:

ch4_Error2.py

```
number = int(input("Give a number between 1 and 200. "))
if number > 99 and number % 2 == 0:
    print("That's a large, even number.")
elif number > 99 and number % 2 != 0:
    print("That's a large, odd number.")
elif number < 100 or number % 2 == 0:
    print("That's a small, even number.")
else:
    print("That's a small, odd number.")
```

In the preceding code, there is an error in the algorithm that causes us to get incorrect feedback when entering some odd numbers. Take a look at the second `elif` statement. That `or` will produce an error.

If we run this program, we get an output. Let's run it with the number `99`:

```
Give a number between 1 and 200. 99
That's a small, even number.
```

Now, the problem here is that `99` is not an even number. Somewhere in the algorithm, we introduced an error in the conditions. In this case, instead of using an `and` operator, we used `or`:

```
elif number < 100 or number % 2 == 0:
    print("That's a small, even number.")
```

Once we replace the `or` with `and`, we can run the program again:

ch4_Error2_fixed.py

```
number = int(input("Give a number between 1 and 200. "))
if number > 99 and number % 2 == 0:
    print("That's a large, even number.")
elif number > 99 and number % 2 != 0:
    print("That's a large, odd number.")
elif number < 100 and number % 2 == 0:
    print("That's a small, even number.")
else:
    print("That's a small, odd number.")
```

Using `99` as the input, we get the following output:

```
Give a number between 1 and 200. 99
That's a small, odd number.
```

Running the program with `98` as the input, we get the following:

```
Give a number between 1 and 200. 98
That's a small, even number.
```

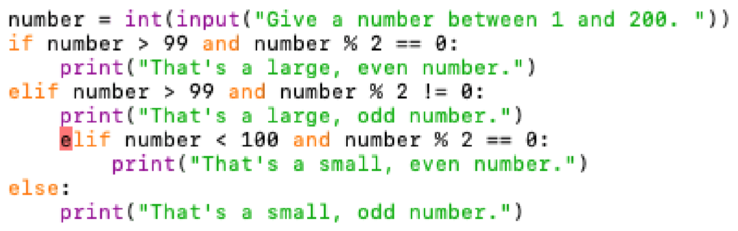
As you can see, unless we're paying attention, we can miss errors in our conditions and logical operators. Because the program is able to run with these errors in our algorithm, catching where exactly we made the mistake is harder to do than when we incorporate errors that stop the program from running.

Finally, let's take a look at an indentation error using the same code for the condition testing. This time, with an indentation error added, we have the following:

ch4_Error3.py

```
number = int(input("Give a number between 1 and 200. "))
if number > 99 and number % 2 == 0:
    print("That's a large, even number.")
elif number > 99 and number % 2 != 0:
    print("That's a large, odd number.")
    elif number < 100 and number % 2 == 0:
        print("That's a small, even number.")
else:
    print("That's a small, odd number.")
```

In this case, we can't run the program. The second `elif` statement is indented incorrectly. When we try to run the program, we get an `Invalid Syntax` error message. Clicking **OK** on the message will take us to the code and the indentation error is highlighted, as shown in the following screenshot:



```
number = int(input("Give a number between 1 and 200. "))
if number > 99 and number % 2 == 0:
    print("That's a large, even number.")
elif number > 99 and number % 2 != 0:
    print("That's a large, odd number.")
    elif number < 100 and number % 2 == 0:
        print("That's a small, even number.")
else:
    print("That's a small, odd number.")
```

Figure 4.5 – Indentation error

Notice that the `print()` code below the `elif` statement is also indented incorrectly. Once we fix those two errors, we can run the code, as we did previously in this chapter.

Incorporating errors into our algorithms is a common mistake. As you can see from the previous examples, identifying some of the errors can be hard to do, since the program may be running as if there is no problem.

I may not have caught many errors with conditions in my algorithms, but that may just be because I never realized there was a mistake to begin with. That's one of the reasons why it's really important to run various instances of our programs to ensure that the results we get make sense. We will be discussing more errors as we look at programs and computational thinking problems throughout this book. In the meantime, test your programs and test them often. Triple check your math, your indentations, and your logic.

Summary

In this chapter, we discussed inductive and deductive reasoning, logical reasoning, logical operators, and Boolean logic. As we discussed, most of the algorithm designs fall under deductive reasoning. We learned how to use statements, such as `if`, `if-else`, `if-elif-else`, and nested statements, to write programs that test conditions.

In addition, we learned that some errors are difficult to identify, so it is important to verify our programs and test them often.

After going through this chapter, you are now better equipped to write algorithms using logical reasoning. You also have the understanding to apply inductive and deductive reasoning when designing and planning algorithms and use Boolean logic and operators in your algorithms. You are now also able to test your algorithms for errors by identifying possible mistakes such as indentation errors, conditional errors, and formula errors.

In the next chapter, we will be taking a deeper dive deeper into problem analysis, using the computational thinking elements to break down problems so that we can create meaningful and useful algorithms.

5
Exploring Problem Analysis

In this chapter, we will explore problem analysis in depth while using some of what we have been learning, such as logical reasoning, Boolean logic, and algorithmic design. We will work through problem definition, decomposition, and analysis in this chapter.

In this chapter, we will cover the following topics:

- Understanding the problem definitions
- Learning to decompose problems
- Analyzing problems

In order to further understand problems, we'll need to look at a more complex problem and define it so we can begin the algorithmic design process. In this chapter, you will learn how to define problems and decompose them in order to design algorithms. In doing so, you'll also learn about dictionaries in Python. After reading this chapter, you'll be able to use the computational thinking process in order to design and create an algorithm that addresses complex problems.

Technical requirements

You will need the latest version of Python to run the codes in this chapter. You will find the full source code used in this chapter here: `https://github.com/PacktPublishing/Applied-Computational-Thinking-with-Python/tree/master/Chapter05`

Understanding the problem definitions

As we discussed in *Chapter 2*, *Elements of Computational Thinking*, computational thinking uses four elements in order to solve problems:

- **Problem decomposition**: This is the process of breaking down data.
- **Pattern recognition**: This is the process of finding similarities or patterns.
- **Abstraction**: This element deals with generalizing the pattern.
- **Algorithm design**: This is where we define the set of instructions for the solution to the problem.

In this section, in order to learn more about how to analyze problems, we're going to analyze a larger problem and work through the steps needed to create the algorithm. To be able to create algorithms, it is imperative that we analyze the problems and clearly identify what we are trying to solve. That is, *what is our algorithm for? Why do we need to build it?* Looking at the decomposition of problems and then defining what we need will provide us with a better algorithm at the end.

We will work through a problem in the next section.

Problem 5A – Building an online store

Let's take a look at the following problem. You are starting an online store. It's in its infancy, but you'll have three different types of item available. They are keychains, water bottles, and t-shirts. For this particular problem, we will go through a three-step process:

1. **Making assumptions**
2. **Things to consider**
3. **Building a dictionary**

We will look the preceding steps in the upcoming sections.

Making assumptions

Let me state some assumptions about this store that we are going to use:

- This is a company that provides items for clients to share with their customers.
- Each item can have a logo and/or personalized information, such as name, email, and phone number.

We will now move on to the next section, which is about things to consider.

Things to consider

Now let's take a look at some of the things that you'll need to think about before we even start working on an algorithm:

- *Are the items personalized?*
- *Will personalization be charged by character, line, or item?*
- *Will the price be fixed or will it change when customers make bulk purchases?*
- *Will there be discounts if a client orders more than one type of item?*
- *How much is the base price-point for each item?*

The preceding points are not the only questions that we could go over. But they're the questions that we'll start taking a look at when we decompose the problem.

Before we do that, let's talk about how we can include the information in a program for each of the items. If you recall from *Chapter 3*, *Understanding Algorithms and Algorithmic Thinking*, we can use a dictionary in **Python** to save our menu of items. In this case, we have keychains, water bottles, and t-shirts.

Building a dictionary

Before we take a look at the complexities presented by this problem and decomposing that information, we can build our own dictionary. We can make it so that the price for each item in the dictionary is the base price (the price that does not contain any customizations or discounts), as follows:

- Cost per keychain: $0.75
- Cost per t-shirt: $8.50
- Cost per water bottle: $10.00

Now let's build the dictionary. Remember that you can do this without the dictionary, but creating a dictionary allows you to update the pricing, if necessary, at a later date. You can also create functions to solve this problem. We are using logic and the dictionary for this problem. The following code shows you how to build a dictionary:

ch5_storeDictionary.py

```
online_store = {
    'keychain': 0.75,
    'tshirt': 8.50,
    'bottle': 10.00
    }
print(online_store)
```

From the preceding code snippet, keep in mind that the `print()` function is not needed here, but I use it often in order to ensure that the code is working properly while I continue to build the algorithms. Also notice that the names of the variables—`keychain`, `tshirt`, and `bottle`—are simplified.

Here's what that output will look like:

```
{'keychain': 0.75, 'tshirt': 8.5, 'bottle': 10.0}
```

What the output shows me is that the prices are saved correctly for each of the variables. I'm using that `print` function to test my dictionary and ensure that it runs correctly before I start working on what I need from that dictionary.

This helps us when we are writing the code and reuse the variables in multiple areas of that code. Having these simple and easy-to-identify variables will allow us to change and add to the algorithm without adding errors.

In this section, we learned that the problem analysis and definitions help us identify how to best design our solution. Remember, when we are looking at problems, the definitions we use, both before we write the algorithm and within the algorithm, are critical for our design and final product. Let's now look at the decomposition of the problem.

Learning to decompose problems

When we decompose problems, we're identifying what we need the algorithm to provide us with. The end user will need to see something seamless. Look at the flowchart in *Figure 5.1*; this is a basic decision-making flowchart to help us design our algorithm.

Let's make another assumption first, that is, if the user enters more than 10, the price will be lower. We're only going to do less than 10 or more than or equal to 10 in this case. However, if you needed to subdivide this further, you can add more cases, such as the following:

- Less than or equal to 10
- More than 10 and less than or equal to 50
- More than or equal to 50

You can have as many cases as you need. For the purposes of this algorithm, we're going to keep it to two cases, since we also have to include personalization costs and we don't want to create an overly complicated algorithm.

The following diagram shows you the flowchart for the algorithm:

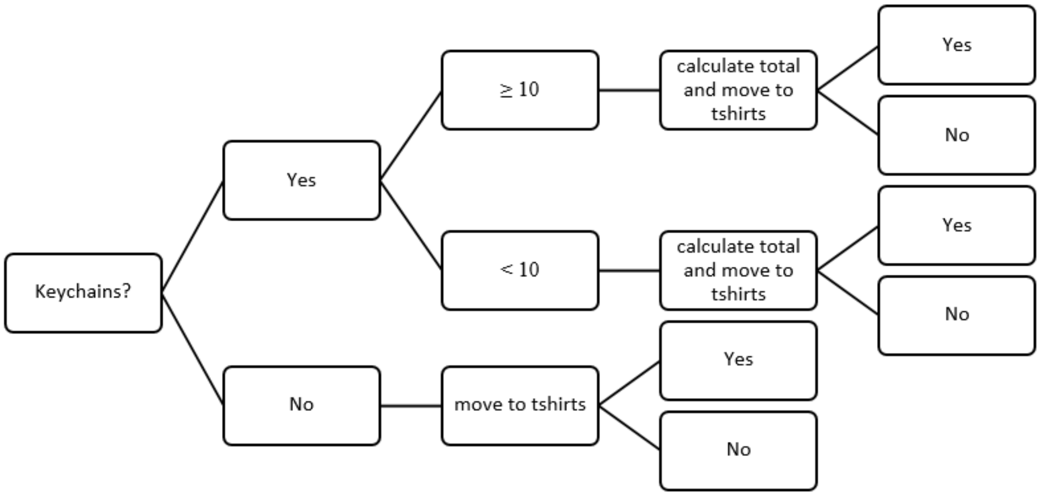

Keychains?
Yes
No
≥ 10
< 10
move to tshirts
calculate total and move to tshirts
calculate total and move to tshirts
Yes
No
Yes
No
Yes
No

Figure 5.1 – Initial decision-making flowchart

As you can see in the preceding diagram, this isn't a completed flowchart. After we make the decisions about the t-shirts, we need to move on to the bottles. How we write the algorithm will depend on what we'd like to output. Right now, we're providing the user the information they'd get when they check out of the online store that you created.

In the next section, let's use the preceding flowchart to create an algorithm.

Converting the flowchart into an algorithm

The diagram in *Figure 5.1* allows us to look at the decision-making process for the algorithm we're writing. We'll want to look at the following key points when writing the algorithm:

- **Dictionary and input**: Input can be within the algorithm or user-entered; dictionaries are built within the algorithm. This means, to use a dictionary, we have to define it in our algorithm before we are able to use it.
- **Cost**: This is the base cost for each item.
- **Personalization costs**: This is added to the base cost.

We'll now look at the preceding points in detail in the upcoming sections.

Building a dictionary and giving inputs

Before we add any of the complications, let's look at how to grab the price of each item and use it at the base price. We'll need a count for the number of each item. The following code shows you this:

ch5_storeQuantities.py

```
online_store = {
    'keychain': 0.75,
    'tshirt': 8.50,
    'bottle': 10.00
    }
keychain = online_store['keychain']
tshirt = online_store['tshirt']
bottle = online_store['bottle']
choicekey = int(input('How many keychains will you be
purchasing? If not purchasing keychains, enter 0. '))
choicetshirt = int(input('How many t-shirts will you be
purchasing? If not purchasing t-shirts, enter 0. '))
choicebottle = int(input('How many t-shirts will you be
purchasing? If not purchasing water bottles, enter 0. '))
print('You are purchasing ' + str(choicekey) + ' keychains, '
+ str(choicetshirt) + ' t-shirts, and ' + str(choicebottle) + '
water bottles.')
```

From the preceding code snippet, notice that we added the variables under the dictionary. This will be useful later. These variables are named `choicekey`, `choicetshirt`, and `choicebottle`. Naming the variables allows us to return to them and change code, as needed. In this case, each variable asks for input from the person running the program in order to get the number of keychains, t-shirts, and bottles they are ordering. Again, there are multiple ways to tackle this problem, but we're using what we've learned so far to create an algorithmic solution.

When we run the previous code for `3` keychains, `0` t-shirts, and `10` water bottles, this is our output:

```
How many keychains will you be purchasing? If not purchasing
keychains, enter 0. 3
How many t-shirts will you be purchasing? If not purchasing
t-shirts, enter 0. 0
How many t-shirts will you be purchasing? If not purchasing
water bottles, enter 0. 10
You are purchasing 3 keychains, 0 t-shirts, and 10 water
bottles.
```

As you can see, we have a program that takes user input, then confirms to the user the choices they have made for each of the items.

Let's see the next section about cost.

Making changes to the cost

Now let's add the changes in cost. Let's say that if a customer is purchasing over 10 items, then the updated costs are as follows:

- Keychains: $ 0.65
- T-shirts: $ 8.00
- Water bottles: $ 8.75

To make the preceding changes, we can make the program update the difference in cost, which is shown in the following code:

ch5_storeCost.py

```
online_store = {
    'keychain': 0.75,
    'tshirt': 8.50,
    'bottle': 10.00
    }
choicekey = int(input('How many keychains will you be
purchasing? If not purchasing keychains, enter 0. '))
choicetshirt = int(input('How many t-shirts will you be
purchasing? If not purchasing t-shirts, enter 0. '))
choicebottle = int(input('How many t-shirts will you be
purchasing? If not purchasing water bottles, enter 0. '))
print('You are purchasing ' + str(choicekey) + ' keychains, '
+ str(choicetshirt) + ' t-shirts, and ' + str(choicebottle) + '
water bottles.')
if choicekey > 9:
    online_store['keychain'] = 0.65
if choicetshirt > 9:
    online_store['tshirt'] = 8.00
if choicebottle > 9:
    online_store['bottle'] = 8.75
keychain = online_store['keychain']
tshirt = online_store['tshirt']
bottle = online_store['bottle']
print(online_store)
```

Now that we have updated the code, I'd like to print out my progress to make sure that the code is working properly and changes take place. In this case, I wanted to make sure that the costs would update if I had totals greater than 10. (That is, when a customer orders more than 10 of an item, it updates the cost for each item to the lower cost.) The output of the preceding code is as follows:

```
How many keychains will you be purchasing? If not purchasing
keychains, enter 0. 10
How many t-shirts will you be purchasing? If not purchasing
t-shirts, enter 0. 14
How many t-shirts will you be purchasing? If not purchasing
water bottles, enter 0. 10
You are purchasing 10 keychains, 14 t-shirts, and 10 water
bottles.
{'keychain': 0.65, 'tshirt': 8.0, 'bottle': 8.75}
```

You can now see from the preceding output that the dictionary has updated the values based on the totals the user provided.

Now we need to go ahead and provide the cost. We can provide the total item cost or the total cost of the full purchase, or both (let's do both). Take a look at the following code snippet:

ch5_storeTotals.py

```
keychain = online_store['keychain']
tshirt = online_store['tshirt']
bottle = online_store['bottle']
print('You are purchasing ' + str(choicekey) + ' keychains, '
+ str(choicetshirt) + ' t-shirts, and ' + str(choicebottle) + '
water bottles.')
```

The preceding code snippet is added so that we can have a `print` statement to confirm the input from the user. By printing that statement at the end of the code, we are checking with the user whether the program can read the numbers correctly, and whether the user entered the right numbers.

We can continue the code to add the costs per item:

ch5_storeTotals.py

```
totalkey = choicekey * keychain
totaltshirt = choicetshirt * tshirt
totalbottle = choicebottle * bottle
grandtotal = totalkey + totaltshirt + totalbottle
print('Keychain total: $' + str(totalkey))
print('T-shirt total: $' + str(totaltshirt))
print('Water bottle total: $' + str(totalbottle))
print('Your order total: $' + str(grandtotal))
```

The `print` statements at the end of the preceding code snippet provide the breakdown of each item total as well as the full order total. After asking for the input for all items, the code then prints the sub-totals for the cost of each of the items. The result of the preceding code is as follows:

```
How many keychains will you be purchasing? If not purchasing
keychains, enter 0. 10
How many t-shirts will you be purchasing? If not purchasing
t-shirts, enter 0. 7
How many t-shirts will you be purchasing? If not purchasing
```

```
water bottles, enter 0. 14
You are purchasing 10 keychains, 7 t-shirts, and 14 water
bottles.
Keychain total: $6.5
T-shirt total: $59.5
Water bottle total: $122.5
Your order total: $188.5
```

Now that we have the totals of the items without personalization, we need to be able to take into account the costs of that personalization, if ordered.

In the next section, let's take a look at what personalization costs are and the decisions we'll need to make before we move on.

Adding personalization

For now, let's limit the personalization of keychains, t-shirts, and water bottles to binary questions, that is, either the user wants personalization or not. We are not looking at tiered costs of personalization, which you may have seen. If you wanted to add tiers, you'd need to make more decisions, such as cost of choosing fonts, length of the personalization, and so on. We'll forgo those for now, but feel free to add to this code in order to address those kinds of customizations. Let's add another assumption for the personalization:

- $1.00 for the keychains
- $5.00 for the t-shirts
- $7.50 for the water bottles

We'll need to create the preceding conditions and then implement them into our variables. Let's look at the code in parts. The following file contains each of the parts we'll break down.

Recall that our algorithm first asked for input for the number of items they were purchasing. The following code snippet takes user input in order to take personalization into account:

ch5_storePersonalize.py

```
perskey = input('Will you personalize the keychains for an
additional $1.00 each? Type yes or no. ')
perstshirt = input('Will you personalize the t-shirts for an
additional $5.00 each? Type yes or no. ')
```

```
persbottle = input('Will you personalize the water bottles for
an additional $7.50 each? Type yes or no. ')
if perskey == ('yes' or 'Yes'):
    online_store['keychain'] = online_store['keychain'] + 1.00
if perstshirt == ('yes' or 'Yes'):
    online_store['tshirt'] = online_store['tshirt'] + 5.00
if persbottle == ('yes' or 'Yes'):
    online_store['bottle'] = online_store['bottle'] + 7.50
keychain = online_store['keychain']
tshirt = online_store['tshirt']
bottle = online_store['bottle']

totalkey = choicekey * keychain
totaltshirt = choicetshirt * tshirt
totalbottle = choicebottle * bottle
grandtotal = totalkey + totaltshirt + totalbottle
```

The preceding code snippet asks the user the binary questions on personalization. After grabbing the input, the code then makes some decisions based on the user input and defines the `keychain`, `tshirt`, and `bottle` variables and the totals for the choices. The following snippet then uses the totals to print out the information for each item purchased as well as the final total:

```
print('Keychain total: $' + str(totalkey))
print('T-shirt total: $' + str(totaltshirt))
print('Water bottle total: $' + str(totalbottle))
print('Your order total: $' + str(grandtotal))
```

From the preceding code, notice that the `keychain`, `tshirt`, and `bottle` variables are defined after all our customizations based on total numbers and personalization. Remember that in algorithm design, the order matters. If we locate those variables earlier in the program, the conditions, such as personalization, that follow will not affect those variables.

So, to be able to get everything we need for our variables, we need to define them after defining some of the conditions that affect them, such as customization. Take a look at the preceding code to note where the variables are. Feel free to play with the code by changing where you define the variables to see whether your end results change.

Here's a visual flowchart with the keychain decision-making process:

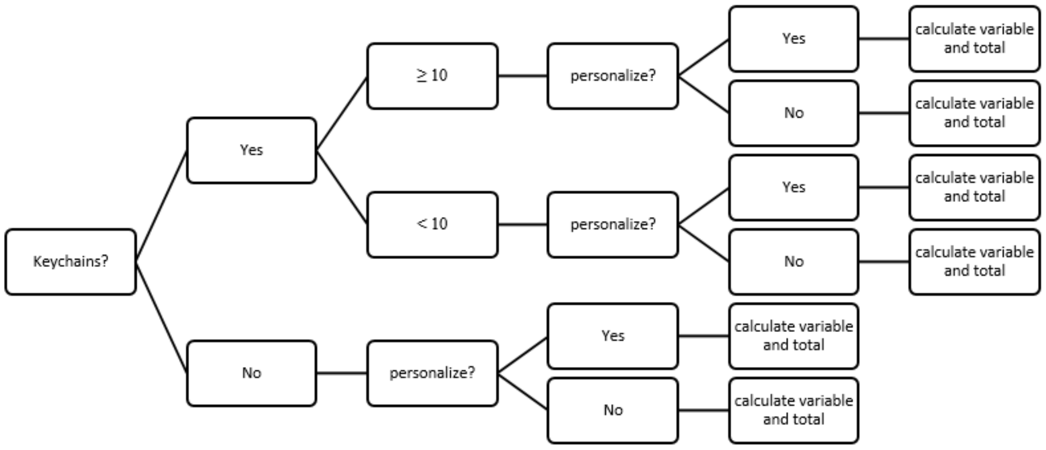

Keychains?
Yes
No
≥ 10
< 10
personalize?
personalize?
personalize?
Yes
No
Yes
No
Yes
No
calculate variable and total
calculate variable and total
calculate variable and total
calculate variable and total
calculate variable and total
calculate variable and total

Figure 5.2 – Keychain decision-making flowchart

As you can see from the preceding diagram, this is only for **Keychains**. We need to repeat the process for the other two variables. In the diagram, you can see the decision-making process for the item. First, the user indicates the number of items bought, then whether they will personalize them or not.

Depending on each answer, the total is calculated by the program. For example, if there is no personalization, the total is calculated sooner in the decision-making tree. We can rewrite this program using functions (as I mentioned before) to simplify some of the processes. For now, we are focusing on learning how to break down problems, analyze conditions, and how to design algorithms that take into account multiple decisions. Remember to complete the diagram for the other items so that the decision-making process is easier to code when designing the algorithm.

In this section, we learned how to use a flowchart to create an algorithm. We also learned about building a dictionary for our algorithm.

Before we move on, let's look at the process of analyzing problems. While we were creating this algorithm, we did that while we decomposed the problem. However, there are some key components of problem analysis that we should consider before our next chapter.

Analyzing problems

When analyzing problems, there are some steps that we can keep in mind to help us ensure that we are creating the best possible algorithm:

- Clearly read and understand the problem.
- Identify the main purpose of the solution.
- Identify the constraints of the problem.
- Identify the decision-making flow.
- Establish the possible algorithms that could solve the problem.
- Identify the best possible algorithm tools for the problem.
- Test the algorithm pieces frequently.
- Verify that the algorithm provides the solution for the identified problem.

If we go back to our problem, we went through this process throughout the chapter:

- We had an online store with three items.
- Item cost was dependent on quantity purchased.
- Item price was also dependent on personalization customizations.
- We created flowcharts to help us identify the decision process and how to code it.
- We verified our code through code lines that allowed us to check whether the algorithm was producing the correct response multiple times.
- We revisited and reordered pieces of code, as needed.
- We verified that the algorithm's output was in line with the problem we had identified.

The preceding process bears repeating, that is, this is not a linear process. Sometimes we'll write an algorithm and then revisit the decision flowchart, make adjustments, then tackle the algorithm again.

The need for analyzing our problems at multiple *stopping points* becomes even clearer when we are looking at larger problems. *Should we write hundreds of lines of code before testing?* No! Imagine having 300 lines of code, only to find an error on *line 20* that is carried forward throughout the rest of the algorithm.

Testing at every possible progress point will allow us to catch the small mistakes that can cost us in the long run. Remember, it's almost impossible to write a perfect algorithm on the first try. We all make mistakes, small and large, so it is important that we continue to test and analyze our progress.

Let's look at one more problem and go through the process again before leaving this chapter.

Problem 5B – Analyzing a simple game problem

You want to design a number guessing game. The user has to guess a random number.

Let's start by defining our problem, which in this case is a game. Let's identify the known information:

- The computer will need to randomly select a number.
- The user will need to input a number.
- The computer will have to check whether the input from the user matches the randomly generated number.

Now, that's not enough! If I don't match on the first try, do I lose? How many chances do I get? Is the random number a number between 1 and 10 or between 1 and 500? We're going to have to make some decisions before we start coding this. Let's add some parameters:

- The number is between 1 and 100.
- The user will get 5 chances to guess.
- The computer will tell the user if the answer is too high or too low.

Now that we have those parameters, we can create a decision flowchart:

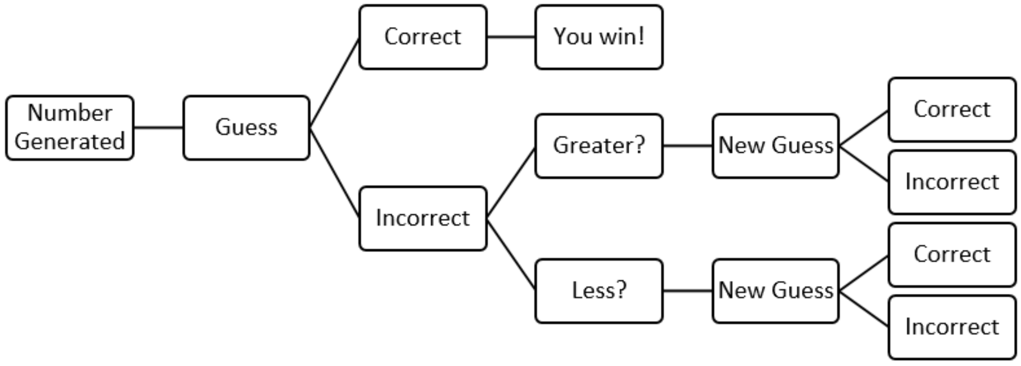


Figure 5.3 – Decision flowchart for guessing game

From the preceding diagram you can see that the chart is not complete. That's because we will use some logic to make the process repeat 5 times. We'll get into that in a moment. For now, notice the decisions. First, a number is generated by the program (but is not revealed). The user then inputs a guess, which is either correct or incorrect. If it's correct, then the user wins the game. If it's incorrect, then the program lets the user know if the answer was too low or too high and asks for a new guess. The process will then repeat itself, as needed. Now, let's write the algorithm.

First, let's generate the random number and get the user to guess it. Add a `print()` function for both the randomly generated number and the input from the user so that you can see that the information is working properly. Remember, we'll take those out later, but it's important to keep checking and rechecking our code as part of our problem analysis process. The following code will do the same:

ch5_guess1.py

```
import random as rand
compnumber = rand.randint(1, 100)
print(compnumber)

usernumber = int(input('Choose a number between 1 and 100.
You'll get 5 guesses or you lose! '))
print(usernumber)
```

You'll notice from the preceding code, the imported `random` module. We also imported it as `rand`. That's just to save time and space. When you import a module in Python, you can rename it. The `random` module gives us a way to generate the number in the range that we had selected.

The `rand.randint(1, 100)` code line includes `1` and `100`. These are the endpoints, or limits for the random number generator. The `rand` function refers to the module, as mentioned, while *randint(a, b)* refers to a random integer between *a* and *b* (including *a* and *b*).

Run the code a few times to see how the number generated by the computer changes each time. The following points show three test cases:

- The following is test case 1 of the preceding code:

  ```
  27
  Choose a number between 1 and 100. You'll get 5 guesses
  or you lose! 10
  10
  ```

  As you can see from the preceding output, `27` is the computer-generated random number and `10` is what the user entered.

- The following is test case 2 results of the previous code:

  ```
  68
  Choose a number between 1 and 100. You'll get 5 guesses
  or you lose! 65
  65
  ```

  As you can see from the preceding output of the code, `68` is the value of the `compnumber` variable, while the user (me) entered the number `65`. So close, yet so far!

- The following is test case 3 output:

  ```
  50
  Choose a number between 1 and 100. You'll get 5 guesses
  or you lose! 23
  23
  ```

  As you can see from the preceding output, the computer chose the number `50`, while the user entered `23`.

For our final version of this game, we won't print out the computer number. That would be cheating! Right now, we're just testing.

Let's go ahead and add one condition—whether or not the first guess is correct. To do so, we'll have to verify `compnumber == usernumber`. We're going to test this again before going into the additional repetitions and logic, so we'll just say if it's true, then you win; if it's false, then you lose:

ch5_guess2.py

```
import random as rand
compnumber = rand.randint(1, 100)
usernumber = int(input('Choose a number between 1 and 100.
You'll get 5 guesses or you lose! '))

if compnumber == usernumber:
    print('You win!')
else:
    print('You lose!')
```

Let's just say I lost on the first try when I ran it. I'm not going to run this until I do win, however, because that could take, well, 100 tries or more. Here's what that looks like when you run the program:

```
Choose a number between 1 and 100. You'll get 5 guesses or you
lose! 35
You lose!
```

Now let's talk about repeating a line of code. We're giving the user 5 guesses. *How can we do that in Python?*

In Python, we can use `for` loops to iterate through code. We know we get 5 guesses, so we'll have to use something like `for number in range(5):` to get us started with the logic, which is shown in the following code:

ch5_guess3.py

```
import random as rand
compnumber = rand.randint(1, 100)
i = 5
for number in range(5):
    usernumber = int(input('Choose a number between 1 and 100.
You have ' + str(i) + ' guesses left. '))
    if compnumber == usernumber:
        print('You win!')
    else:
        i = i - 1
print('You're out of guesses! You lose! ')
```

From the preceding code, *did you notice that* `i` *variable?* We are using that variable so that the user knows how many guesses they have left. So if we had 5 guesses, the code will start at `i = 5`; then, if the user is wrong, it will use the line `i = i - 1`, which alerts the user they now have 4 guesses left, and so on. Take a look at what happens when we run that program:

```
Choose a number between 1 and 100. You have 5 guesses left. 14
Choose a number between 1 and 100. You have 4 guesses left. 98
Choose a number between 1 and 100. You have 3 guesses left. 48
Choose a number between 1 and 100. You have 2 guesses left. 12
Choose a number between 1 and 100. You have 1 guesses left. 54
You're out of guesses! You lose!
```

Now, we're not really being fair. As mentioned earlier, we want to give the user a hint each time they attempt a guess. Now that we have the condition checking whether they are equal, we can add an `elif` condition to check whether it's larger or smaller. The following code shows this:

ch5_guess4.py

```
import random as rand
compnumber = rand.randint(1, 100)
i = 5
for number in range(5):
    usernumber = int(input('Choose a number between 1 and 100. 
You have ' + str(i) + ' guesses left. '))
    if compnumber == usernumber:
        print('You win!')
        exit()
    elif compnumber > usernumber:
        print('Your number is too small!')
        i = i - 1
    elif compnumber < usernumber:
        print('Your number is too large!')
        i = i - 1
print('You're out of guesses! You lose! ')
```

The preceding code now provides some feedback to the user. If the number was greater than the computer-generated number, the user receives the feedback `'Your number is too large!'`, and if the user number is less than the computer-generated number, then they receive the feedback `'Your number is too small!'`. We also used an `exit()` code if the user wins. That's because we want the game to stop when we win.

This gives us a fighting chance to win this game, take a look at what the output looks like now:

```
Choose a number between 1 and 100. You have 5 guesses left. 50
Your number is too small!
Choose a number between 1 and 100. You have 4 guesses left. 75
Your number is too large!
Choose a number between 1 and 100. You have 3 guesses left. 65
Your number is too small!
Choose a number between 1 and 100. You have 2 guesses left. 70
Your number is too large!
Choose a number between 1 and 100. You have 1 guesses left. 68
You win!
```

Now look at what happens when we lose the game:

```
Choose a number between 1 and 100. You have 5 guesses left. 10
Your number is too small!
Choose a number between 1 and 100. You have 4 guesses left. 40
Your number is too large!
Choose a number between 1 and 100. You have 3 guesses left. 20
Your number is too small!
Choose a number between 1 and 100. You have 2 guesses left. 30
Your number is too small!
Choose a number between 1 and 100. You have 1 guesses left. 35
Your number is too large!
You're out of guesses! You lose!
```

As you can see, you get a different final message. I confess it took me quite a few tries to win a game so I could get the output that follows, but you can see the game where the second guess was correct:

```
Choose a number between 1 and 100. You have 5 guesses left. 10
Your number is too small!
Choose a number between 1 and 100. You have 4 guesses left. 90
You win!
```

We are going to stop this game with that last algorithm. We could actually make this game better if we wanted to, but it does the job that we needed it to do. Some of the changes that you could consider making to your game are as follows:

- Adding an option that alerts the user of a number already guessed.
- Adding an option that alerts the user that they ignored a previous hint (so if the user gave a number that was too small and gave one that was even smaller, the computer would alert them).

I'm sure there are more customizations that you could try. But for now, we went through that problem and followed the points that we should consider when analyzing problems:

1. We read and understood the problem.
2. We identified the purpose—creating a computer player versus user player guessing game.
3. We identified the constraints of the problem—the range of numbers, the number of guesses, and providing hints.
4. We created a decision flowchart.
5. We wrote and established an algorithm for the problem.
6. We looked at how to create a simple algorithm that would iterate rather than having to write each condition individually.
7. We tested the algorithm at multiple points.
8. We verified that the algorithm ran accurately for both wins and losses.

What you don't get to see here is the number of errors I went through before I got to the algorithms shown. While writing, I had to use the preceding steps to help me identify errors, check the best algorithms, and iterate through the programs. This is a process that we'll continue to use.

Summary

In this chapter, we discussed problem definition, decomposition, and analysis. We used problems to help us go through the process of identifying problems, decomposing them into the relevant parts and identifying constraints, and analyzing our algorithms. We used flowcharts to help us learn about decision-making when designing algorithms and how to organize ideas.

We learned to test our algorithms often. This provided us with the skills and understanding to identify errors early rather than wait until we had too many lines of code, which made it hard to identify those errors. We used an online store and a guessing game to help us understand some of the functionalities available in Python. Throughout the process, we used Boolean code to verify inputs, we used nested `if` statements, and we learned about how to use dictionaries in solving the problems presented.

In addition, we got a chance to use a dictionary for an algorithm that used user input and variables. Using the algorithm gave us flexibility for defining some variables and editing the variables once we run the algorithm or within the algorithm.

In the next chapter, we will go in depth into the solution process and design, delving deeper into more complex problems and the Python language.

6
Designing Solutions and Solution Processes

In this chapter, we will design **solutions** to multiple problems, using previously learned content, such as **analysis** of the problem and the **computational thinking process**. We will incorporate **logical processing** to create visual representations of the decision process that will guide our algorithm design. Visual representations discussed include **diagrams**, **flow charts**, and other helpful processes. In this chapter, we will learn about the key elements of solution design; how to create, use, and apply diagrams in our solution processing and design; and we will look at applying the solution design process to various problems.

In this chapter, we will cover the following topics:

- Designing solutions
- Diagramming solutions
- Creating solutions

In order to further our knowledge of algorithms and solution design, we need to look more closely at the frontend of problems. We will begin by discussing the process of designing solutions in depth.

Technical requirements

You will need the latest version of Python to run the codes in this chapter. You will find the full source code used in this chapter here: `https://github.com/PacktPublishing/Applied-Computational-Thinking-with-Python/tree/master/Chapter06`

Designing solutions

When we are designing solutions, we often use a **design thinking model**, even if we don't always realize it. Design thinking is described by different models, but we'll be looking at the five-step model that is most commonly seen when using design thinking.

In conjunction with computational thinking, the design thinking process can help us evolve our ideas before we start diagramming solutions. It should be noted that we don't go through the design thinking process linearly, much like in computational thinking. Think about the steps in computational thinking:

- Problem decomposition
- Pattern recognition
- Abstraction
- Algorithm design

We have defined all these steps in previous chapters, most recently in the introduction to *Chapter 5*, *Exploring Problem Analysis*. Looking at them again, we know that we can go back to the decomposition as we're writing and designing the algorithm. That's what we mean by non-linear processes.

The design thinking model works the same way. It was designed by the Hasso-Plattner Institute of Design at Stanford University. The main steps of the model include the following:

- **Empathize**: Understand the problem from the audience or stakeholder perspective.
- **Define**: Identify the objectives, the decisions that need to be made, any biases introduced, and any details pertaining to the problem.

- **Ideate**: Brainstorm ideas, which go with the diagramming we'll be doing in the next section of this chapter.
- **Prototype**: Design the algorithm solution and check it often.
- **Test**: Check your algorithm often throughout the process and go back to previous steps as needed.

As you can see, I've adapted the design thinking model to align more with a computational thinking process. The main goal when we're using these models and combining them is to break down the harder problems into simpler chunks so we can solve and design the best algorithms. This does not take the place of computational thinking. It simply provides a better idea of how we can address the process. The following diagram can help demonstrate how the process may work:

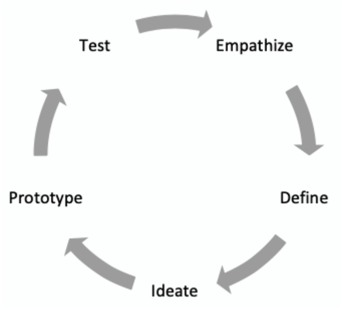


Figure 6.1 – Design thinking model

As you can see, rather than using the linear model most frequently shown, the preceding model shows the process as *cyclical*. That said, going back to *empathize* can happen from any point, so going back and forth between these steps is the best way to use the design thinking model.

Let's take a look at a scenario where we are using design thinking in conjunction with computational thinking.

Problem 1 - A marketing survey

Let's say you're working with a marketing firm and they ask you to put together a survey to gather feedback about a website. Here are some of the steps you may go through:

1. **Identifying stakeholders**: This includes the people you'll survey as well as the people who will use the information after the survey, for example.
2. **Identifying questions**: This is where you define what information you hope to find from the survey.

3. **Designing the survey**: This includes not only the questions you identified but the aesthetics of what the survey will look like.
4. **Information gathering**: This is where you decide how you will communicate with the people who will fill out the survey, such as email, a link on a website, or similar.
5. **Data analysis**: You can write a Python algorithm to help you with data analysis, including creating tables and graphs based on the data collected.
6. **Data sharing**: This is where you will plan the visuals, reports, and data presentation to the original stakeholders.

Let's be clear: this is an oversimplification of the process. But let's say you realize you need to include another group for the survey. Say you were only initially getting feedback from students at a school but realized you wanted to add teachers and parents. Well, then you would go back to *step 1* and identify in which ways the rest of your information would be affected. You may want to change the look of the survey or add a different one for adults versus children. You may need to add questions that are for only one group, which affects your decision-making in the algorithm for the survey.

Now let's take a look at these steps within the *design thinking* process.

For our problem, identifying the stakeholders and questions are part of *steps 1*, *2*, and *3* of the design thinking model: *empathize*, *define*, and *ideate*. Building the algorithm is both part of *prototype* and *test*, which are *steps 4* and *5*. Adding people to the survey takes us back to *steps 1–3*. And the cycle repeats until we have a working algorithm for our scenarios. Throughout the computational thinking model and using its elements, you'll use the design thinking process embedded within. It's a natural part of the decision-making process.

Now that we've looked at the design thinking model, let's take a look at how to visually represent the decision-making using diagramming solutions.

Diagramming solutions

When we are designing algorithms, we often use diagrams and flowcharts to help us analyze the process and visually see where our decisions are made. These diagrams allow us to create better algorithms. You'll remember that we created a flowchart in *Chapter 5*, *Exploring Problem Analysis*, when we were building a store (*Figure 5.1* and *Figure 5.2*).

The process of creating these diagrams varies by developer or coder. For example, I usually create a brainstorm for the problem, then a flowchart from that information. To look at that process, let's go back to our survey problem from earlier in this chapter. Look at the following brainstorm. It is not complete, as you can add a lot of sub-topics. This brainstorm assumes we are surveying stakeholders to evaluate and share feedback on a school website.

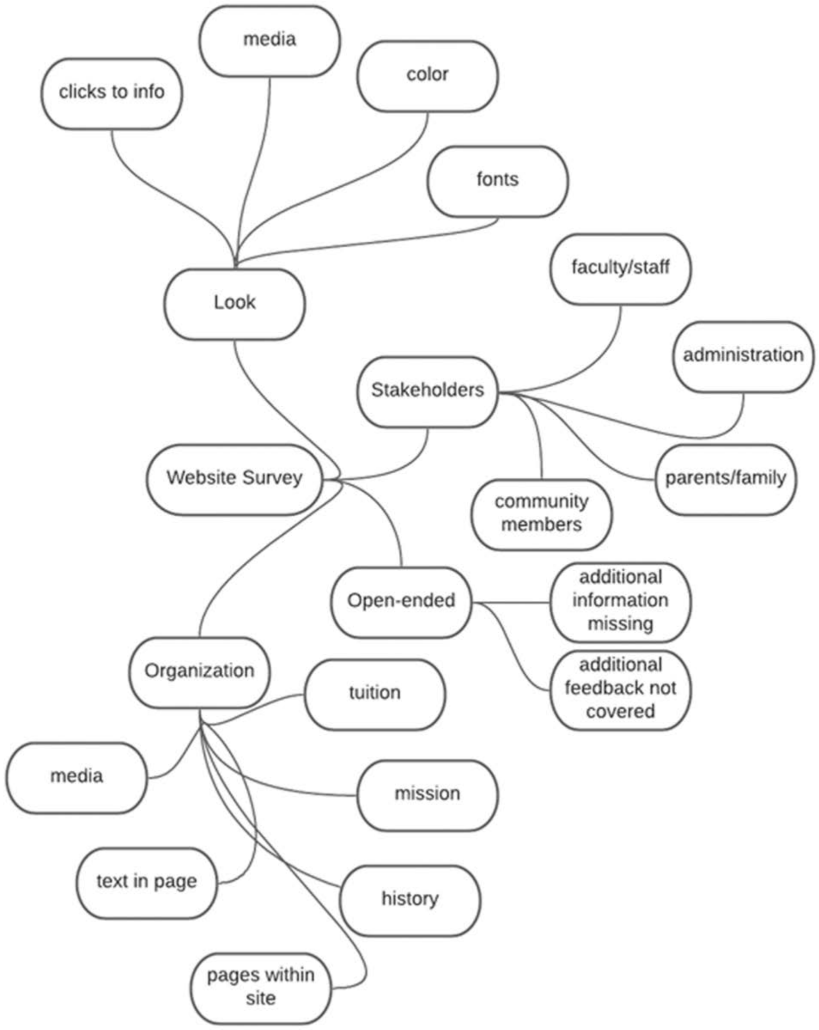


Figure 6.2 – Brainstorm diagram

As you can see from the diagram, there are many considerations to be made. The actual survey design may be provided to us as programmers or we may take part in designing the survey. If we have the survey, our brainstorm may look different, as we navigate through the questions and decide how to best place them within the algorithm. This is part of the empathizing process. We are looking at our information from multiple angles, from multiple stakeholders' perspectives, and deciding how we'll write an algorithm to help us get to where we need. The purpose of an informal diagram such as the brainstorm is that it allows us to begin organizing ideas before trying to create a more detailed and organized flowchart. When we work on the diagram, we are defining and ideating our algorithm. That's why it's important to sketch out our plans before beginning to code directly.

With regard to flowcharts, we saw a few in the last chapter when discussing the creation of a store in Python. Now let's take a look at a flowchart for decision-making based on some decisions.

It is important to note that surveys can be difficult to create from scratch. Part of the reason is that there may be questions that depend on each other. For example, let's say you ask the user to state whether they approve of the color choices or not. If they do, you can move on. But if they don't, you may want to provide other color schemes for review. That question would only appear for those who choose the **No** option. Our flowchart could be rather complicated if we were to tackle all the information for our brainstorm, so we'll focus on a few questions within the *Look* category of the brainstorm. Take a look at the following flowchart:

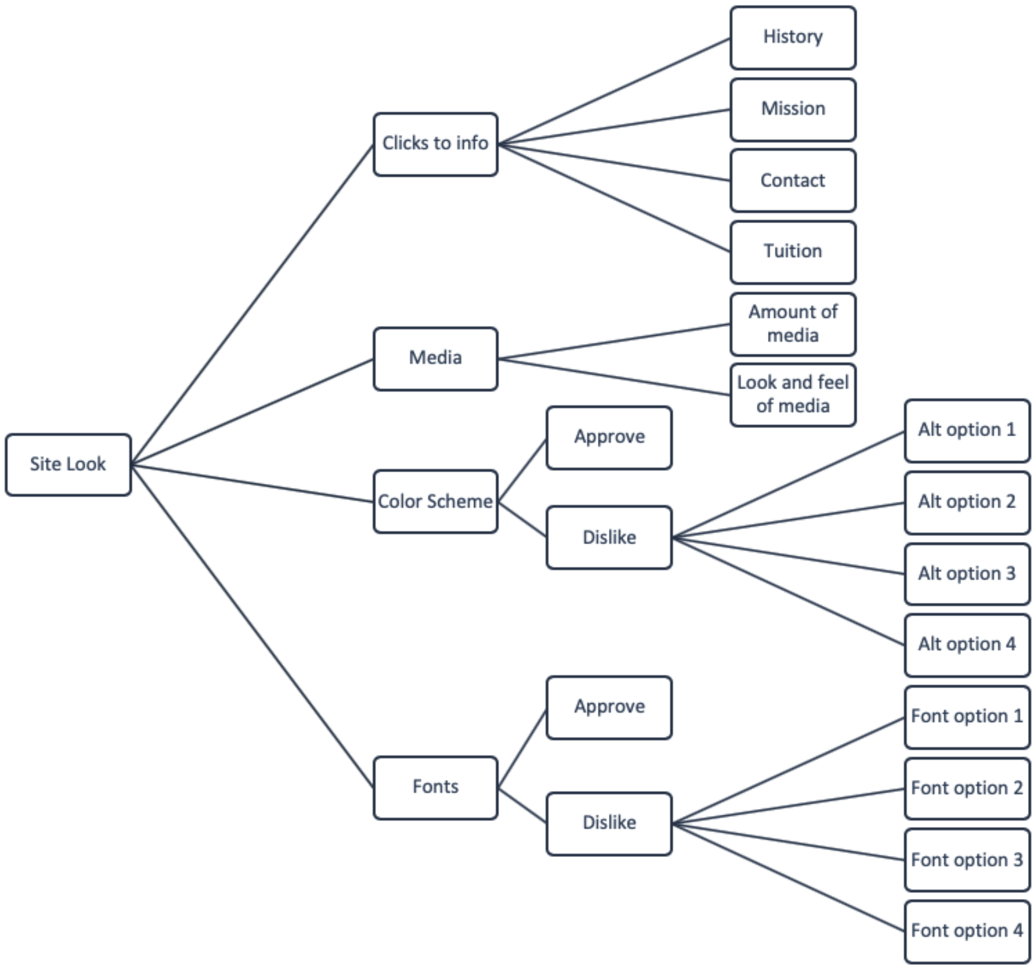


Figure 6.3 – Flowchart for one element of the survey

As you can see from the flowchart, some things are not clearly visible, such as what happens when you complete one question, where you go after each decision, and so on. When I create flowcharts, I sometimes add arrows to help me see what happens after each step. The following flowchart shows some of the arrows added:

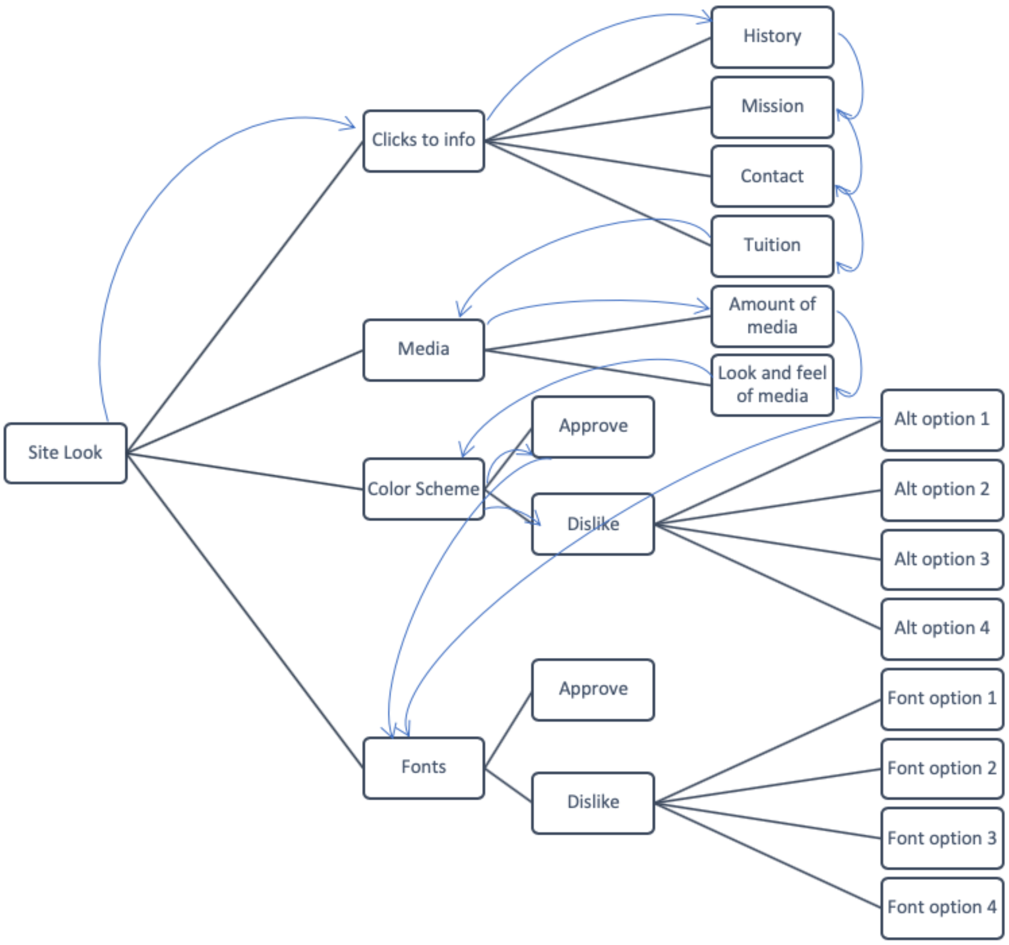


Figure 6.4 – Flowchart with arrows

As can be seen from the preceding flowchart, not all arrows are added, but look closely at **Color Scheme**. If a user approves of the color scheme, then they go directly to the **Fonts** section. If they don't, they are shown options. Assuming one option is shown at a time, then the user would go to **Fonts** after they choose one they like. It is also possible to add a prompt that asks the user if they'd like to see the options again, which would bring them back to the **Alt option 1**. Arrows can be added to show those details.

It all depends on what is easiest for you as a developer and programmer to understand yourself. Think of these as your journal notes if you were a writer. The way you organize your ideas can be personal, just make sure your end result and algorithm can be easily used by the people it is intended for.

Now let's take a look at how to put everything together and create solutions to some problems.

Creating solutions

When we are presented with problems, we want to create solutions that address the information we have been provided, with an algorithm that provides everything needed and that is easily understood by the user. In this section, we'll take the content we've been learning in this chapter in order to design solutions to problems.

As we create these solutions using our brainstorms and flowcharts, we should be considering the following:

- *Does the solution we have planned address the problem?*
- *Does the solution design show a clear path for the algorithm to be successful?*

And if the answers to those questions are yes, then we can start coding the solution. Remember, we need to test the algorithm as often as we can. Here are some things to keep in mind when writing the algorithm:

- Add comments to identify sections you may need to go back to and that clearly help to identify and define your variables, dictionaries, functions, and any key components.
- Check that you don't have any errors, such as those discussed in *Chapter 5, Exploring Problem Analysis*.
- Run your program as often as possible to test for errors.

For the solution process, we're going to use a slightly different problem than the survey we were working on earlier in the chapter. We will tackle components you can use for that problem as we go through this book, such as adding images, showing graphics, and more. But for now, let's stick with some more basic Python in order to practice the process of creating a solution.

Problem 2 - Pizza order

I know – food. But it's one of the best ways to demonstrate logic and algorithm creation, so bear with me. Let's say we have a pizzeria. We sell only one type of crust because we're a specialty kind of place. We sell two different sizes of pizza: personal and family. There are two sauce options: marinara and garlic cream. There are three cheese options: no cheese, regular cheese, and extra cheese.

There are five toppings to choose from (I'm limiting those because we're just learning the process): mushrooms, pepperoni, Italian sausage, onions, and peppers. And no, we're not putting olives anywhere near my pizzeria.

Let's break down that problem. We want an algorithm to capture the options chosen by the user to order their pizza. Things we're not going to take into consideration right now are cost and additional items in the order, such as an additional pizza, beverages, desserts, and so on.

Here's what we know:

- **Size**: personal or family
- **Sauce**: marinara or garlic cream
- **Cheese**: no cheese, regular cheese, extra cheese
- **Toppings**: mushrooms, pepperoni, Italian sausage, onions, peppers

Now that we have that, let's look at a flowchart with the information:

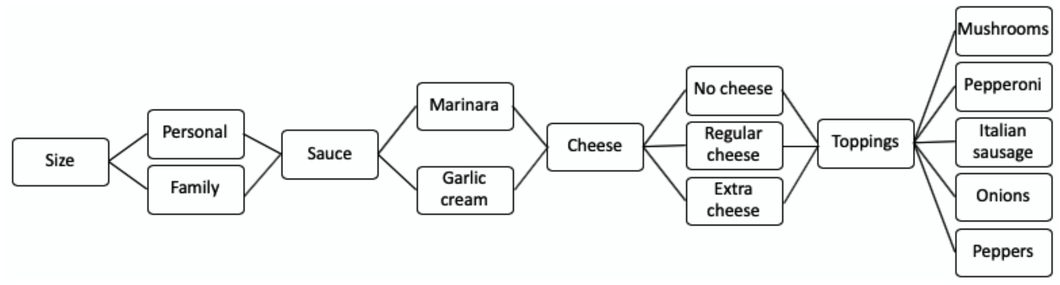


Figure 6.5 – Flowchart for pizzeria decision-making

As you can see, the figure shows a fairly *linear decision-making process* for this particular problem. One thing we haven't considered is asking the user if they wish to make any changes. That may need to happen at each step. Say you changed your mind while choosing cheese to go for a marinara instead of a garlic cream sauce. You'll need to have a way to go back, so we'll need to keep that in mind as we create the algorithm.

Keep in mind that we're sticking to text codes currently, so we'll use input from the user in numbers and letters for now. However, there are ways to incorporate Python into more robust algorithms that incorporate images, buttons, and more.

Take a look at the following snippet from the algorithm:

ch6_pizzeria.py

```
#Get input for your variables for size and sauce first.

size_choice = str(input("Is this a personal or family pizza? Type personal or family. "))

sauce_choice = str(input("Which sauce would you like? Marinara or garlic cream? Type m for marinara and g for garlic cream. "))
if sauce_choice == "g":
    sauce = "garlic cream"
else:
    sauce = "marinara"

#The cheese choice will dictate a few more options. Define the variable first.
cheese_choice = str(input("Would you like cheese on your pizza? Type y for yes and n for no. "))
```

Notice in the snippet that we defined the size and the sauce first. I will reiterate here that there are other ways to tackle this particular logic process. For example, we can save some of the variables to dictionaries and work with arrays. For now, we're using what we've learned so far to create our algorithms, but we'll get a chance to learn about other approaches later in this book.

The preceding snippet has a final choice of cheese. Regardless of the option here, we need to make decisions on toppings. That will need to happen twice since we'll need it for both yes and no.

Take a look at the following snippet with those options, which is a continuation of the preceding code:

ch6_Pizzeria.py

```
#Toppings need to happen whether or not you want cheese.
if cheese_choice == "y":
    cheese2_choice = str(input("Would you like regular cheese
or extra cheese? Type r for regular and e for extra cheese. "))
    if cheese2_choice == "r":
        cheese = "regular cheese"
    else:
        cheese = "extra cheese"
    toppings1_input = str(input("Would you like mushrooms on
your pizza? Type y for yes and n for no. "))
    if toppings1_input == "y":
        toppings1 = "mushrooms"
    else:
        toppings1 = "no mushrooms"
else:
    cheese = "no cheese"
if cheese_choice == "n":
    toppings1_input = str(input("Would you like mushrooms on
your pizza? Type y for yes and n for no. "))
    if toppings1_input == "y":
        toppings1 = "mushrooms"
    else:
        toppings1 = "no mushrooms"
print("You want a " + size_choice + " pizza with " + sauce + "
sauce, " + cheese + ", and " + toppings1 + ".")
```

As you can see from the snippet, we only worked with mushrooms. The output for this particular code after choosing family size, garlic sauce, regular cheese, and mushrooms looks as follows:

```
Is this a personal or family pizza? Type personal or family.
family
Which sauce would you like? Marinara or garlic cream? Type m
for marinara and g for garlic cream. g
```

```
Would you like cheese on your pizza? Type y for yes and n for no. y
Would you like regular cheese or extra cheese? Type r for regular and e for extra cheese. r
Would you like mushrooms on your pizza? Type y for yes and n for no. y
You want a family pizza with garlic cream sauce, regular cheese, and mushrooms.
```

Using the code provided and taking a look at the output, try to put together the rest of the code for the remaining four ingredients. And I guess that if you are creating your own pizza, you're welcome to change the options provided here. Just keep the olives to yourself.

Now, as mentioned before, we may need to go back and make changes. Let's take a look at a snippet that does that for you:

ch6_Pizzeria2.py

```
ready_end = str(input("Do you need to make any changes? Type y for yes and n for no. "))
if ready_end == "y":
    size_choice = str(input("Is this a personal or family pizza? Type personal or family. "))

    sauce_choice = str(input("Which sauce would you like? Marinara or garlic cream? Type m for marinara and g for garlic cream. "))
    if sauce_choice == "g":
        sauce = "garlic cream"
    else:
        sauce = "marinara"

    cheese_choice = str(input("Would you like cheese on your pizza? Type y for yes and n for no. "))
```

As you can see from the snippet of code, there is a decision that needs to be made about the changes needed. If yes, then we present the questions again. If no, then we print the choices for the user. Take a look at the following output with the fully run program:

```
Is this a personal or family pizza? Type personal or family. family
```

```
Which sauce would you like? Marinara or garlic cream? Type m
for marinara and g for garlic cream. g
Would you like cheese on your pizza? Type y for yes and n for
no. n
Would you like mushrooms on your pizza? Type y for yes and n
for no. y
Do you need to make any changes? Type y for yes and n for no. y
Is this a personal or family pizza? Type 1 for personal and 2
for family. family
Which sauce would you like? Marinara or garlic cream? Type m
for marinara and g for garlic cream. m
Would you like cheese on your pizza? Type y for yes and n for
no. n
Would you like mushrooms on your pizza? Type y for yes and n
for no. y
You want a family pizza with marinara sauce, no cheese, and
mushrooms.
```

As shown in the code, the questions were asked twice because we made a change in our options. Depending on how often you want to ask that question, you'll need to continue to repeat some of this code. There are ways to simplify that, so we'll go over those options in more depth in our Python language program chapter (*Chapter 8*, *Introduction to Python*) and later chapters.

Before we move on, let's take a look at one more problem to go through the design process again.

Problem 3 - Delays and Python

One of my first problems in Python was to create an algorithm that would react differently depending on the color chosen. This is similar to what you'd have if you were creating a traffic light. Each light has a different delay. So let's create an algorithm that addresses that. We'll make it a user-chosen color between green, yellow, and red, just to keep the traffic light theme. So let's put some assumptions together:

- Green will mean a 5-second delay
- Yellow will mean a 2-second delay
- Red will mean a 4-second delay

There's no reason for these specific delays; I just wanted to keep them all under 5 seconds. Now, let's say that we're playing a game and the user has to choose a color. If they choose yellow or red, they'll get a delay and then will be asked again. The goal is to get a *You win! You can go now* message from the program. So let's create a flowchart for this:

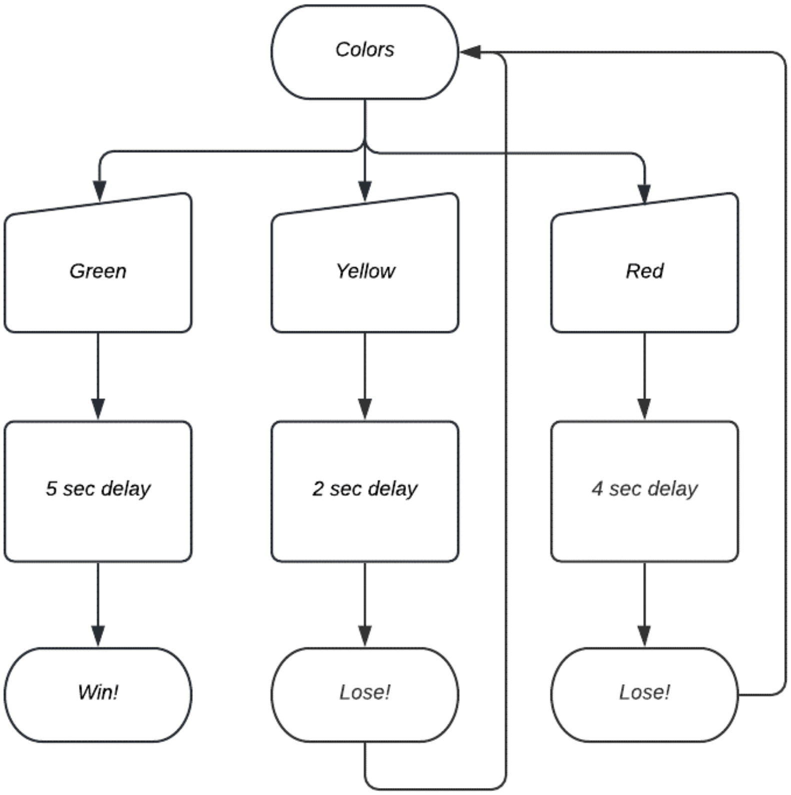


Figure 6.6 – Flowchart for traffic light game

As you can see from the flowchart, the game restarts if you choose yellow or red. Now that we have the basics of what the game will look like, we have to code it.

Important Note:

To be able to use delays, we'll need to import the `time` library. Use the code `import time` to do so. To include a delay, we use the code `time.sleep()`.

Let's take a look at a snippet of our code:

ch6_sleep.py

```
import time
print("Let's play a game. Choose a color to learn your destiny.
Choose wisely or you'll have to start over. ")
i = 0
while i < 4:
    color = str(input("Choose a color: red, green, or yellow.
"))
    if color == "green":
        print("You must wait 5 seconds to learn your fate.")
        time.sleep(5)
        print("You win! Excellent choice!")
        break
    elif color == "yellow":
        print("You must wait 2 seconds to learn your fate.")
        time.sleep(2)
        print("You lose! You must start over.")
        i = i + 1
    else:
        print("You must wait 4 seconds to learn your fate.")
        time.sleep(4)
        print("You lose! You must start over.")
        i = i + 1
```

As you can see, the algorithm contains some of the code we've looked at in previous chapters when talking about loops, Boolean statements, and more. This particular code returns to the beginning for three rounds if the user has not won the game. We used an `if-elif-else` statement to go through the color scenarios. The output of the game playing three rounds looks as follows:

```
Let's play a game. Choose a color to learn your destiny. Choose
wisely or you'll have to start over.
Choose a color: red, green, or yellow. yellow
You must wait 2 seconds to learn your fate.
You lose! You must start over.
Choose a color: red, green, or yellow. red
```

```
You must wait 4 seconds to learn your fate.
You lose! You must start over.
Choose a color: red, green, or yellow. green
You must wait 5 seconds to learn your fate.
You win! Excellent choice!
```

As you can see from the game output, all three rounds were played. Each delay happened according to the statement, which you'll have to test for yourself since I can't show time delays with text.

Having the flowchart made creating this algorithm simpler than if I'd started coding as soon as I'd read the problem. It's important to get used to fleshing out the processes you'll need prior to writing your algorithms. Designing solutions can be a long and tedious process, but the more organized we are at the start, the better our algorithms will be.

Summary

In this chapter, we discussed how to design, diagram, and create solutions to problems. We went over the non-linear process of design thinking in order to understand how to best design solutions. The design thinking model is a five-step process: **Empathize**, **Define**, **Ideate**, **Prototype**, and **Test**. Using this five-step process within the computational thinking process can help us to avoid many problems and pitfalls.

We also created brainstorms and flowcharts to establish the decision-making process of our algorithms to solve problems.

In the next chapter, we will use our knowledge of algorithm design and designing solutions in order to identify challenges within solutions and debug programs.

7
Identifying Challenges within Solutions

In this chapter, we will be evaluating algorithms and diagrams as we learn to navigate through some common errors and to determine whether possible adjustments can be made to an existing algorithm to simplify it. We will evaluate the solutions based on the problem description to verify whether the solution aligns with the problem. We will be learning about identifying pitfalls in the solution design process. As a note, we will expand on the content of this chapter, later in this book, in *Section 2*, *Applying Python and Computational Thinking*, and *Section 3*, *Data Processing, Analysis, and Applications Using Computational Thinking and Python*, of this book as we dive deeper into the **Python** programming language.

To learn about debugging, let's remind ourselves that the computational thinking process is not linear. Even when we are working from the original problem, we will sometimes redefine the problem or need to adjust the generalization due to a change in the population our algorithm is for or if we want to tweak our design of the algorithm. But sometimes, we come at problems after an algorithm has been designed and used. Depending on our roles, we'll be evaluating algorithms for errors, changes needed, and so on. Understanding how to find and analyze errors can help us, regardless of whether we are absolute Python beginners or deep in our careers.

Throughout this chapter, you will learn how to identify and fix bugs in your program and how to avoid pitfalls in the algorithm designs.

In this chapter, we will cover the following topics:

- Identifying errors in algorithm design
- Debugging algorithms
- Comparing solutions
- Refining and redefining solutions

Technical requirements

You will need the latest version of Python to run the codes in this chapter. You will find the full source code used in this chapter here: `https://github.com/PacktPublishing/Applied-Computational-Thinking-with-Python/tree/master/Chapter07`

Identifying errors in algorithm design

Errors in algorithms are just a fact of life for any coder. It's important to get comfortable with making mistakes. As mentioned in *Chapter 5*, *Exploring Problem Analysis*, and *Chapter 6*, *Solution Processes and Design*, it's good practice to test your algorithm and test it often. Waiting until you have finished hundreds or thousands of lines of code to test something is a recipe for disaster. And yes, I was once working on copying a game and did not test at all. Not until I had all 4,585 lines copied. I was young. Truth be told, I never found the error I made. I started over and started testing at every corner. The second time was successful, but I'd wasted weeks copying everything (it was from a book—GitHub wasn't a thing yet) and then trying to figure out the errors. So please don't be me. Please test your algorithms.

Now, before moving on to debugging and working with codes, let's take a look at the errors we can encounter when solving problems.

In this section, we'll focus on the following two broad categories of errors: syntax errors and logic errors.

Syntax errors

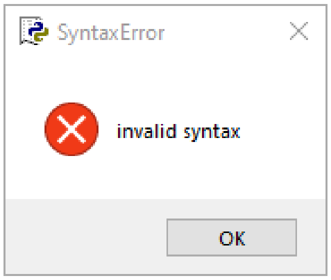
Syntax errors are sometimes called parsing errors. They're errors we create when we forget to indent, add a colon, add quotation marks for strings, and so on. Let's have a look at the different types of syntax errors in the following sections.

Using colons

Colons are used in Python to separate conditions, create loops, and more. The colon is a way to tell the algorithm that the next thing is part of this particular block of code. When we introduce colons in Python, it automatically indents the next line in our code. But if we forget to include a colon where it is needed, the program will not run successfully. Let's take a look at a syntax error:

```
for i in range(1, 10)
    print(i)
```

If we run this code, we get an error message that says *invalid syntax*. The following screenshot shows the pop-up window that appears when we try to run this program:


SyntaxError

invalid syntax

Figure 7.1 – Error pop-up window

If we run this from the Python shell instead, here's how that error appears:

```
SyntaxError: invalid syntax
```

As you can see, the Python program alerts us when we have code that contains errors.

Notice that there is a colon missing after the range in the code. Now, take a look at the fixed syntax in the following code:

ch7_syntaxerror1.py

```
for i in range(1, 10):
    print(i)
```

When we run the fixed code, the program runs and prints the numbers 1 through 9, shown as follows:

```
1
2
3
4
5
6
7
8
9
```

You may recall that the range function does not include the upper endpoint. If we wanted to print the number 10, our range would have needed to be `range(1, 11)`.

Now, let's take a look at other punctuation used in Python that can lead to some errors, namely, parentheses, nested parentheses, and brackets.

Using nested parentheses and brackets

In addition to errors in syntax that involve colons, which are my most common error, there are also errors when we have nested parentheses. We must always check that every open parenthesis has a closing parenthesis. The same is true for brackets. Let's look at the following code, which contains an error with the parentheses:

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

As you can see, there are two open parentheses in the name definition, but only one closing parenthesis. When we run that program, we get an invalid syntax error from Python. Here's what happens when we run that program in the Python shell or interpreter:

```
SyntaxError: invalid syntax
```

Now here's the same code without the error, notice that we even took the `str()` out because it was not needed, which resulted in simplifying our code and eliminating the error at the same time

ch7_syntaxerror2.py

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

Now when we run the code, the program asks for the name then prints it. The output looks as follows:

```
What is your name? Monique
Monique
```

As you can see, the program now runs without problems.

In *Chapter 3*, *Understanding Algorithms and Algorithmic Thinking*, we used a dictionary to create a menu with pricing for each menu item. Dictionaries contain brackets to denote when the dictionary begins and when it ends. Let's look at a few lines of code:

```
cars = {
    "Hyundai": "Kona",
    "Honda": "CR-V",
    "Toyota": "Camry"

print(cars)
```

If we look at the program, the dictionary is missing the closing bracket, }, so we get a syntax error, the same as in our previous examples. The following snippet shows the corrected program:

ch7_syntaxerror3.py

```
cars = {
    "Hyundai": "Kona",
    "Honda": "CR-V",
    "Toyota": "Camry"
    }
print(cars)
```

As you can see, once the bracket has been added, the program will run and print the following output:

```
{'Hyundai': 'Kona', 'Honda': 'CR-V', 'Toyota': 'Camry'}
```

Each entry from the dictionary is printed in one line, divided by a comma. It is helpful to add `print` statements as we are writing algorithms to ensure we do not have any errors. I typically then remove unnecessary print functions once I've tested them, but they do come in handy when we're writing long algorithms and need to test them to avoid issues.

There are many other errors we can incorporate while writing algorithms in Python. Let's take a look at a few more syntax errors.

Other syntax errors

There are many other errors in syntax that can be introduced, especially in longer programs. If you look at the dictionary we just used, forgetting a comma will also create a syntax error, for example. Generally, these syntax errors are quickly identified when we try to run a program. Python will highlight where an indentation is expected or when a bracket is missing. Syntax errors are typically easy to identify, but there are many other types of errors.

Errors in logic

In *Chapter 4*, *Understanding Logical Reasoning*, we discussed logic errors that we can encounter:

- Using the wrong variable in an equation or statement
- Using the wrong operator to test conditions
- Using wrong indentation when checking for conditions

Now we'll look at other errors in logic that have a specific callout from Python and what each error represents.

Errors in logic are also called runtime errors. The following table shows some of the built-in errors in Python and what they represent:

Exception	**Cause/Description**
`AssertionError`	Failure of an `assert` statement
`AttributeError`	Failure of an attribute assignment or reference
`EOFError`	Raised when the `input()` function hits an end-of-file condition
`FloatingPointError`	Failure of a floating-point operation
`GeneratorExit`	Raised when a generator's `close()` method is called
`ImportError`	Raised when an import module is not found
`IndexError`	Raised when the index of a sequence is out of range
`KeyError`	Raised when a key is not found in a dictionary
`KeyboardInterrupt`	Raised when the user hits the interrupt key (*Ctrl* + *C* or *Delete*).
`MemoryError`	Raised when an operation runs out of memory
`NameError`	Raised when a variable is not found in local or global scope
`NotImplementedError`	Raised by abstract methods
`OSError`	Raised when system operation causes system-related error
`OverflowError`	Raised when the result of an arithmetic operation is too large to be represented
`ReferenceError`	Raised when a weak reference proxy is used to access a garbage-collected referent
`RuntimeError`	Raised when an error does not fall under any other category
`StopIteration`	Raised by a `next()` function to indicate that there is no further item to be returned by the iterator

Exception	Cause/Description
`SyntaxError`	Raised by a parser when a syntax error is encountered
`IndentationError`	Raised when there is incorrect indentation
`TabError`	Raised when indentation consists of inconsistent tabs and spaces
`SystemError`	Raised when the interpreter detects an internal error
`SystemExit`	Raised by the `sys.exit()` function
`TypeError`	Raised when a function or operation is applied to an object of an incorrect type
`UnboundLocalError`	Raised when a reference is made to a local variable in a function or method, but no value has been bound to that variable
`UnicodeError`	Raised when a Unicode-related encoding or decoding error occurs
`UnicodeEncodeError`	Raised when a Unicode-related error occurs during encoding
`UnicodeTranslateError`	Raised when a Unicode-related error occurs during translating
`ValueError`	Raised when a function gets an argument of a correct type but improper value
`ZeroDivisionError`	Raised when the second operand of division or modulo operation is zero

Table 7.1 - Table of exceptions and causes/descriptions

As you can see, there are many different types of errors that are flagged as exceptions in Python. You can get the list of Python exceptions by running the following code:

ch7_errors.py

```
print(dir(locals()['__builtins__']))
```

When we run that code, the output provides the error values as follows:

```
['ArithmeticError', 'AssertionError', 'AttributeError',
'BaseException', 'BlockingIOError', 'BrokenPipeError',
'BufferError', 'BytesWarning', 'ChildProcessError',
'ConnectionAbortedError', 'ConnectionError',
'ConnectionRefusedError', 'ConnectionResetError',
'DeprecationWarning', 'EOFError', 'Ellipsis',
'EnvironmentError', 'Exception', 'False', 'FileExistsError',
'FileNotFoundError', 'FloatingPointError', 'FutureWarning',
'GeneratorExit', 'IOError', 'ImportError', 'ImportWarning',
'IndentationError', 'IndexError', 'InterruptedError',
'IsADirectoryError', 'KeyError', 'KeyboardInterrupt',
'LookupError', 'MemoryError', 'ModuleNotFoundError',
'NameError', 'None', 'NotADirectoryError', 'NotImplemented',
'NotImplementedError', 'OSError', 'OverflowError',
'PendingDeprecationWarning', 'PermissionError',
'ProcessLookupError', 'RecursionError', 'ReferenceError',
'ResourceWarning', 'RuntimeError', 'RuntimeWarning',
'StopAsyncIteration', 'StopIteration', 'SyntaxError',
'SyntaxWarning', 'SystemError', 'SystemExit', 'TabError',
'TimeoutError', 'True', 'TypeError', 'UnboundLocalError',
'UnicodeDecodeError', 'UnicodeEncodeError', 'UnicodeError',
'UnicodeTranslateError', 'UnicodeWarning', 'UserWarning',
'ValueError', 'Warning', 'ZeroDivisionError', '__build_
class__', '__debug__', '__doc__', '__import__', '__loader__',
'__name__', '__package__', '__spec__', 'abs', 'all', 'any',
'ascii', 'bin', 'bool', 'breakpoint', 'bytearray', 'bytes',
'callable', 'chr', 'classmethod', 'compile', 'complex',
'copyright', 'credits', 'delattr', 'dict', 'dir', 'divmod',
'enumerate', 'eval', 'exec', 'exit', 'filter', 'float',
'format', 'frozenset', 'getattr', 'globals', 'hasattr',
'hash', 'help', 'hex', 'id', 'input', 'int', 'isinstance',
'issubclass', 'iter', 'len', 'license', 'list', 'locals',
'map', 'max', 'memoryview', 'min', 'next', 'object', 'oct',
'open', 'ord', 'pow', 'print', 'property', 'quit', 'range',
'repr', 'reversed', 'round', 'set', 'setattr', 'slice',
'sorted', 'staticmethod', 'str', 'sum', 'super', 'tuple',
'type', 'vars', 'zip']
```

As mentioned, these are built-in exceptions in Python. There is a way to define our own exceptions, but we will not be going into them in this particular book.

Please note that these are not the only errors we will encounter when we are programming. We can have errors due to our own mistakes in calculations, as we discussed in *Chapter 4*, *Understanding Logical Reasoning*. We can introduce errors in our *Boolean* logic. The goal is to avoid as many as we can so that our program runs without problems. And remember, test your algorithms and test them often.

Now let's look at a few algorithms with errors and try to identify the errors so we can correct them.

Debugging algorithms

There is a debugger we can run in Python using the `breakpoint()` function (which is built-in). We can introduce this code into our program and insert it where we are unsure of our code. Adding `breakpoint()` will then check for bugs and errors. When we run a `breakpoint()` function, we'll get a `pdb` output, which stands for **Python Debugger**. As a note, this built-in function appears in **Python 3.7** and newer versions. The previous debugger for **Python 3.6** and older was `pdb.set_trace()`.

When we run the debugger, we can use four commands:

- `c`: Continues the execution
- `q`: Quits the debugger/execution
- `n`: Steps to the next line within the function
- `s`: Steps to the next line in this function or a called **function**

Let's take a look at a code and run each of the commands outlined:

ch7_debugger.py

```
number = 5
number2 = 'five'

print(number)

breakpoint()

print(number2)
```

Looking at this code, you can see the `breakpoint()` command after `print(number)`. The code will run normally until it reaches the `breakpoint()` command. At this stage, the execution stops. If we hit the *c* key, then it will just continue to run the program. Take a look at what that output looks like.

Notice in the code there are three dots between two slashes, `/…/`. This is done because the paths may differ from your computer to mine. Yours will include the full path where the program is located:

```
5
> /Users/.../Python/ch7_debugger.py(8)<module>()
-> print(number2)
(Pdb) c
five
```

As you can see, it went on to print the string, `five`, as it just continues the program. Now let's look at the output when we run the `q` command, which quits the program:

```
5
> /Users/.../Python/ch7_debugger.py(8)<module>()
-> print(number2)
(Pdb) q
Traceback (most recent call last):
  File "/Users/.../Python/ch7_debugger.py", line 8, in <module>
    print(number2)
  File "/Users/.../Python/ch7_debugger.py", line 8, in <module>
    print(number2)
bdb.BdbQuit
```

As you can see, once we use the `q` command, we get a **traceback error** because the program quits. It printed the line above the `breakpoint()` code, but not the second `print(number2)` command. Now, let's see what happens when we type `n`, which should take us to the next line:

```
5
> /Users/.../Python/ch7_debugger.py(8)<module>()
-> print(number2)
(Pdb) n
five
--Return--
> /Users/.../Python/ch7_debugger.py(8)<module>()->None
-> print(number2)
(Pdb)
```

As you can see, when we typed n, the program continued to run and printed the second command line. When it does so, you can see the `-> None` output and the code that ran: `print(number2)`. Finally, let's look at slightly altered code to see what happens when we use s while running the debugger:

ch7_debugger2.py

```
number = 5
number2 = 'five'

print(number)

breakpoint()

print(str(number) + number2)
```

When we run this program and the debugger, we get the following output if we use s:

```
5
> /Users/.../Python/ch7_debugger2.py(8)<module>()
-> print(number + " " + number2)
(Pdb) s
TypeError: unsupported operand type(s) for +: 'int' and 'str'
> /Users/.../Python/ch7_debugger2.py(8)<module>()
-> print(number + " " + number2)
(Pdb)
```

As you can see, the program encountered `TypeError` and provides more information. I tried to combine an integer and a string. So, we would need to fix that code to run it correctly. Before I do so, let's look at what happens when I try to continue the code with c:

```
5
> /Users/.../Python/ch7_debugger2.py(8)<module>()
-> print(number + " " + number2)
(Pdb) c
Traceback (most recent call last):
  File "/Users/.../Python/ch7_debugger2.py", line 8, in
<module>
```

```
    print(number + " " + number2)
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

As you can see, I can get the same information from both commands, with slightly different responses from the program. To fix the problem, I'd have to convert the number into a string, which I can do in the `print` line using the following code:

ch7_debugger3.py

```
number = 5
number2 = 'five'

print(number)

breakpoint()

print(str(number) + " " + number2)
```

Now that I've fixed the code so that the items in the print line are all strings, the output looks as follows when I use `c` to continue:

```
5
> /Users/.../Python/ch7_debugger3.py(8)<module>()
-> print(str(number) + " " + number2)
(Pdb) c
5 five
```

As you can see, the program now printed the correct information, combining the number as a string, with the `five` string. The double quotes add a space between them, which we have seen before, but will discuss again when we look at Python basics in *Chapter 8, Introduction to Python*.

Now, let's take a look at some solutions to the same problem so we can analyze them.

Comparing solutions

As we look at problems, I've mentioned that we have multiple ways of doing the same things in Python. Depending on what we are trying to accomplish, some commands may be better than others in our algorithms. Let's start by taking a look at a couple of solutions for one problem.

Problem 1 - Printing even numbers

You've been asked to write an algorithm that prints even numbers based on a range that the user provides. That is, if the user enters the range 2 through 20, then the program would print 2, 4, 6, 8, 10, 12, 14, 16, 18, and 20. Let's assume we want to include the endpoints if they are even.

Let's take a look at the first of two possible solutions. Remember, one solution may not be better than the other. A lot will depend on what the goal for your full algorithm is. *Is a list more appropriate? A dictionary? A function?* Those questions are important when we design solutions.

Algorithm solution 1 - Printing even numbers

Recall that we will be taking user input to create a list of even numbers given a range. Take a look at the following code, which asks the user for the input then prints out the numbers:

ch7_evenalgorithm1.py

```
print("This program will print the even numbers for any range
of numbers provided.")

endpoint1 = int(input("What is the lower endpoint of your
range? "))
endpoint2 = int(input("What is the upper endpoint of your
range? "))

endpoint2 = endpoint2 + 1

for i in range(endpoint1, endpoint2):
    if i % 2 == 0:
        print(i)

```

Notice that `endpoint2` was converted into `endpoint2 + 1`. That is because if we do not add `1`, then the upper endpoint will not be included if it is an even number. The program also begins with a printed message for the user that states what the program does.

When I run this program with the endpoints `2` and `6`, I get the following output:

```
This program will print the even numbers for any range of
numbers provided.
What is the lower endpoint of your range? 2
What is the upper endpoint of your range? 6
2
4
6
```

As you can see, both endpoints are even and included. If we run the program with the endpoints `3` and `9`, we get the following output:

```
This program will print the even numbers for any range of
numbers provided.
What is the lower endpoint of your range? 3
What is the upper endpoint of your range? 9
4
6
8
```

Even though the endpoint is technically `10` now, the upper limit of the range is not included, so the largest even number below `10` is `8`. Now, I can run this program for a much larger range, but the larger the range, the harder it is to scroll to get all the numbers. So, let's take a look at a different way to get our even numbers.

Algorithm solution 2 - Printing even numbers

As we saw from the previous example, each even number is being printed to a different line. Let's see whether we can change that and instead create a list. Lists in Python can be empty. We use any name for them, then equal them to items inside braces or just the empty braces.

For example, I can create an empty list called `evenNumbers = []`. Let's see what that looks like in the following algorithm:

ch7_evenalgorithm2.py

```
print("This program will print the even numbers for any range
of numbers provided.")

```

```
endpoint1 = int(input("What is the lower endpoint of your
range? "))
endpoint2 = int(input("What is the upper endpoint of your
range? "))

endpoint2 = endpoint2 + 1

evenNumbers = []

for i in range(endpoint1, endpoint2):
    if i % 2 == 0:
        evenNumbers.append(i)


print(evenNumbers)
```

You can see that the first few lines of code are the same. The only difference in this particular code is how the numbers are printed. The list is created before the `for` loop. Then, each of the numbers is appended to the list using the `evenNumbers.append(i)` code. Finally, we print our list to get the following output:

```
This program will print the even numbers for any range of
numbers provided.
What is the lower endpoint of your range? 2
What is the upper endpoint of your range? 10
[2, 4, 6, 8, 10]
```

As you can see, the even numbers are all included in one list, which is easier to read than if printed one at a time, one line at a time. *Imagine if you had to print even numbers in the range 300–1,000*. A list would make that easier to read when we run the program. The output would look as follows for the second algorithm:

```
This program will print the even numbers for any range of
numbers provided.
What is the lower endpoint of your range? 300
What is the upper endpoint of your range? 1000
[300, 302, 304, 306, 308, 310, 312, 314, 316, 318, 320, 322,
324, 326, 328, 330, 332, 334, 336, 338, 340, 342, 344, 346,
348, 350, 352, 354, 356, 358, 360, 362, 364, 366, 368, 370,
372, 374, 376, 378, 380, 382, 384, 386, 388, 390, 392, 394,
396, 398, 400, 402, 404, 406, 408, 410, 412, 414, 416, 418,
420, 422, 424, 426, 428, 430, 432, 434, 436, 438, 440, 442,
444, 446, 448, 450, 452, 454, 456, 458, 460, 462, 464, 466,
468, 470, 472, 474, 476, 478, 480, 482, 484, 486, 488, 490,
492, 494, 496, 498, 500, 502, 504, 506, 508, 510, 512, 514,
516, 518, 520, 522, 524, 526, 528, 530, 532, 534, 536, 538,
540, 542, 544, 546, 548, 550, 552, 554, 556, 558, 560, 562,
564, 566, 568, 570, 572, 574, 576, 578, 580, 582, 584, 586,
588, 590, 592, 594, 596, 598, 600, 602, 604, 606, 608, 610,
612, 614, 616, 618, 620, 622, 624, 626, 628, 630, 632, 634,
636, 638, 640, 642, 644, 646, 648, 650, 652, 654, 656, 658,
660, 662, 664, 666, 668, 670, 672, 674, 676, 678, 680, 682,
684, 686, 688, 690, 692, 694, 696, 698, 700, 702, 704, 706,
708, 710, 712, 714, 716, 718, 720, 722, 724, 726, 728, 730,
732, 734, 736, 738, 740, 742, 744, 746, 748, 750, 752, 754,
756, 758, 760, 762, 764, 766, 768, 770, 772, 774, 776, 778,
780, 782, 784, 786, 788, 790, 792, 794, 796, 798, 800, 802,
804, 806, 808, 810, 812, 814, 816, 818, 820, 822, 824, 826,
828, 830, 832, 834, 836, 838, 840, 842, 844, 846, 848, 850,
852, 854, 856, 858, 860, 862, 864, 866, 868, 870, 872, 874,
876, 878, 880, 882, 884, 886, 888, 890, 892, 894, 896, 898,
900, 902, 904, 906, 908, 910, 912, 914, 916, 918, 920, 922,
924, 926, 928, 930, 932, 934, 936, 938, 940, 942, 944, 946,
948, 950, 952, 954, 956, 958, 960, 962, 964, 966, 968, 970,
972, 974, 976, 978, 980, 982, 984, 986, 988, 990, 992, 994,
996, 998, 1000]
```

The reason I only printed this one versus the first algorithm is that the first algorithm would take pages, and we don't want to waste printed pages in this book. You can see that one is just easier to use and more appropriate than the other due to the ease of reading the larger group of numbers.

This is why we need to look at all of our algorithms and determine whether they are the best possible way to express what we need. While some algorithms work, they may not be the best solution, and sometimes that's okay. But other times, making some changes, sometimes as subtle as adding a couple of lines of code, as we did with *algorithm 2*, can change our output fairly dramatically and be much more helpful for us.

As we compared these two algorithms, we were also refining and redefining our solution, which we will do more of in the next section.

Refining and redefining solutions

If we look at algorithms long enough, we can always find ways to refine them and redefine them. Think about how many updates we get for apps on our phones. Someone is always playing with the apps, making them more secure, adding levels to games, updating the art files, and so on. As programmers/coders, we are always trying to make our work better.

We are going to start this section with an algorithm. The following program prints out the names of three pets:

ch7_pets1.py

```
cat = "Whiskers"
dog = "King Kong"
bird = "Pirate"

print("The cat's name is " + cat + ", the dog's name is " + dog
+ \
      ", and the bird's name is " + bird + ".")
```

This simple code has everything within it, so there's no user input this time. You can see the \ character used after `dog +` in the `print()` command. This backslash allows us to add the remaining code in the next line so we can more easily read it.

The output for the code looks as follows:

```
The cat's name is Whiskers, the dog's name is King Kong, and
the bird's name is Pirate.
```

As you can see, it's a simple sentence with the pet names.

Now, let's say we have a cat, dog, and bird, but their names are not the same. We can instead use a function that takes three arguments. Keep in mind we will go into all of the definitions and information on functions in *Chapter 8*, *Introduction to Python*. For now, let's look at what that algorithm could look like with a function. We'll name the function `myPets()`. Take a look at the following algorithm:

ch7_pets2.py

```
def myPets(cat, dog, bird):
    print("The cat's name is " + cat + ", the dog's name is " +
dog +\
          ", and the bird's name is " + bird + ".")
```

```

myPets(cat = "Whiskers", dog = "King Kong", bird = "Pirate")
```

The algorithm looks very similar to the previous one, except the definitions of the names are in the last line of the code. The function is called, using the information from that line to fill in the blanks from the definition in the algorithm lines above it. The output looks the same as the previous code:

```
The cat's name is Whiskers, the dog's name is King Kong, and
the bird's name is Pirate.
```

Now, as you can see, this only printed one function because we only provided information for one, but we can call the function as many times as we want to with as many values as we want. Take a look at this algorithm:

ch7_pets3.py

```
def myPets(cat, dog, bird):
    print("The cat's name is " + cat + ", the dog's name is " +
dog +\
          ", and the bird's name is " + bird + ".")

myPets(cat = "Whiskers", dog = "King Kong", bird = "Pirate")
myPets(cat = "Mimi", dog = "Jack", bird = "Peyo")
myPets(cat = "Softy", dog = "Leila", bird = "Oliver")
```

As you can see, the function will now be called three times. We only have one `print()` command, but the function definition means that `print()` command will be used any time the function is called. Take a look at what the output looks like:

```
The cat's name is Whiskers, the dog's name is King Kong, and
the bird's name is Pirate.
The cat's name is Mimi, the dog's name is Jack, and the bird's
name is Peyo.
The cat's name is Softy, the dog's name is Leila, and the
bird's name is Oliver.
```

Notice that three different sentences were printed with the three sets of pet names provided when we called the function.

When we're writing algorithms, it's important to take into consideration what we need now and what we might need later. Using the first algorithm was fine for one instance, but if we wanted to run the algorithm for every person in a community or every student in a classroom, for example, the second algorithm is more helpful. Redefining what we need and refining our algorithms helps us to improve what we get out of our programs.

Note that, as mentioned, we will talk about functions more in *Chapter 8*, *Introduction to Python*. One of the things we'll address is creating a function for an unknown number of arguments. *For example, what if I only had a dog and a bird?* We can address that with a few changes to the algorithm. We will look into that soon. For now, we know just a little more about why we need to sometimes compare algorithms and redefine and redesign them to better fit our needs.

Summary

In this chapter, we discussed errors in algorithm design and how to debug solutions. We also learned how to compare solutions and refine and redesign solutions when needed. After reading this chapter, you should know more about syntax errors in algorithms and how to use the debugger using the `breakpoint()` command in **Python 3.7** and above. The built-in debugger provides you with four courses of action: `c` = *continue*, `q` = *quit*, `n` = *next line*, and `s` = *step*.

Using the debugger allows us to identify pieces of code where we may have made mistakes. We can add this line to any place in our code to determine the issue.

We also looked at algorithms that provide the same output but using different code. By comparing algorithm solutions, we can identify which of them are more useful, what better suits our problem or situation, and why we should use one over the other. Remember that algorithms are lists of instructions. Knowing which instructions to use given the broader use of the algorithm is critical. Some solutions may be more appropriate for your problem than others. Take into consideration the algorithm's purpose, the snippets in the algorithm, and how they will be used within the bigger algorithm and make your determinations accordingly. Each problem and each solution is unique.

As we finish out *Section 1*, *Introduction to Computational Thinking*, of this book, we have learned about the computational thinking process, always looking at possible scenarios to help us to understand the usefulness of the process, how to brainstorm and create flowcharts for decisions, and how to design our algorithms. In *Section 2*, *Applying Python and Computational Thinking*, we will begin looking at the Python language more in depth so we can tackle more complex problems, such as those dealing with data and functions. We'll also take a more in-depth look at the Python programming language, and we'll apply the knowledge to multiple types of problems in the further chapters.

Section 2: Applying Python and Computational Thinking

There are multiple options when it comes to programming languages. The Python programming language is a powerful, open source, object-oriented programming language. It is a high-level programming language with an ever-growing number of libraries and packages that continue to expand its capabilities. We can use Python to program things from simple mathematical algorithms to complex data science algorithms, and even machine learning. Understanding Python also means understanding multiple areas of computational thinking.

In this section, you will learn how to use the computational thinking process when solving problems using the Python programming language.

This section comprises the following chapters:

8
Introduction to Python

In this chapter, we will be learning about the Python commands and functionalities while applying them to problems. As we delve into the first chapter of *Section 2*, *Applying Python and Computational Thinking*, we will be using more complex Python programming. In this chapter, we will focus more on language, while the remaining chapters will focus on application.

In this chapter, we will cover the following topics:

- Introducing Python
- Working with dictionaries and lists
- Using variables and functions
- Learning about files, data, and iteration
- Using object-oriented programming

As we dig deeper into the Python programming language, remember that some of the content has been covered in previous chapters as we looked at the computational thinking process, such as the use of dictionaries and functions. This chapter will allow you to find critical information more easily when looking for Python commands to meet your computational thinking problem needs.

Technical requirements

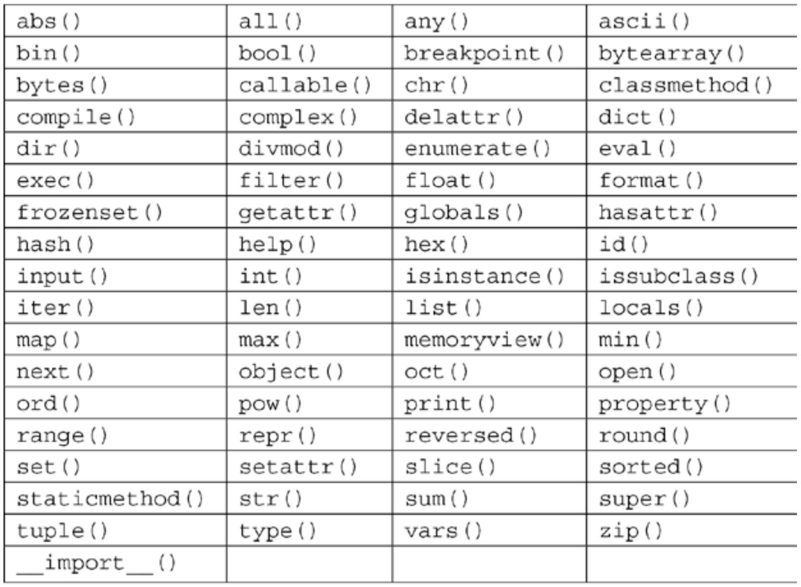
You will need the latest version of Python to run the codes in this chapter. You will find the full source code used in this chapter here: `https://github.com/PacktPublishing/Applied-Computational-Thinking-with-Python/tree/master/Chapter08`

Introducing Python

Python is one of the fastest-growing programming languages due to its ease of use. One of the draws of Python is that we can write the same programs we do with languages such as C, C++, and Java, but with fewer lines of code and simpler language and syntax. Another big draw of Python is that it is **extensible**, which means we can add capabilities and functionalities to it.

While not all functionalities are inherently built in, we can use libraries to add what we need. Those libraries are available to download and use. For example, if we wanted to work with data and data science, there are a few libraries we can download, such as **Pandas**, **NumPy**, **Matplotlib**, **SciPy**, **Scikit Learn**, and more. But before we get into those libraries, let's look at how the Python language works and learn its basics.

Python has some built-in reference functions. The following table shows the functions in alphabetical order:



abs()	all()	any()	ascii()
bin()	bool()	breakpoint()	bytearray()
bytes()	callable()	chr()	classmethod()
compile()	complex()	delattr()	dict()
dir()	divmod()	enumerate()	eval()
exec()	filter()	float()	format()
frozenset()	getattr()	globals()	hasattr()
hash()	help()	hex()	id()
input()	int()	isinstance()	issubclass()
iter()	len()	list()	locals()
map()	max()	memoryview()	min()
next()	object()	oct()	open()
ord()	pow()	print()	property()
range()	repr()	reversed()	round()
set()	setattr()	slice()	sorted()
staticmethod()	str()	sum()	super()
tuple()	type()	vars()	zip()
__import__()			

Table 8.1 – Python built-in functions

While we won't go over all the functions in this book, we will use some of them as we look at Python and its algorithms. Let's start with some of the mathematical functions listed here.

Mathematical built-in functions

In Python, some of the mathematical functions are already built in, such as the `abs()`, `eval()`, `max()`, `min()`, and `sum()` functions. These are not the only mathematical functions built in, but we'll take a closer look at these particular ones in order to understand how Python deals with them.

The `abs()` function will help us find the absolute value of a number, either an integer or a float. Take a look at the following code snippet:

ch8_absFunction.py

```
x = abs(-3.89)
print(x)
```

When we run this program, we'll get the absolute value of `-3.89`. Remember that the absolute value of a number is its distance from 0. Take a look at the output when we run this program:

```
3.89
```

Since the absolute value is always positive, when we run `abs(-3.89)`, we get `3.89`.

Another helpful mathematical function is the `eval()` function. We can define a variable in this function and then call for Python to evaluate an algebraic expression using that value. In the Python shell, we can first define the variable as follows:

```
>>> p = 2
```

Now that the variable is defined, we can call the `eval()` function with any expression. For example, take a look at the following input and output:

```
>>> eval('2 * p - 1')
```

As you can see, Python used the previously defined `p` value of `2`, and substituted then evaluated the expression to produce this output:

```
3
```

The Python program also works as a calculator, so you can perform mathematical operations as you normally would. Here are a few examples:

```
>>> 10-8
```

As you can see, Python knows to read the dash as a subtraction and produces the result of the mathematical expression:

```
2
```

In the next case, Python interpreted + as a mathematical symbol and provided the result of the sum expression:

```
>>> 4+5
```

The output is:

```
9
```

Notice the last expression, `10**5`. In Python, we can use two stars (`**`) to denote exponents:

```
>>> 10**5
```

The output is:

```
100000
```

Now, let's take a look at the `max()` function. This function can be used on an iterable list, but we can test it with just two numbers:

```
>>> max(12, 30)
```

You can clearly see what the output is:

```
30
```

Let's take another example:

```
>>> max(100, 10)
```

This is the output:

```
100
```

As you can see from the output, the function always chooses the largest item. You can add a third item and the function will still choose the maximum value. These built-in functions are smart in a way that they are coded to work with what is provided – two or three items, for example – without having to tell Python explicitly how many items we'll be introducing:

```
>>> max(230, 812, 109)
```

The output obtained is:

```
812
```

As you can see, we did not have to add anything to the built-in function to find the maximum of three values.

The `min()` function is the opposite; it chooses the minimum value. Take a look at the following minimum function:

```
>>> min(230, 812, 109)
```

This is the output:

```
109
```

As you can see, the function used the same list we used for the maximum function, but this time the output is `109`, which is the minimum value of that group of numbers.

There are other mathematical functions, such as `sum()`. If you're a Python beginner, it's recommended that you play around with those functions to learn how they work. These functions will be key to your algorithms as you design solutions for computational thinking problems. We will also use some of these functions as we look at other Python functionalities, such as dictionaries and arrays.

Working with dictionaries and lists

Before we get too deep into dictionaries and lists, it's important to note that Python does not contain arrays in the built-in functions. We can use lists and perform a lot of the traditional functions on lists that we'd use for arrays. However, for more robust functionalities with arrays, a library is needed, such as NumPy.

Python has four collection data types, which are as follows:

- **Lists**: Ordered and changeable; can have duplicates
- **Tuples**: Ordered and unchangeable; can have duplicates
- **Sets**: Unordered and unindexed; cannot have duplicates
- **Dictionaries**: Unordered, changeable, and indexed; cannot have duplicates

As noted, we won't be going into NumPy libraries or other data libraries just yet. For now, we're going to focus on dictionaries and lists.

Defining and using dictionaries

You may recall that we used dictionaries in *Chapter 3*, *Understanding Algorithms and Algorithmic Thinking*, when we created a menu of items. Dictionaries in Python are collections that have the following three characteristics:

- They are unordered.
- They are changeable.
- They are indexed by keys.

Dictionaries are organized as value pairs. For example, we can have a dictionary with value pairs of states and their capitals. Take a look at the following dictionary:

ch8_dictionary1.py

```
states = {
    'Ohio':'Columbus',
    'Delaware':'Dover',
    'Pennsylvania':'Harrisburg',
    'Vermont':'Montpelier'
    }
print(states)
```

If we were to print this dictionary without any conditions to the `print` statement, we would get the following output:

```
{'Ohio': 'Columbus', 'Delaware': 'Dover', 'Pennsylvania':
'Harrisburg', 'Vermont': 'Montpelier'}
```

As you can see, each value pair is printed at once. Just like we built the dictionary in this way, Python also has a built-in `dict()` function. The same dictionary can be built using that function, as follows:

ch8_dictionary2.py

```
states = dict([
    ('Ohio','Columbus'),
    ('Delaware','Dover'),
    ('Pennsylvania','Harrisburg'),
    ('Vermont','Montpelier')
])
print(states)
```

You can see that the dictionary is constructed very similarly in both examples, with some changes in syntax, such as the use of colons in the first instance, versus parentheses and commas in the second one. However, the `print` statement produces the same result.

Dictionaries are said to be key-value pairs because the first item is the key and the second, paired item is the value. So, for `Ohio` and `Columbus`, `Ohio` is the *key* while `Columbus` is the *value*. We can use the key to call any value. For example, I can call `states['Ohio']` and it should return `'Columbus'`. Take a look at the code:

```
>>> states['Ohio']
```

This is the output:

```
'Columbus'
```

We can do that for any key-value pair in the dictionary. But if we try to call a key that's not in the dictionary, such as `Alabama`, then we get the following error:

```
>>> states['Alabama']
```

This leads to the following error being displayed:

```
Traceback (most recent call last):
  File "<pyshell#23>", line 1, in <module>
    states['Alabama']
KeyError: 'Alabama'
```

Notice that this gives a `KeyError`, mentioned in the list of errors in *Chapter 7*, *Identifying Challenges within Solutions*, in the *Errors in logic* section. But let's say we do want to add `Alabama` and the capital, which is `Montgomery`. We can use the following code to do so:

```
>>> states['Alabama'] = 'Montgomery'
```

We can call the dictionary again after entering the following code:

```
>>> print(states)
```

This gives us the following output:

```
{'Ohio': 'Columbus', 'Delaware': 'Dover', 'Pennsylvania':
'Harrisburg', 'Vermont': 'Montpelier', 'Alabama': 'Montgomery'}
```

Notice that the dictionary added the `'Alabama':'Montgomery'` key-value pair at the end of the dictionary.

We can also delete key-value pairs using the `del` code, as shown:

```
>>> del states['Delaware']
```

Now, if we go ahead and print the `states` dictionary, `Delaware` is no longer on the list:

```
>>> print(states)
```

This is the output after deleting the state:

```
{'Ohio': 'Columbus', 'Pennsylvania': 'Harrisburg', 'Vermont':
'Montpelier', 'Alabama': 'Montgomery'}
```

Now, you can continue to add items or delete them without having to go into the main algorithm.

With dictionaries, you can also add multiple values to one key. Let's say I play three sports (I don't). I could list those in a dictionary and match them to one key value. Take a look at the following algorithm:

ch8_dictionary3.py

```
miscellaneous = {
    'sports' : ['tennis', 'bowling', 'golf'],
    'age' : '40',

    'home' : 'lake'
    }
```

We can print the full dictionary using `print(miscellaneous)`, as follows:

```
>>> print(miscellaneous)
```

This is how we get the output:

```
{'sports': ['tennis', 'bowling', 'golf'], 'age': '40', 'home':
'lake'}
```

As you can see, the printed dictionary includes all the values. If I want to print only the sports, then I can use the `miscellaneous['sports']` code:

```
>>> miscellaneous['sports']
```

This is the output:

```
['tennis', 'bowling', 'golf']
```

Notice that we did not use the `print()` function at all. We used the dictionary, which we called `miscellaneous`, and called the `'sports'` key. Note that we were able to get this result because we called the dictionary in the **IDLE** command window in Python. If you want to include this in your algorithm, you'll still need to use the `print()` function.

While we won't go into all the functionalities of dictionaries here, you can see how they can be helpful in creating algorithms that contain key-value pairs. Python allows us to add and alter the dictionary, call values, and more with the use of simple code without having to access the entire dictionary to do so.

Next, we'll take a look at lists.

Defining and using lists

Lists in Python are ordered and changeable. We can create lists for anything, such as types of animals, colors, fruit, numbers, or really whatever we want. Since we can have duplicate values, we could have a list for three apples that just says `apple, apple, apple`. For example, take a look at the list shown:

ch8_list1.py

```
fruits = ['apple','apple','apple']
print(fruits)
```

When we print this list, all the items are included. The output is as follows:

```
['apple', 'apple', 'apple']
```

As you can see, we have the same value three times. We would not be able to do that with a dictionary, which does not allow duplicate members.

Now, let's take a look at what we can do with lists. Let's start with the following list of animals:

ch8_list2.py

```
animals = ['dog', 'cat', 'bird', 'lion', 'tiger', 'elephant']
```

The first item in the list is `'dog'`. Lists have indexes, starting at 0. So, if we printed `animals[0]`, we'd get `dog`. We can check it here:

```
>>> print(animals[0])
```

This is the output with index 0:

```
dog
```

There are six items on the list, but the last index is `[5]`. So, to print `elephant`, we'd need to print using that index:

```
>>> print(animals[5])
```

```
elephant
```

You can also use negative numbers for indexes in lists. The `[-1]` index refers to the last index, so it represents `elephant`, the same as index `[5]`:

```
>>> print(animals[-1])
```

```
elephant
```

The `[-2]` index refers to the second-to-last index, so `tiger`, and so on.

We can also print multiple items from a list by specifying a range of indexes. Take a look at the following code:

```
>>> print(animals[1:4])
```

This is the output:

```
['cat', 'bird', 'lion']
```

As you can see, we printed the second item on the list, which corresponds to index `[1]`, and the next two items.

With lists, we can also add items, replace items, delete items, check for the length, and more. To add an item, we need to use the `append()` method. Let's add the `duck` item to our list:

```
>>> animals.append('duck')
>>> print(animals)
```

Here is the added item:

```
['dog', 'cat', 'bird', 'lion', 'tiger', 'elephant', 'duck']
```

Notice that our list now has `duck` at the end of the list. But let's say we want to remove `bird` and substitute it with `butterfly`:

First, we have to identify the index for `bird`. That index is `2`, as it is the third item on the list:

```
>>> animals[2] = 'butterfly'
>>> print(animals)
```

Here is the output, with `bird` replaced:

```
['dog', 'cat', 'butterfly', 'lion', 'tiger', 'elephant',
'duck']
```

The list now contains `butterfly`.

Removing an item is fairly straightforward, as we just use the `remove()` method and the item we want to remove:

```
>>> animals.remove('lion')
>>> print(animals)
```

As you can see, `lion` has been removed from the list:

```
['dog', 'cat', 'butterfly', 'tiger', 'elephant', 'duck']
```

We can also remove an item by index using the `pop()` method. Using index `1` removes the second item on the list. Refer to the following code:

```
>>> animals.pop(1)
```

```
'cat'
```

Then, we can try the following:

```
>>> print(animals)
```

```
['dog', 'butterfly', 'tiger', 'elephant', 'duck']
```

Notice that Python identifies the item at index `1` that was *popped* out of the list. When we print the list again, the item is no longer there. If we want to remove the last item, we do not have to specify an index. Refer to the following code:

```
>>> animals.pop()
```

```
'duck'
```

Then, we can try the following:

```
>>> print(animals)
```

```
['dog', 'butterfly', 'tiger', 'elephant']
```

As mentioned, when no index is specified, the last item on the list is *popped* and removed from the list.

There is yet another way to remove an item from the list, by using the `del` keyword:

```
>>> del animals[1]
>>> print(animals)
```

This is the output:

```
['dog', 'tiger', 'elephant']
```

Our list lost the second item on the list. We can also use the `del` keyword to delete the list entirely:

```
>>> del animals
>>> print(animals)
```

This error is the output here:

```
Traceback (most recent call last):
  File "<pyshell#49>", line 1, in <module>
    print(animals)
NameError: name 'animals' is not defined
```

As you can see, I can no longer print the list because it is not defined. Notice we received a `NameError` description of `name 'animals' is not defined`. This is one of the other errors mentioned in *Chapter 7*, *Identifying Challenges within Solutions*.

Important Note:
For the next few code examples, I ran my original code from the `ch8_list2.py` file again to get my original list.

We can clear an entire list without eliminating the actual list by using the `clear()` method, as follows.

```
>>> animals.clear()
>>> print(animals)
```

This is the output we get:

```
[]
```

In this case, the list now prints as an empty list without giving an error message. That's because the list is still there and defined, it's just empty.

Now, let's say I wanted the length of my list. We can find the length of the list by using `len()`. Again, I went back to the original list to run the following code:

```
>>> print(len(animals))
```

We get the following output:

```
6
```

Our original list contained six elements – that is, six animals.

Now, let's define another list that contains colors:

ch8_list3.py

```
animals = ['dog', 'cat', 'bird', 'lion', 'tiger', 'elephant']

colors = ['blue', 'red', 'yellow']

print(animals)
print(colors)
```

The output for this algorithm is two lists. Now, if we wanted to combine the two lists, we could do so using the `extend()` method.

Let's take a look at how we can append `colors` to the `animals` list:

```
>>> animals.extend(colors)
>>> print(animals)
```

This is the appended list:

```
['dog', 'cat', 'bird', 'lion', 'tiger', 'elephant', 'blue',
'red', 'yellow']
```

Our list now includes all our animals and all our colors. We could have also used the following method to extend `colors` with `animals`. The difference is that the colors would appear first on our list:

```
>>> colors.extend(animals)
>>> print(colors)
```

This is how the list now appears:

```
['blue', 'red', 'yellow', 'dog', 'cat', 'bird', 'lion',
'tiger', 'elephant']
```

We can also sort lists, which is helpful when we want to have an alphabetical list or if we want to sort a list of numbers. Take a look at the two lists in the following algorithm:

ch8_list4.py

```
animals = ['dog', 'cat', 'bird', 'lion', 'tiger', 'elephant']
numbers = [4, 1, 5, 8, 2, 4]
```

```
print(animals)
print(numbers)
```

This is how they appear unsorted:

```
['dog', 'cat', 'bird', 'lion', 'tiger', 'elephant']
[4, 1, 5, 8, 2, 4]
```

Let's sort both lists and see what happens:

```
>>> numbers.sort()
>>> print(numbers)
```

We will get this output:

```
[1, 2, 4, 4, 5, 8]
```

Then, we can try the following:

```
>>> animals.sort()
>>> print(animals)
```

We get this as the output:

```
['bird', 'cat', 'dog', 'elephant', 'lion', 'tiger']
```

As you can see, the numbers were sorted from smallest to largest, while the animals were sorted in alphabetical order.

Let's talk for a second about why this is so helpful. Imagine that you have a list of items displayed on your website that come from a Python list. They are all sorted and perfectly displayed. But let's now say you want to add more items. It's easier to add them in any order to your list using the methods we have discussed, then we can sort them so that they continue to be in alphabetical order.

Of course, these are not the only things we can do with lists. Python allows us to work with lists in many ways that are user-friendly and function similar to what arrays do in other programming languages. When we need to use them in other ways, we can search for libraries that contain those functionalities.

Both lists and dictionaries are important in the Python programming language. In this section, you saw that dictionaries use key-value pairs, while lists include values. Both can be edited using methods and functions built into the Python programming language. You'll see them again when we apply Python to more complex computational thinking problems in *Section 3*, *Data Processing, Analysis, and Applications Using Computational Thinking and Python*, of this book.

Now, we'll take a look at how we use variables and functions in Python.

Using variables and functions

In Python, we use variables to store a value. We can then use the value to perform operations, evaluate expressions, or use them in functions. Functions give sets of instructions for the algorithm to follow when they are called in an algorithm. Many functions include variables within them. So, let's first look at how we define and use variables, then take a look at Python functions.

Variables in Python

Python does not have a command for declaring variables. We can create variables by naming them and setting them equal to whatever value we'd like. Let's take a look at an algorithm that contains multiple variables:

ch8_variables.py

```
name = 'Marcus'
b = 10
country_1 = 'Greece'

print(name)
print(b)
print(country_1)
```

As you can see, we can use a letter, a longer name, and even include underscores in naming our variables. We cannot, however, start a variable name with a number. When we run this program, we get the following output:

```
Marcus
10
Greece
```

Each variable was printed without any problems. If we had used a number to begin any variable name, we would have gotten an error. However, if I had named the `country_1` variable as `_country`, that would be an acceptable variable name in Python.

Now, let's take a look at what we can do with variables.

Combining variables

One thing variables allow us to do is combine them in a `print` statement. For example, I can create a `print` statement that prints `Marcus Greece`. To do so, I can use the + character, as follows:

```
>>> print(name + ' ' + country_1)
```

Notice that between the two + characters, there's `' '`. This is done to add a space so that our `print` statement won't look like `MarcusGreece`. Now, let's combine `b` and `name` in a `print` statement:

```
>>> print(b + name)
```

This will give us an error message:

```
Traceback (most recent call last):
  File "<pyshell#70>", line 1, in <module>
    print(b + name)
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Notice that the error states `TypeError: unsupported operand types for +`. It also states that we have `'int'` and `'str'`, which means we are combining two different data types.

To be able to combine those two data types in a `print` statement, we can just convert `int`, which is our `b` variable, to `str`. Here's what that looks like:

```
>>> print(str(b) + ' ' + name)
```

This easily gives us the desired result:

```
10 Marcus
```

Now that we've made them both strings, we can combine them in the `print` function.

Sometimes, we'll want to create many variables at once. Python allows us to do that with one line of code:

ch8_variables2.py

```
a, b, c, d = 'John', 'Mike', 'Jayden', 'George'

print(a)
print(b)
print(c)
print(d)
```

When we run this algorithm, we get the following output:

```
John
Mike
Jayden
George
```

As you can see, when we print the `a` variable, the first value to the right of the equals sign is printed. The first line of the algorithm assigned four different values to four different variables.

Let's now look at functions, as we'll need them to further our variables conversation.

Working with functions

In *Chapter 7*, *Identifying Challenges within Solutions*, we wrote an algorithm that printed even numbers for any given range of numbers. We're going to revisit that by defining a function to do that work for us. Let's take a look at an algorithm for the even numbers problem:

ch8_evenNumbers.py

```
def evenNumbers(i, j):

    a = i - 1
    b = j + 1
    for number in range(a, b):
        if number % 2 == 0:
            print(number)
            a = a + 1
```

If we run this program in Python, there is no output; however, we can now run the function for any range of numbers. Notice that we have added an `a` and `b` variable within this function. That's so that the endpoints are included when the program runs.

Let's see what happens when we run the program for the ranges `(2, 10)` and `(12, 25)`:

```
>>> evenNumbers(2, 10)
```

The output is as follows:

```
2
4
6
8
10
```

Then, we can try the following:

```
>>> evenNumbers(12, 25)
```

The output is as follows:

```
12
14
16
18
20
22
24
```

As you can see, instead of having to go inside the algorithm to call the function, once we've run the function, we can just call it for any range in the Python shell. As before, if our range is too large, the shell will show a really long list of numbers. So, we can define another variable, this time a list, and append the values to that list within the function:

ch8_evenNumbers2.py

```
listEvens = []

def evenNumbers(i, j):
    a = i - 1
    b = j + 1
    for number in range(a, b):
        if number % 2 == 0:
            listEvens.append(number)
            a = a + 1
    print(listEvens)
```

Notice that we defined the list outside of the function. We can do it either way. It can live within the function or outside. The difference is that the list exists even if the function is not called; that is, once I run the algorithm, I can call the list, which will be empty, or call the function, which will use the list. If outside the function, it's a global variable. If inside, it only exists when the function is called.

Let's try to run that now for the range `(10, 50)`:

```
>>> evenNumbers(10, 50)
```

This is the output:

```
[10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38,
40, 42, 44, 46, 48, 50]
```

As you can see, the even numbers between and including `10` and `50` are now included in the list and printed.

Let's look at another example of a function that takes a `(name)` string argument:

ch8_nameFunction.py

```
def nameFunction(name):
    print('Hello ' + name)
```

Once we run this algorithm, we can call the function for any name. Take a look:

```
>>> nameFunction('Sofia')
```

We get this output:

```
Hello Sofia
```

Let's input another name:

```
>>> nameFunction('Katya')
```

We will see the following output:

```
Hello Katya
```

As you can see, once a function is defined in an algorithm and the algorithm is run, we can call that function, as needed.

Now, we did use an iteration in the previous example with even numbers, so it's time we started looking more closely at files, data, and iteration, so let's pause and take a look at that. As we get to iteration, we'll run into a few more function examples.

Learning about files, data, and iteration

In this section, we're going to take a look at how to handle files, data, and iteration with Python. This will give us information on how to use an algorithm to open, edit, and run already-existing Python files or new files. With iteration, we'll learn how to repeat lines of code based on some conditions, limiting how many times or under what conditions a section of the algorithm will run for. We'll start with files first.

Handling files in Python

In Python, the main function associated with files is the `open()` function. If you are opening a file in the same directory, you can use the following command line:

```
fOpen = open('filename.txt')
```

If the file is in another location, you need to find the path for the file and include that in the command, as follows:

```
fOpen = open('C:/Documents/filename.txt')
```

There are a few more things we can tell the program to do in addition to opening the file. They are listed as follows:

- `r`: Used to open the file for reading (this is the default, so it does not need to be included); creates a new file if the file does not exist.
- `w`: Used to open the file for writing; creates a new file if the file does not exist.
- `a`: Used to open a file to append; creates a new file if the file does not exist.
- `x`: Used to create a specified file; if the file already exists, returns an error.

There are two more things that can be used in addition to the methods listed. When used, they identify how a file needs to be handled – that is, as binary or text:

- `t`: Text (this is the default, so if not specified, it defaults to text)
- `b`: Binary (used for binary mode – for example, images)

Take a look at the following code. Since this code is a single line and specific to each path, it is not included in the repository:

```
fOpen = open('filename.txt', 'rt')
```

The preceding code is the same as the previous code, `fOpen = open('filename.txt')`. Since `r` and `t` are the defaults, not including them results in the same execution of the algorithm.

If we wanted to open a file for writing, we could use the `fOpen = open('filename.txt', 'w')` code. To close files in Python, we use the `close()` code.

Some additional methods for text files are included in the following list. This is not an exhaustive list, but contains some of the most commonly used methods:

- `read()`: Can be used to read lines of a file; using `read(3)` reads the first 3 data.
- `tell()`: Used to find the current position as a number of bytes.
- `seek()`: Moves the cursor to the original/initial position.
- `readline()`: Reads each individual line of a file.
- `detach()`: Used to separate the underlying binary buffer from `TextIOBase` and return it.
- `readable()`: This will return `True` if it can read the file stream.
- `fileno()`: Used to return an integer number for the file.

As you can see, you can use Python to manipulate and get information from text files. This can come in handy when adding lines of code to an existing text file, for example. Now, let's look at data in Python.

Data in Python

Before we get into data, let's clarify that we're not talking about data types, which we discussed in *Chapter 1*, *Fundamentals of Computer Science*. For this chapter, we're looking at data and ways to interact with it, mostly as lists.

Let's look at some of the things we can do with data.

First, grab the `ch8_survey.txt` file from the repository. You'll need it for the next algorithm.

Say you asked your friends to choose between the colors blue, red, and yellow for a group logo. The `ch8_survey.txt` file contains the results of those votes. Python allows us to manipulate that data. For example, the first line of the file says `Payton - Blue`. We can make that line print out `Payton voted for Blue` instead. Let's look at the algorithm:

ch8_surveyData.py

```
with open("ch8_survey.txt") as file:
    for line in file:
        line = line.strip()
        divide = line.split(" - ")
        name = divide[0]
        color = divide[1]
        print(name + " voted for " + color)
```

Let's break down our code to understand what is happening:

- The first line opens the survey file and keeps it open.
- The next line, `for line in file`, will iterate through each line in the file to perform the following commands.
- The algorithm then takes the information and splits it at the dash. The first part is defined as the name (`name = divide[0]`) and the second is defined as the color (`color = divide[1]`).
- Finally, the code prints out the line with the `voted for` text added.

Take a look at the following output:

```
Payton voted for Blue
Johnny voted for Red
Maxi voted for Blue
Jacky voted for Red
Alicia voted for Blue
Jackson voted for Yellow
Percy voted for Yellow
```

As you can see, you now have each line adjusted to remove the dashes and include the `voted for` phrase. But what if you wanted to count the votes? We can write an algorithm for that as well:

ch8_surveyData2.py

```
print("The votes for Blue are in.")

blues = 0

with open("ch8_survey.txt") as file:
    for line in file:
        line = line.strip()
        name, color = line.split(' - ')
        if color == "Blue":
            blues = blues + 1

print(blues)
```

As you can see, we are counting the votes by verifying each line and using the `if color == "Blue"` code. When we run this algorithm, we get the following output:

```
The votes for Blue are in.
3
```

As you can see, the algorithm prints out the initial `print()` command, then prints the counted votes for `Blue`.

We can also work with data to find things such as the mean, median, and mode, but we won't go over those now. If we need them in the future for one of our applied problems, we'll go ahead and use them. However, most of those data-specific problems will use libraries to simplify some of the algorithms and computations.

Now, let's talk a bit more about iteration, which we've been using but will need to define further to better understand its use.

Using iteration in algorithms

Before we go into iteration, let's define the term. **Iteration** means repetition. When we use iteration in algorithms, we are repeating steps. Think of a `for` loop , which we've used in problems before, such as the even numbers problem—the algorithm iterated through a range of numbers. It repeated for all the numbers in our range.

Let's take a look at another example:

ch8_colorLoop.py

```
colors = ['blue', 'green', 'red']

for color in colors:
    print(color)
```

The iteration in this algorithm is that it will repeat the `print` process for each of the colors in the original colors list.

This is the output:

```
blue
green
red
```

As you can see, each of the colors was printed.

We can also get input from a user, then iterate to perform some operation. For example, take a look at the following algorithm:

ch8_whileAlgorithm.py

```
ask = int(input("Please type a number less than 20. "))

while ask > 0:
    print(ask * 2)
    ask = ask - 1
```

As you can see, we defined `ask` as the input variable. Then, we printed double the number and reduced the number by `1` as long as `ask` was greater than `0`.

The output looks like this:

```
Please type a number less than 20. 8
16
14
12
10
8
6
4
2
```

The output shows the list created by doubling the initial number, which is *8* x *2* = *16*, then continues until the `ask` variable is no longer greater than `0`.

Next, let's take a look at how we can iterate through multiple lists. The algorithm uses two lists and then prints a single statement using information from both:

ch8_Iterations.py

```
jewelry = ['ring', 'watch', 'necklace', 'earrings',
'bracelets']

colors = ['gold', 'silver', 'blue', 'red', 'black']

for j, c in zip(jewelry, colors):
    print("Type of jewelry: %s in %s color. " %(j, c))
```

When we run the algorithm, we get the following output:

```
Type of jewelry: ring in gold color.
Type of jewelry: watch in silver color.
Type of jewelry: necklace in blue color.
Type of jewelry: earrings in red color.
Type of jewelry: bracelets in black color.
```

Take a look at the `print` statement: `print("Type of jewelry: %s in %s color." %(j, c))`. The `%s` symbols will be replaced by the `(j, c)` values, respectively. So, the first `%s` symbol gets the items from `j` and the second `%s` symbol gets the items from `c`, where `j` is the item from the jewelry list and `c` is the color from the colors list.

As you can see, we can iterate through lists in multiple ways. These are only a few examples to get comfortable with loops and how to use information in our algorithms. As we dive deeper into more complex problems, the algorithms will get more sophisticated, so we'll revisit many of the topics in this chapter and earlier chapters.

Before moving on to the next chapter, we need to look at **Object-Oriented Programming** (**OOP**).

Using object-oriented programming

OOP is a way to structure data into objects. The Python program is an object-oriented program in that it structures algorithms as objects in order to bundle them based on properties and behaviors. To learn about OOP, we will need to know how to do the following:

- Create classes
- Use classes to create objects
- Use class inheritance to model systems

In Python, we use classes to bundle data. To understand classes, let's create one:

```
>>> class Books:
        pass
```

We can then call the `Books()` class and get the location where the class is saved on our computer. Notice that my output will not match yours, as the class will be saved in different locations:

```
>>> Books()
```

This is my output:

```
<__main__.Books object at 0x000001DD27E09DD8>
```

Now that we've created the class, we can add book objects to the class:

```
>>> a = Books()
>>> b = Books()
```

Each of these instances is a distinct object in `Books()`. If we were to compare them, since they are distinct, `a == b` would return `False`.

Now, let's take a look at a class that creates an address book. By creating the class, we can add entries as needed:

ch8_addressBook.py

```
class Entry:

    def __init__(self, firstName, lastName, phone):
        self.firstName = firstName
        self.lastName = lastName
        self.phone = phone
```

In this algorithm, we've created an `Entry` class that stands for items in an address book. Once we run the algorithm, we can add entries to the address book and call information on them. For example, take a look at the following code:

```
>>> Johnny = Entry('John', 'Smith', '201-444-5555')
```

This code enters `Johnny` into the address book. Now, we can call Johnny's first name, last name, and phone number separately, as follows:

```
>>> Johnny.firstName
```

This is my output:

```
'John'
```

We can call the last name:

```
>>> Johnny.lastName
```

This is the obtained output:

```
'Smith'
```

We can also call for phone number:

```
>>> Johnny.phone
```

We can see the output for this:

```
'201-444-5555'
```

We can add as many entries as we'd like, then call them as needed:

```
>>> max = Entry('Max', 'Minnow', '555-555-5555')
>>> emma = Entry('Emma', 'Dixon', '211-999-9999')
>>> emma.phone
```

This is our output:

```
'211-999-9999'
```

After adding the entries, we can call a last name:

```
>>> max.lastName
```

We get this output:

```
'Minnow'
```

As you can see, we added two new entries, then called Emma's phone number and Max's last name.

Once we have classes, we can have one class that inherits methods and attributes for other classes. Because the classes are inheriting the attributes and methods, the original class is called the **parent class**, while the new class is called a **child class**.

Going back to our address book, we can create a child class by passing the class to the new one. I know, it sounds confusing, but take a look at the algorithm:

```
>>> class Job(Entry):
      pass
```

We can now add more items using that child class as well. See the following example:

```
>>> engineer = Job('Justin', 'Jackson', '444-444-4444')
```

We can do more with classes, but let's try to pull this chapter together by solving a problem and designing an algorithm that uses at least some of the components learned about.

Problem 1 - Creating a book library

Let's say you have a lot of books and want to create an algorithm that stores information about the books. You want to record each book's title, author, publication date, and number of pages. Create an algorithm for this scenario.

First, let's think about the problem:

- The number of books I own changes constantly, so I'd want to create something that I can add information to, as needed.
- I also want to be able to remove books that I no longer own.

While we could use a library, the best solution for this particular problem would be a class. Let's start by creating a class called `Books`:

ch8_BookLibrary.py

```
class Books:

    def __init__(self, title, author, pubDate, pages):
        self.title = title
        self.author = author
        self.pubDate = pubDate
        self.pages = pages

book1 = Books('The Fundamentals of Fashion Design', 'Sorger &
Udale', '2006', '176')

book2 = Books('Projekt 1065: A Novel of World War II', 'Gratz',
'2016', '309')
```

As you can see, we have our class defined. The class has four arguments: `title`, `author`, `pubDate`, and `pages`. After the definition, two books have been added. When we run this program, nothing happens, really, but we can then call information on either book:

```
>>> book1.title
```

We will get this output:

```
'The Fundamentals of Fashion Design'
```

We will call the publishing date of `book2`:

```
>>> book2.pubDate
```

We obtain this output:

```
'2016'
```

You can see that I can now call any of the elements saved for each of the books after I've run the algorithm.

Now, take a look at how a third book can be added within the Python shell, as follows:

```
>>> book3 = Books('peanut butter dogs', 'Murray', '2017',
'160')
```

Since the book has been added, we can call information about this book as well:

```
>>> book3.title
```

We get this output:

```
'peanut butter dogs'
```

We can call the total pages of `book3`:

```
>>> book3.pages
```

We will get the following output:

```
'160'
```

As you can see, we can add books to our class from within the algorithm, or after running the algorithm in the Python shell. Now, let's take a look at another problem.

Problem 2 - Organizing information

We've been asked to create an algorithm that takes three numbers as input and provides the sum of the numbers. There are multiple ways we can do this, but let's look at using the `eval()` function:

ch8_Sums.py

```
a = int(input("Provide the first number to be added. "))
b = int(input("Please provide the second number to be added.
"))
c = int(input("Provide the last number to be added. "))

print(eval('a + b + c'))
```

Notice that we defined each of the input variables as an `int` type. This is defined so that the evaluation is done correctly.

Here is the output for our algorithm:

```
Provide the first number to be added. 1
Please provide the second number to be added. 2
Provide the last number to be added. 3
6
```

If we had forgotten to add the type for each of the numbers, the function would have evaluated that as `123` instead because it just adds each string to the next one. So, if our input had been `John`, `Mary`, `Jack`, our output would have been `JohnMaryJack`.

We previously didn't go into the `sum()` function. Let's take a look at using that function instead:

ch8_Sums2.py

```
a = int(input("Provide the first number to be added. "))
b = int(input("Please provide the second number to be added.
"))
c = int(input("Provide the last number to be added. "))

numbers = []

```

```
numbers.append(a)
numbers.append(b)
numbers.append(c)

print(sum(numbers))
```

Using `sum` in this case requires that we add our inputs to a list, as `sum()` works with iterables, such as lists. Although this solution has more code, the output is exactly the same as our `eval()` function, as you can see here:

```
Provide the first number to be added. 1
Please provide the second number to be added. 2
Provide the last number to be added. 3
6
```

As you can see, we get the same answer as before using a different Python function. Let's look at one more problem before we move on to the next chapter.

Problem 3 - Loops and math

For this problem, we have to create an algorithm that prints out the squares of all the numbers given a range of two numbers. Remember, we can print each individually, but if the range is large, it's best we have a list instead. We'll also need to iterate in the range, and we have to add 1 to the maximum if we want to include the endpoints.

Take a look at the following algorithm:

ch8_SquareLoops.py

```
print("This program will print the squares of the numbers in a given range of numbers.")

a = int(input("What is the minimum of your range? "))
b = int(input("What is the maximum of your range? "))

Numbers = []
b = b + 1
for i in range(a, b):
    j = i**2
    Numbers.append(j)
    i = i + 1

print(Numbers)
```

Notice that we added a `j` variable to our algorithm. We didn't use `i = i**2` because that would change the value of `i`, which would affect our iteration in the algorithm. By using `j`, we can then use `i` to iterate through the range given. Let's look at our output:

```
This program will print the squares of the numbers in a given
range of numbers.
What is the minimum of your range? 4
What is the maximum of your range? 14
[16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196]
```

This algorithm prints out our list of squares for the range provided. It also has an initial `print` statement that explains what the code will do.

Now that we've looked at a few examples, let's look at parent classes and child classes and how inheritance works.

Using inheritance

In Python, we can pass methods and properties from one class to another using inheritance. The parent class has the methods and properties that will be inherited. The child class inherits from the parent class. The parent class is an object and the child class is an object, so we are defining the properties of classes.

Let's see how that works using a science example. We're going to create a class called `mammals`. Not all mammals are viviparous. A mammal is viviparous when it gives birth to live young. A mammal that is not viviparous is the platypus. The platypus lays eggs instead of giving birth to live young. If we were writing an algorithm for this, we would want the animals to inherit the characteristics of the parent – in this case, `mammals`. Let's take a look at the following snippet of code:

ch8_mammals.py

```
class mammals:
     def description(self):
       print("Mammals are vertebrate animals.")

     def viviparous(self):
        print("Mammals are viviparous, but some are not.")

class monkey(mammals):
     def viviparous(self):
       print("Monkeys are viviparous.")

class platypus(mammals):
     def viviparous(self):
       print("The platypus is not viviparous. It's an egg-
laying mammal.")

obj_mammals = mammals()
obj_monkey = monkey()
obj_platypus = platypus()

obj_mammals.description()
obj_mammals.viviparous()

obj_monkey.description()
obj_monkey.viviparous()

obj_platypus.description()
obj_platypus.viviparous()
```

From the preceding code, notice that the `mammals()` class uses a description and then information about mammals and viviparity. `monkey` is defined using the same description as the `mammals` class, but then includes a different statement for viviparity. The same thing happens with `platypus`. The `monkey` and `platypus` classes are child classes of the `mammals` class.

The three classes, the parent and the two children, are then simplified into a variable so that they can be used by calling that variable. Finally, the algorithm prints the description and viviparity statements for the parent and the two children. Let's take a look at that output:

```
Mammals are vertebrate animals.
Mammals are viviparous, but some are not.
Mammals are vertebrate animals.
Monkeys are viviparous.
```

```
Mammals are vertebrate animals.
The platypus is not viviparous. It's an egg-laying mammal.
```

As you can see, all three classes used the same description. That's because we didn't make any changes to the description for the children. When we defined the classes, we only changed the parts we wanted to be different from the parent class. Everything else is inherited from the parent class.

Parent classes and children are widely used in Python for multiple purposes. For example, in gaming, you may have enemies that all have some of the same characteristics. Rather than defining each enemy with all the characteristics separately, we can create a parent class that includes all the common characteristics, then change the individual characteristics of all the enemies as child classes. This is just one of the ways in which inheritance can be used. There are many other uses and it helps us save time and avoid errors by only defining them with the parent classes.

Now that we've had some experience with classes and learned about OOP, let's wrap up the chapter.

Summary

In this chapter, we discussed the basics of Python programming. We looked at some of the built-in functions, worked with dictionaries and lists, used variables and functions, learned about files, data, and iteration, and learned about classes and OOP.

As we mentioned in this chapter and when solving previous problems, Python provides multiple ways for us to solve the same problems. One example of that is provided in the *Problem 2 - Organizing information* section of this chapter, where we used the `eval()` and `sum()` functions in two different algorithms to produce the same result. As we continue to learn about Python and how to write our algorithms, choosing which functions, variables, arguments, and more to use will start to become second nature. Some of the more challenging concepts and content in this chapter have to do with data analysis, such as the survey we used when introducing data in Python in the *Data in Python* section of this chapter, and classes. It's important to get some practice in those areas so that you can get a better understanding of those topics.

After this chapter, you can now work with built-in functions, distinguish between lists, dictionaries, and classes, and solve problems that combine multiple different concepts, such as variables, iterations, and lists. In the next chapter, we will be looking at input and output in more depth to help us when designing algorithms.

9
Understanding Input and Output to Design a Solution Algorithm

In this chapter, we will take a deeper look at problems in order to identify the input and output necessary to design an algorithm for the problems. We will use concepts we learned in *Chapter 8*, *Introduction to Python*, where we discussed object-oriented programming, dictionaries, lists, and more. As you practice getting input and using it in your algorithm, you'll be able to see that the output of an algorithm is dependent on the input information.

In this chapter, we will cover the following topics:

- Defining input and output
- Understanding input and output in computational thinking

In this chapter, we will focus on understanding different types of input and how output is used in programming using the **Python** programming language. To be able to get a better understanding of these topics, we have to first look at their definitions.

Technical requirements

For this chapter, you will need to have the latest Python version installed.

You can find the source code used in this chapter here: `https://github.com/PacktPublishing/Applied-Computational-Thinking-with-Python/tree/master/Chapter09`

Defining input and output

We will begin this section by studying input and its definition, which, in turn, is used to provide a result, or output.

Input is what is put into a system or machine. In computers, we have input devices that the computer then interprets to provide a result, or **output**. As we look at the types of inputs and how we use input in algorithms, you'll see the output we get from those algorithms. Let's take a look at some examples of input devices:

- **Keyboard**: When we use a keyboard, the computer interprets the keys and uses that interpretation in various programs, such as document writing and search bars.
- **Mouse**: The mouse is used to navigate our physical screens, helping us click through content and scroll through pages.
- **Joystick**: A joystick can be used to play computer games.
- **Microphone**: In more modern machines, microphones are not only used to communicate through phone and video apps, but also to take verbal commands for **Artificial Intelligence** (**AI**) assistants such as **Cortana** and **Siri**, as well as voice-to-text commands.

All of the aforementioned inputs are programmed into our computers in order to be useable. In Python, we write algorithms that use inputs such as these. **User** input is when a program asks the user for information that will then be used to produce the output. Keep in mind that input and output can happen at any time within the algorithm. We can also provide output feedback that requires additional input from the user, and so on. Let's take a look at how we can get user input from Python.

Python has one main prompt for users, the `input()` function. This function is used to take user input, and then it automatically converts the input into the user expected output.

We have used these main prompts before in multiple programs, such as the store example in *Chapter 5*, *Exploring Problem Analysis*. Let's take a look at what some input commands look like in algorithms. Take a look at the following code snippet:

ch9_input1.py

```
name = input("What is your name? ")
print("Nice to meet you " + name + ".")
```

The snippet asks the user for their name, and then prints a statement using the information from the input. The result of this code is as follows:

```
What is your name? Mikayla
Nice to meet you Mikayla.
```

Now, we could just ask for input without using any prompting questions, but the user will not know what is being asked or how it will be used. Take a look at the snippet without the prompts, shown as follows:

ch9_input2.py

```
name = input()
print("Nice to meet you " + name + ".")
```

When we run the preceding program, nothing prints out in our shell. However, I knew there was something I needed to type, so take a look at what happened when I just typed a letter:

```
d
Nice to meet you d.
```

As you can see, the user wouldn't have known what to do because the window doesn't ask anything; it just gives a blank space and the user has to assume that's where you type the input. But that can cause a lot of confusion, since there's nothing telling the user whether it needs input or what kind of thing they have to enter.

We can alleviate some of that confusion by using `print` statements, for example, by providing the user with information about what input we'll be asking for. We can also include a statement as part of the `input()` function. For example, we could use a line of code such as `name = input('What is your name?')`, which would show the statement in quotations first so that the user knows what we are asking. Either way, providing a statement so that the user knows what is needed is really important when we design our algorithms.

As you saw in the preceding example, the `input()` function took a string, and then used the string in a `print()` statement. Let's look at what a list would look like using the `input` command:

ch9_input3.py

```
#Create the list
names = []

#Ask user how many names will be added
name = int(input("How many names will be in the list? "))

#Iterate to add each name to the list
for i in range(0, name):
    people = str(input())

    names.append(people)

print(names)
```

In the preceding snippet, we first create the list, which is not shown to the user. The user is then asked how many names will be added to the list. After that, we iterate through the list so that the user can input each name separately. Finally, the names are appended and the list printed. Take a look at the output for this algorithm:

```
How many names will be in the list? 4
Manny
Lisa
John
Katya
['Manny', 'Lisa', 'John', 'Katya']
```

Notice that the names don't have a prompt, so the algorithm assumes the user will know what to do after inputting the value 4. That can be alleviated with a simple addition within the iteration. Take a look at the following code snippet:

ch9_input4.py

```
#Create the list
names = []

#Ask user how many names will be added
name = int(input("How many names will be in the list? "))

#Iterate to add each name to the list
for i in range(0, name):
    people = input("Type the next name on the list. ")

    names.append(people)

print(names)
```

As you can see, each name will now have a prompt that asks to type the next name, which looks like the following output when the algorithm is run:

```
How many names will be in the list? 4
Type the next name on the list. Milo
Type the next name on the list. Sonya
Type the next name on the list. Gabriel
Type the next name on the list. Maxine
['Milo', 'Sonya', 'Gabriel', 'Maxine']
```

The preceding output shows the completed list and the prompts for each of the names that were added to that list. Simple additions for those prompts within the algorithm can alleviate confusion on the part of users when entering the inputs.

As you can see from the previous snippets in this section, the inputs were read as integers, strings, and lists. Python automatically converted the information so it could be used by the algorithm.

In Python, there's also a way to define multiple variables with separate inputs from one line of code. Take a look at the following code snippet:

ch9_input5.py

```
name1, name2 = input("Enter First Name: "), input("Enter Last 
Name: ")

print(name1 + " " + name2)
```

As you can see in the preceding code, the quotations (`" "`) in the middle of the `print()` command are used to separate the inputs. Take a look at the output for this algorithm:

```
Enter First Name: John
Enter Last Name: Doe
John Doe
```

As you can see, the program asks for the first and last names, which were called in the first line of the algorithm. This is by no means the only way that we can use for getting inputs from users, nor the only input we'll use.

In this section, we looked at some of the ways in which we can define input within an algorithm. We also learned how we can get input from a user to use later in the algorithm. Now let's move to the next section, where we'll look at input and output within some computational thinking problems.

Understanding input and output in computational thinking

In order to better understand input and output, we will take a look at computational thinking problems in this section. As we go through the design process of the algorithms, we will focus on identifying the inputs needed and what output we require from our algorithms. Let's take a look at our first problem.

Problem 1 – Building a Caesar cipher

A **Caesar cipher** is a system of cryptography used to code messages. Cryptography is used to secure information so that only the intended users can read the messages. The Caesar cipher uses a shift in the letters to encode the messages. For example, the letter *a* shifted 3 places would be *d*. To build an algorithm that does this for us, we'll need a few things on hand:

- A message that will be coded
- How much we'll shift each letter
- A printed coded message

Let's think about what the aforementioned points mean. The message will need to be an input from the user. The shift will also be an input, as we don't want the program to always use the same code. Otherwise, it will be easy to figure out the original message once you know the shift. The printed coded message is our output. Here are some of the steps to help create the algorithm:

1. We first need to input the message.
2. Then we need to define the input shift.
3. After that, we need to iterate through each letter.
4. After iteration, we adjust each letter with the shift we defined.
5. Finally, we print out the new, coded message.

We will exemplify the preceding steps with the help of the following cipher algorithm snippet:

ch9_problem1.py

```
#Print initial message for the user
print("This program will take your message and encode it.")

#Ask for the message
msg = input("What message would you like to code? ")

#Ask for shift
shift = int(input("How many places will you shift your message? 
"))
msgCipher = ""
```

```
#Iterate through the letters, adjusting for shift
for letter in msg:
  k = ord(letter)
  if 48 <= k <= 57:
    newk = (k - 48 + shift)%10 + 48
  elif 65 <= k <= 90:
    newk = (k - 65 + shift)%26 + 65
  elif 97 <= k <=122:
    newk = (k - 97 + shift)%26 + 97
  else:
    newk = k
  msgCipher += chr(newk)
print("Your coded message is below.")
print(msgCipher)
```

Notice that in the iteration, we're doing some math to figure out the value for each letter in the alphabet. We use some conditional statements to define what each letter's value is after using the user-defined shift value.

Take a look at the output that results when we run this algorithm:

```
This program will take your message and encode it.
What message would you like to code? Code this message
How many places will you shift your message? 2
Your coded message is below.
Eqfg vjku oguucig
```

As you can see from the preceding output, notice the first word in the *message* and the *coded message* – `Code`:

- The `C`, moved two letters, is now an `E`.
- The `o` is now `q`, and so on.

For each letter in the message, the letters have shifted two places. The output is our `print()` statement plus the coded message, which is given by the line `print(msgCipher)`. Our output includes both statements for clarity. *Is the first message necessary?* No. However, in some algorithms, it's best to have some lines that allow the user to know what is happening with the algorithm.

Let's take a look at another problem.

Problem 2 – Finding maximums

You are asked to create a program that finds the maximum value from a list of numbers. The list of numbers is provided by the user. So, create an algorithm for this problem.

First, we need to identify the inputs and output for this particular problem. They are listed as follows:

1. Number of items in the list (input)
2. Numbers in the list (input)
3. Maximum value in the list (output)

Recall from earlier in this chapter, in the *Defining input and output* section, that we can define an empty list, and then ask the user to let the program know how many items will be entered into the list. The following program exemplifies the same:

ch9_problem2.py

```
#Define the list name
maxList = []
#Ask user how many numbers will be entered
quant = int(input("How many data points are you entering? "))
#Iterate, append points, and find maximum
for i in range(0, quant):
    dataPoint = int(input("Enter number: "))
    maxList.append(dataPoint)
#Print maximum value
print("The maximum value is " + str(max(maxList)) + ".")
```

Notice from the preceding code that we used the `max()` function to find the maximum value within the list. In addition, we had to add `int()` types in order to have the algorithm correctly identify the values as numbers. The code goes through each number from `0` to the number of data points we received from the user input, which we defined as the `quant` variable. As the algorithm iterates through the numbers, it compares them and finds the maximum value.

Let's take a look at what the output looks like:

```
How many data points are you entering? 6
Enter number: 1
Enter number: 2
Enter number: 8
Enter number: 4
Enter number: 5
Enter number: 7
The maximum value is 8.
```

As you can see from the preceding output, the user stated that `6` numbers would be entered. The algorithm then prompted the user for each of the values. Finally, the maximum value was identified.

Let's now look at how we can build a guessing game.

Problem 3 – Building a guessing game

You've been asked to build a game where a user gets as many chances as needed to identify the digits of a four-digit number in the correct order. When a user inputs a guess, the algorithm will state whether any numbers are correct.

The algorithm will identify whether the numbers are in the correct location, but will not identify which of the numbers are correct or which are in the correct location. If the player guesses the correct number, the game ends. You may find this game similar to a popular game called **Mastermind**, where a player sets four color pins into the game and a second player guesses.

In this case, we're using numbers rather than colors, and the computer program will help us identify whether we have any numbers correct and whether they are in the correct location.

To build this game, let's look at the inputs and outputs that we'll need:

- A randomly generated 4-digit number (input)
- The user's guesses (input)
- Feedback about numbers and locations from the algorithm (output)
- A message finalizing the game after the number is guessed (output)

When we generate the number, we'll need to find a way to compare the digits of that number to the digits that the user inputs. Note that we need to import the `random` library for this particular algorithm so we can generate the random four-digit number. Let's take a look at the code snippet:

ch9_problem3.py

```
import random as rand

number = rand.randint(1000,10000)

guess = int(input("What's your first guess? "))

#Algorithm checks if number is correct.
if (guess == number):
    print("That's right! You win!")
else:
        i = 0
```

As you can see, this is one instance where we'll need input from the user. However, this is only the first guess, so we'll need to get more input from the user and provide some output that provides feedback. Take a look at the following code snippet from the same algorithm:

```
#Condition so that user keeps guessing until they win.
        while (guess != number):
            i = i + 1
            #Remember you can also write as i += 1
            j = 0
            guess = str(guess)
            number = str(number)
            #Check which numbers are correct and mark incorrect 
with 'N'
            guessY = ['N']*4
            #Make sure you check each digit, so run loop 4 
times
            for q in range(0, 4):
                    if (guess[q] == number[q]):
                            j += 1
                    guessY[q] = guess[q]
                else:
                    continue
            #If only some digits are correct, run next
condition
            if (j < 4) and (j != 0):
                print("You have " + str(j) + " digit(s)
right.")
                print("These numbers were correct.")
                for m in guessY:
                    print(m, end = " ")
                #Ask for next input
                guess = int(input("What is your next guess?
"))
```

Notice that we need to ask for each guess after a failed attempt. The algorithm needs to know what the next guess is in order to continue running if the user hasn't identified the correct four-digit number. That's why the guess input value happens in multiple places within the algorithm.

Once the user guesses correctly, the program needs to print a final statement, and in this case, we'll let the user know how many attempts it took to guess:

```
            #If only some digits are correct, run next
condition
            if (j < 4) and (j != 0):
                print("You have " + str(j) + " digit(s)
right.")
                print("These numbers were correct.")
                for m in guessY:
                    print(m, end = " ")

                #Ask for next input
                guess = int(input("What is your next guess?
"))

            #No digits correct
            elif (j == 0):
                print("None of these digits are correct.")
```

```
                    guess = int(input("What is your next guess? "))

        if guess == number:
            print("It took you " + str(i)+ " tries to win!")
```

As you can see, this algorithm has input at multiple places based on conditions met, as well as output. The outputs include `print` statements and feedback on correct or incorrect digits. Let's take a look at what the output looks like when the program is executed:

```
What's your first guess? 1111
None of these digits are correct.
What is your next guess? 2222
None of these digits are correct.
What is your next guess? 3333
You have 1 digit(s) right.
These numbers were correct.
3 N N N What is your next guess? 3444
You have 3 digit(s) right.
These numbers were correct.
3 4 N 4 What is your next guess? 3454
You have 3 digit(s) right.
These numbers were correct.
3 4 N 4 What is your next guess? 3464
You won in 6 attempts.
```

Notice that after the third attempt, we had one correct digit, which was denoted as `3 N N N`. That means that the first digit is 3. We had to then guess the rest of the numbers. Providing feedback within the program, or output, allows the user to continue this game.

As you can see, understanding input and output allows us to create programs that can provide feedback, iterate based on conditions, and much more.

In this section, we learned how to use input and output when solving problems. We were provided with scenarios that allowed us to explore some of the types of inputs, such as user-defined inputs and algorithm-defined inputs. We also learned how to write algorithms that provide clear output for the users.

Summary

In this chapter, we discussed input and output in Python algorithms. Input is how we get information for the algorithm to respond to. For example, a user can provide input to an algorithm that has prompts for that information, as we saw in the sample problems. Python and other programming languages can also get input from a mouse, scanners, cameras, and other devices that the program interacts with.

In order to write our algorithms, it is important to identify what inputs we need, when we need them in the design, and what we need as output from the program. In *Chapter 3*, *Understanding Algorithms and Algorithmic Thinking*, we used user input to find the cost of lunches. We had created a dictionary to help us find that information. Similarly, after reading this chapter, you now have the skills required to build algorithms to solve some additional problems, such as our digit guessing game and our maximum finder.

Identifying input and output is critical before we design algorithms, as the conditions and decisions we make in our algorithms depend on what is entered as input and what we need the program to provide as output.

In the next chapter, we will discuss control flow, which is critical in understanding how an algorithm is read and how the instructions are performed. We will also look more closely at functions while exploring control flow in algorithms.

10
Control Flow

In this chapter, we will take a deeper look at problems and identify the input and output necessary to design an algorithm for our problems. Throughout this chapter, you will learn how algorithms are read and the order in which instructions are carried out. You will also learn how to use functions and loops to manipulate the control flow in your algorithms.

We will cover the following topics in this chapter:

- Defining control flow and its tools
- Using if, for, and range() and other control flow tools
- Using loops and conditionals
- Revisiting functions

By the end of the chapter, we will learn how control flow is defined, how we use the `if`, `for`, and `range()` functionalities when designing algorithms in computational thinking, and how to incorporate these functionalities into function definitions in our algorithms. Let's take a look at what control flow is first.

Technical requirements

You will need the latest version of Python to run the codes in this chapter. You will find the full source code used in this chapter here: `https://github.com/PacktPublishing/Applied-Computational-Thinking-with-Python/tree/master/Chapter10`

Defining control flow and its tools

Simply defined, **control flow** is the order in which an algorithm reads and executes instructions, functions, and statements. Control flow means the computer will make a decision. Think about it this way: when I go outside my building, I'll either use an umbrella or I won't use an umbrella. That depends on whether or not it is raining (and whether or not I remembered my umbrella, I guess). Depending on that condition, we make a decision. That process is the control flow process in algorithm design.

Let's look at some of the control flow statements available in the Python programming language:

- `while`
- `if`
- `for`
- `range()`
- `break`
- `continue`
- list comprehensions
- The `pass` statement
- The `try` statement

These tools allow the algorithm to do things such as run so long as a certain condition or set of conditions is met, stop or break when a condition happens, continue for a range of values only, and so on. Let's take a closer look at some of these tools.

Using if, for, and range() and other control flow statements

Let's start with `if` statements, which we first discussed in *Chapter 4*, *Understanding Logical Reasoning*. These are probably the most commonly used and known statements in algorithm design. You may recall learning about **conditional statements** in geometry as you studied reasoning and proofs. In those classes, you would write statements in *if-then* format. Take the following statement:

When it rains, I wear a raincoat.

This is not a conditional statement, at least not yet. If we were going to write it as a conditional sentence, then we'd have to have the *if-then* format, much like this sentence. Take a look at the converted statement that follows:

If it rains, then I wear a raincoat.

As you can see, we use conditions as part of our everyday lives. We just don't point them out.

When writing algorithms, we have to explicitly state what we need the algorithm to do, so we have to explicitly state these statements. In programming, we have to state each condition. Additionally, if we have a series of conditions that need to be met, we sometimes need to *nest* the statements. This is best explained with some examples. Let's start with how we use nested statements.

Using nested if statements

Even in our everyday lives, some conditions depend on others. For example, if it's Monday, we have to go to work. If we have to go to work and it's raining, we may need an umbrella. But if we're not going to work (and assuming we're staying home), we won't need to check whether we need an umbrella. That's the same with nested statements. We use them to check one condition, then another, which is nested.

Let's say we are playing a dice game. You roll a die and get points as follows:

- 2, 4, or 6 = 10 points
- 1 or 3 = 5 points
- 5 = 0 points and removes all previous points

Each round, a player has to roll twice. Let's look at a flowchart with the first roll shown, as follows:

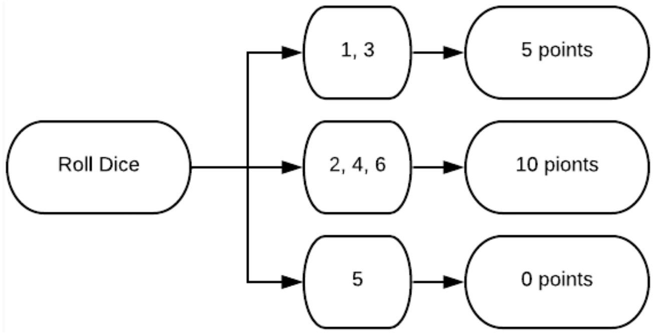


Figure 10.1 – Flowchart of first roll scores

As you can see from the flowchart, the score depends on the number rolled. Now, let's say you had rolled a **1**. Then, let's take a look at what your second roll could look like in the following diagram:

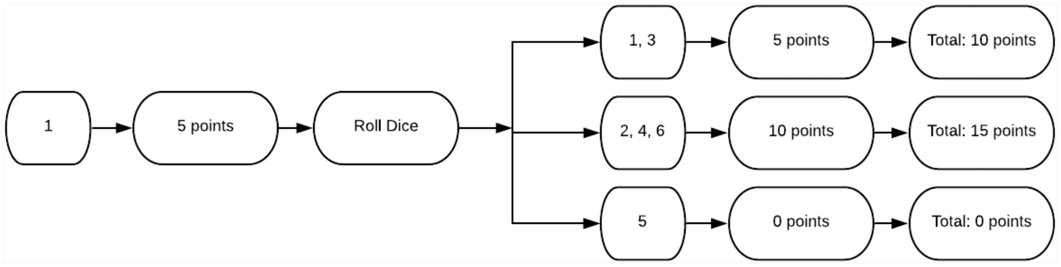


Figure 10.2 – Flowchart of the second roll after rolling 1

As you can see, we have three possible total points by the player. To translate this into an algorithm, we'll need to do a few things:

1. Define the roll.
2. Determine the score based on the number rolled.
3. Roll again.
4. Determine the final score based on the second roll.

Let's first define that roll. We'll need to virtually roll the dice. That means that we'll need the program to choose a number between `1` and `6`. Let's take a look at how we code that. Keep in mind that this is just a snippet of the larger code contained in this file. We'll use the second `ready` (`ready2`) in a subsequent snippet:

ch10_rollDice.py

```
import random as rand
print("Let's play a game. ")
print("You get 5 points for rolling 1 or 3. You get 5 points
for rolling 2, 4, or 6.")
print("You lose all points if you roll a 5 in either round.")
ready = input("Are you ready to begin? Type r to roll. ")
score = 0

if ready == 'r':
    roll = rand.randint(1, 6)
    print('You rolled a ' + str(roll) + '.')

    if (roll == 1) or (roll == 3):
        score = 5
    elif (roll == 5):
        score = 0
    else:
        score = 10
    ready2 = input('Your round 1 score is ' + str(score) + '.
```

Notice in the preceding snippet that we had to import the `random` library for this particular algorithm. Importing it as `rand` instead of `random` allows us to truncate a bit of the code so that we type less. So, instead of `random.randint()`, we use `rand.randint()`. We need the `random` library because we want to have access to the *random integer function* so that the algorithm can choose a number between `1` and `6` (including both endpoints). Let's take a look at the output as it stands now:

```
Let's play a game.
You get 5 points for rolling 1 or 3. You get 5 points for
rolling 2, 4, or 6.
You lose all points if you roll a 5 in either round.
Are you ready to begin? Type r to roll. r
```

```
You rolled a 3.
Your round 1 score is 5.
```

Notice from the preceding output that the instructions were provided first, then once the player chose to roll by pressing the `r` key, the game chose a random integer and displayed it. It also displayed the current score after round `1`. Now, let's take a look at the next part of the algorithm, which does the second roll. Notice that the indentations follow the previous code snippet. This snippet is contained in the previous larger file as well:

```
ready2 = input('Your round 1 score is ' + str(score) + '. Type 
r to roll again. ')
    roll2 = rand.randint(1, 6)
    print('You rolled a ' + str(roll2) + '.')
    if (roll2 == 1) or (roll2 == 3):
        score += 5
    elif (roll2 == 5):
        score = 0
    else:
        score += 10
    print('Your final score is ' + str(score) + '.')
```

When we run the new code, it shows the *first-round score*, both rolls, and a final score for the game. Remember, if we roll a `5` in the second round, we lose all our points:

```
Let's play a game.
You get 5 points for rolling 1 or 3. You get 5 points for 
rolling 2, 4, or 6.
You lose all points if you roll a 5 in either round.
Are you ready to begin? Type r to roll. r
You rolled a 2.
Your round 1 score is 10. Type r to roll again. r
You rolled a 6.
Your final score is 20.
```

As you can see, the program printed a few things for us. First, we rolled a `2` and a `6`, so we did not lose our points. Second, both `2` and `6` carry a score of `10` each, for a final score of `20`. But let's look back at the nested `if` statements.

To even begin the game, we had to verify whether the player was ready to roll. That first `if` statement is necessary to carry on to the rest of the decisions. *Could we have rolled without the go-ahead?* Yes. But think of all the games out there, including app versions of traditional board games. In those apps, the player always hits a button or the die to play. This is a similar scenario.

After we have said we are ready to roll, decisions have to be made about the points. We needed `if`, `elif`, and `else` statements again to loop through the various options. Now, let's take a look at when we can use `for` loops and `range`.

Using for loops and range

The first thing we need to discuss with `for` loops is that there can sometimes be some confusion over variables and `for` loop conditions. To see what I mean, let's look at an example:

ch10_forLoop1.py

```
for letter in 'mountain':
    print(letter)
```

In the preceding snippet of code, `letter` isn't a variable. It's just telling Python that we want to iterate over each character in the word `mountain`. However, I could have called it anything. The program will do the exact same thing if I wrote it as follows:

```
for pin in 'mountain':
    print(pin)
```

In each case, using `letter`, or `pin`, or whatever word makes me happy at that moment, the output of the program looks like this:

```
m
o
u
n
t
a
i
n
```

As you can see, Python iterated through each letter in the word `mountain` and printed it to the console. This same thing can be done with numbers in a range. For example, if I wanted to print the numbers `1` through `10`, I could use a `for` loop, as follows:

ch10_forLoop2.py

```
for num in range(1, 11):
    print(num)
```

Wait, I said I wanted the numbers 1 through 10, so why is there an 11 in the range function? That's because the `range()` function always includes the minimum value in the range, but not the upper boundary. So, we need to add `1` to whatever our top number is. Let's take a look at what the output for this program looks like:

```
1
2
3
4
5
6
7
8
9
10
```

As you can see, each number is printed on a separate line. So, let's look at what we can do if we just wanted to add those same numbers to a list instead. Instead of printing them one at a time, we could append them to a new list, then print out the list. This is really helpful for some kinds of algorithms and games. But before I get into those, let's first look at how we append the numbers with just a few more lines of code:

ch10_forLoop3.py

```
myNumbers = []
for num in range(1, 11):
    myNumbers.append(num)
print(myNumbers)
```

Now, pay close attention to when you print the list. If you have it indented in correctly, it will print the list every single time you append a new number. But if your indentation is correct, you'll only print the final list so that the output looks as follows:

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

As you can see, we now have representations of the same information. In this case, we printed out the list of numbers. This is helpful if you're trying to append specific things to a list, even if it is with user input. We'll look at another list problem in the next section as we look at `while` loops, but before we move on, let's look at one more `for` loop problem and algorithm.

Let's say you wanted an algorithm that will print out the cubes of a range of numbers. There is one condition: the range will vary depending on user input. Let's take a look at an algorithm that does this:

ch10_forLoop4.py

```
print('This program will provide a list of cubes for you. ')
minRange = int(input('What\'s the minimum for your range? '))
maxRange = int(input('What\'s the maximum for your range? '))
listOfCubes = []

for value in range(minRange, maxRange+1):
    number = value**3
    listOfCubes.append(number)

print('Your list of cubes in the range(' + str(minRange) + ', '
\
      +str(maxRange) + ') is: ')
print(listOfCubes)
```

Let's note a few things from the preceding program:

- We converted the inputs into `int` types. We did this so that we can use the number in the mathematical expression inside the `for` loop.
- We have an empty list so we can append the cubes to it.
- We added `1` to `maxRange` because if not, the number would not be included in our result.

- We have a range based on two variables, which are the user input.
- We have a printed list of the values.

Here is what that program looks like when we enter the `range(3, 6)`:

```
This program will provide a list of cubes for you.
What's the minimum for your range? 3
What's the maximum for your range? 6
Your list of cubes in the range(3, 6) is:
[27, 64, 125, 216]
```

As you can see, the program took the input provided, a minimum of `3` and a maximum of `6`, and created a list of the cubes of the numbers in that range.

> **Important Note:**
> Remember that to get the exponents, we use the `**` symbol in Python. So, *2**2* is *4*, *3**4* is *81*, and so on.

Remember, the *control flow* is the order in which things are done. With `for` loops and `range`, we are asking the program to repeat things without having to enter each line separately, effectively manipulating the control flow to do things in loops rather than one line after another after another.

Now that we've looked at `for` loops and `range`, let's take a look at other ways we can iterate in Python.

Using other loops and conditionals

Python offers a variety of ways to iterate through information in addition to the `for` loop. One of the commonly used loops is a `while` loop. When using a `while` loop, we're checking for a condition constantly. Again, it's easier to understand this by looking at an example.

Let's say we were playing a game and ask the user to provide as many animals as they can. We'd want to be able to do a few things:

1. Check whether the player has animals to add.
2. If yes, append each animal to a list.
3. If no, end the program and provide the final list.
4. Provide the final list and a count for how many animals the player was able to enter.

Let's take a look at an algorithm that does the basics for this game:

ch10_whileLoop1.py

```
myAnimals = []
print('Let\'s see how many animals you can name. Get ready!')
readyPlayer = input('When you are ready to begin, type y. ')
while readyPlayer == 'y':
    if readyPlayer == 'y':
        animalAdd = input('Name an animal. ')
        myAnimals.append(animalAdd)
        readyPlayer = input('Ready for the next one? Type y for 
yes or n for no. ')
howMany = len(myAnimals)
print('You were able to name ' + str(howMany) + ' animals. 
Here\'s your list: ')
print(myAnimals)
```

Let's break this code down a bit:

1. We created an empty list, `myAnimals`.
2. We then asked the player whether they were ready.
3. If the answer was yes, we asked for an animal.
4. Then, we checked whether they had more to add.
5. We checked how many items were added to the list.
6. We printed the list.

Notice that we had to use a `while` loop and `if` statements. That's because we want to continue asking the player to add animals until they are done or can't think of any more to add. So, while the player can still add items, we want the program to keep asking the question. But if the player is done, then they can say no to adding another animal, therefore ending the program.

Now, I want to be clear, this program can be made a lot better. We could add a timer to see how many items can be entered in 30 seconds. We could also add a conditional that breaks the program and explains why it does so when the player tries to enter an animal instead of saying yes first. You could add a condition for when the player tries to enter an animal that's already on the list. *A challenge for you would be to try to add those components to the existing program!*

But let's take a look at what this program looks like when we run it:

```
Let's see how many animals you can name. Get ready!
When you are ready to begin, type y. y
Name an animal. bird
Ready for the next one? Type y for yes or n for no. y
Name an animal. dog
Ready for the next one? Type y for yes or n for no. y
Name an animal. cat
Ready for the next one? Type y for yes or n for no. y
Name an animal. mouse
Ready for the next one? Type y for yes or n for no. y
Name an animal. elephant
Ready for the next one? Type y for yes or n for no. n
You were able to name 5 animals. Here's your list:
['bird', 'dog', 'cat', 'mouse', 'elephant']
```

Notice that the way this program is written requires us to answer whether we want to add an animal each time. That's because the conditions will run if we answer `yes` for as long as we answer `yes`. But once we answer `no`, the program ends, giving us the list of animals and how many we were able to name.

Now, let's take a look at another `while` loop. This time, we'll check whether or not the condition is `True`. Take a look at the following snippet:

ch10_whileLoop2.py

```
while True:
    num = int(input('Please enter an integer 0 through 9. '))
    if num in range(0, 10):
        print(num)
    else:
        print('That\'s not in range. ')
        break
```

In the preceding algorithm, the control flow dictates that we get asked for a number between `0` and `9` repeatedly. The program will continue to ask until we make a mistake. That's because it will continue to be `True` if I give it a number between `0` and `9`. Let's look at a sample output for this:

```
Please enter an integer 0 through 9. 0
0
Please enter an integer 0 through 9. 1
1
Please enter an integer 0 through 9. 2
2
Please enter an integer 0 through 9. 3
3
Please enter an integer 0 through 9. 4
4
Please enter an integer 0 through 9. 39
That's not in range.
```

Notice that the program kept asking the same question over and over again. This is unhelpful sometimes if the user doesn't know how to break the cycle. We could potentially add a line to our statement so that it asks the question but provides a hint. Take a look at the edited code:

ch10_whileLoop3.py

```
while True:
    num = int(input('Please enter an integer 0 through 9.
Tired? Type a number out of range. '))
    if num in range(0, 10):
        print(num)
    else:
        print('That\'s not in range. ')
        break
```

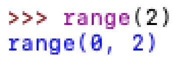
As you can see, we are now telling the user that if they get tired of providing ranges, then they can opt out by providing an input outside of the range. While this example doesn't seem very useful, think of all the applications you could have. This type of algorithm can be used for card games, for example. You could also use something like this to check input against an existing list.

Now, let's take a look at functions again, but now combining some of our loops and adding some capabilities.

Revisiting functions

If you recall from *Chapter 8*, *Introduction to Python*, we looked at built-in functions, but we also looked at how we can define our own functions. We are now going to talk about arguments in *functions* and *loops*, as we delve deeper into how *control flow* works in Python.

Let's think about problems that involve range. The range takes two arguments: a minimum and a maximum. However, in Python, I should note that you can just give one argument, which then assumes your minimum is `0`. For example, if I write `range(8)`, that's the same as `range (0, 8)`. Take a look at what happens if you type `range(2)` in the Python shell:



```
>>> range(2)
range(0, 2)
```

Figure 10.3 – Python range interpretation with one argument

In *Figure 10.3*, you can see that the program interpreted the code as `range(0, 2)`. But let's say you are always changing your range. Think of the range algorithm we wrote earlier. We are now going to rewrite it using a function. This function now has a `for` loop inside it as well:

ch10_functions1.py

```
minNum = int(input('What\'s your minimum number? '))
maxNum = int(input('What\'s your maximum number? '))


def myNumbers(minNum, maxNum):
    myList = []
    for num in range(minNum, maxNum + 1):
        myList.append(num)
    print(myList)

myNumbers(minNum, maxNum)
```

Notice that we are calling the function based on user input. When this program runs, it calls the function based on that input. We will revisit that and run the program by calling multiple ranges within the algorithm in a second, but take a look at what the preceding snippet gives as the output:

```
What's your minimum number? 3
What's your maximum number? 9
[3, 4, 5, 6, 7, 8, 9]
```

Notice how we also adjusted the maximum number in the range in the `for` loop to include the top number provided. That's so that we get the full list of numbers.

Now, let's take out the input completely. We are going to call the function multiple times using different ranges within the algorithm. Take a look at the updated snippet:

ch10_functions2.py

```
def myNumbers(minNum, maxNum):
    myList = []
    for num in range(minNum, maxNum + 1):
        myList.append(num)
    print(myList)


myNumbers(4, 9)
myNumbers(1, 3)
myNumbers(9, 17)
```

The final three statements are where we call the function. Since we defined the function to take two arguments, it uses both the arguments to run the function. Because we called the function three times, we should see three lists as output. Let's take a look:

```
[4, 5, 6, 7, 8, 9]
[1, 2, 3]
[9, 10, 11, 12, 13, 14, 15, 16, 17]
```

As you can see, we printed each range on a separate line. That's one of the most useful things functions can do for us. If we were working with images, for example, which we can do with libraries, then we could create a shape, then define a function with loops that changes some parameters. With one function and a few loops, we can create multiple circles in different locations with different radiuses based on calling the function and using some loops.

Functions are also not limited to two arguments. We can have multiple arguments and define them within the functions. Let's look at a function that uses three arguments:

ch10_functions3.py

```
def menu(appetizer, entree, dessert):
    print('Your appetizer is %s.' %(appetizer))
    print('You entree is %s.' %(entree))
    print('Your dessert is %s.' %(dessert))

menu('street tacos', 'chilaquiles', 'sopapillas')
```

In this case, we're calling the function with the values already given. *I love, and I mean love, Mexican food*. So, that menu would make me extremely happy! Here's what that output looks like:

```
Your appetizer is street tacos.
You entree is chilaquiles.
Your dessert is sopapillas.
```

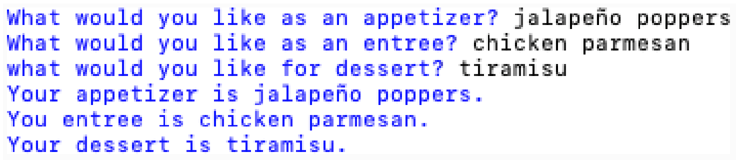
As you can see, the function uses each argument in the `print` statements. The `%s` statements are used to let the program know where the values will be substituted. The `%()` statements let the program know which value to grab from the call function.

Now, let's look at the code if we wanted to get input from the user:

ch10_functions4.py

```
appetizer = input('What would you like as an appetizer? ')
entree = input('What would you like as an entree? ')
dessert = input('what would you like for dessert? ')


def menu(appetizer, entree, dessert):
    print('Your appetizer is %s.' %(appetizer))
    print('You entree is %s.' %(entree))
    print('Your dessert is %s.' %(dessert))

menu(appetizer, entree, dessert)
```

As you can see, our definitions and arguments are repeated. We use `input` statements to get information from the program user, then print out statements. Here's what that looks like now in the Python shell:



```
What would you like as an appetizer? jalapeño poppers
What would you like as an entree? chicken parmesan
what would you like for dessert? tiramisu
Your appetizer is jalapeño poppers.
You entree is chicken parmesan.
Your dessert is tiramisu.
```

Figure 10.4 – User input in function with three arguments

As you can see, the first three lines take the user input, then the last three lines incorporate the inputs into the function definition. Depending on what you are looking to create, this matters. If you are building an online menu for a store, for example, you would want user input, but you would also want to be able to confirm purchases. Same with online libraries if you wanted to confirm an e-book being lent out, and so on. We use **confirmation statements** often in algorithms. After looking at these, go check out some of your favorite websites and see where there are confirmation statements based on user input. You'll see that the use is widespread. Functions allow us to simplify that process.

Before we move on, let's work with a few algorithms that provide the same information using our different loops and functions. Remember, we're looking at iterations because control flow means **order**. The iterations, functions, range, and more are ways in which we tell the program how it should respond to the instructions in the algorithm and when to repeat or move on from our program.

Let's look at a function that prints the triple of the given user maximum input:

ch10_functions5.py

```
numItem = int(input('What is your maximum number for the list
of triples? '))
def cost(numItem):
    while numItem > 0:
        print(numItem * 3)
        numItem -= 1
```

```


cost(numItem)
```

Notice that the `while` loop and function definitions depend on the user input. The program will then print the triple of the value provided by the user, then reduce that number by one and find the triple of that number. Let's take a look at what that means:

```
What is your maximum number for the list of triples? 4
12
9
6
3
```

As you can see, the program found the triple of `4`, which is `12`, then the triple of `3`, which is `9`, and so on. It stopped because we told it to run the loop while the number was greater than `0`. But recall that we can also just add those to a list, and we can use a `for` loop instead.

Let's take a look at a similar program with a function, but a `for` loop rather than the `while` loop and using `range`:

ch10_functions6.py

```
numItem = int(input('What is your maximum number for the list
of triples? '))
myList = []
def cost(numItem):
    for x in range(1, numItem + 1):
        newNum = 3 * x
        myList.append(newNum)
    print(myList)


cost(numItem)
```

Notice we defined some more things and added an empty list to the algorithm. While it has a few more lines of code, it does essentially the same thing as the previous code. Notice too that we started the range at `1`; otherwise, it would also include `0` in the list. Take a look at the output for this algorithm:

```
What is your maximum number for the list of triples? 4
[3, 6, 9, 12]
```

We have the same information but organized from least to greatest and in a list. From our algorithm, the order of things happening is important, of course. We first gathered the user input. The algorithm then defined the empty list, before defining a function. The function then used a `for` loop that uses the input to create a range and iterate over the range of numbers. Each iteration was then appended to the list. Finally, the algorithm printed the list.

As you can see, there are multiple ways to get to the same information. We just have to look at what works best for our scenarios, how to organize the information so that the program can read it, and write an algorithm that will organize the information in a way that can be read when running the program.

Summary

In this chapter, we discussed control flow and order by looking at `for` loops, `range`, `while` loops, and functions. **Control flow** refers to the order a program reads an algorithm. Typically in Python, one line is read right after the other. In this chapter, we learned how you can control that order. Specifically, we learned that we could do so by iterating through data. Here are some important points to remember: `while` loops run so long as a condition is met, `for` loops iterate over a sequence (string, numerical, list, dictionary, set, or tuple), and `range` is used to create a sequence of numbers.

We also learned that we can combine these things when creating conditions, defining functions, and designing our algorithms. The most important thing to continue keeping in mind is that order matters, so we need to be careful about when we define necessary variables and how to write the algorithms so that they don't run infinitely or break before they should. Control flow is important so that our algorithms work without errors.

In the next chapter, we will use the knowledge gained so far to complete the computational thinking process when solving challenges in multiple disciplines.

11
Using Computational Thinking and Python in Simple Challenges

In this chapter, we will take a look again at the computational thinking process and apply it to design algorithms that help us solve various scenarios and problems. As we wrap up *Section 2*, *Applying Python and Computational Thinking*, of this book, we will combine some of the knowledge about Python capabilities with the computational thinking process to solve these problems.

In this chapter, we will cover the following topics:

- Defining the problem and Python
- Decomposing the problem and using Python functionalities
- Generalizing the problem and planning Python algorithms
- Designing and testing the algorithm

By the end of this chapter, you will be able to design algorithms that best fit the scenarios presented. You will also be able to identify Python functions that best align with the problems presented and generalize your solutions.

Technical requirements

You will need to install the latest version of Python to run the code in this chapter.

The source code used in this chapter can be found in the GitHub repository here: `https://github.com/PacktPublishing/Applied-Computational-Thinking-with-Python/tree/master/Chapter11`

Defining the problem and Python

We are going to dive into our first scenario. You are launching a campaign for a product you've designed. You're looking for investments that add up to $100,000 in pledges. You want to create an algorithm that saves information on the pledges made for you, while also providing information about the products each pledge provides. There are some things you need to identify before you write an algorithm:

- *How many types of pledges will you run in your campaign?*
- *Will there be a donation-only pledge? Do donations get anything at all, such as a sticker, electronic certificate, or other product swag?*
- *What will each of the pledges give the person pledging?*

All of these questions will help us plan. But we also have to take into account other things, such as what we need from the algorithm. *Is it simply to list the number of pledges or will it also be used to check against an inventory or to create that inventory? Will it count down from a set number of pledges available for each level? Will it automatically update every time someone makes a pledge?*

As you know, when we are presented with a problem, it is critical that we identify what the problem really is. We need to set our parameters for the problem in order to provide a solution. In this section, we're going to want to answer a lot of these questions first. In order to do that, we need to decompose the problem and identify which functionalities in Python would help us solve this problem.

Decomposing the problem and using Python functionalities

We can create an algorithm that is used to design those pledges. We'd be designing an algorithm that tells us how many items we need of each type based on a set number of pledges.

For example, say we have three tiers of pledges and a donation-only pledge. For the highest tier, let's call that **Tier 1**, you get the following:

- Double the item you are selling at a 20% discount
- An accessories bundle for the item, which includes four items
- A carrying case for the item and accessories
- Access to a 1-year membership for the site

The middle tier, or **Tier 2**, gets you the following items:

- The item you are selling at a 20% discount
- A carrying case for the item
- Access to a 6-month membership for the site

The lowest tier, or **Tier 3**, gets you the following:

- The item you are selling at a 20% discount
- A carrying case for the item

Let's say we want an algorithm that will give us the number of items we'd need depending on how many pledges we allow for each tier. The **fourth tier**, which is donation only, would not affect our algorithm, since this algorithm would be used simply to determine how many of each item we'd need depending on the number of pledges.

But things aren't as simple as just the numbers. Let's say that you need $100,000 in pledges, as mentioned earlier in this chapter. Then you'd need to figure out how much you'd charge for the tiers and put a price point on them. You'd need to make enough pledges available to at least reach that goal of $100,000.

You have identified the retail price of your item as $800. Here are the costs for each of the items:

- $640 for the item itself (given a 20% discount, and only applicable to Tiers 1, 2, and 3)
- $100 in accessories (Tier 1 only)
- $40 carrying case (applicable to Tiers 1, 2, and 3)
- Membership costs will be passed at $10 a month (let's say the original cost is $35 a month)

For Tier 1, the pledge would need to be a minimum of $1,540:

- $640 \times 2 = 1280$
- $12 \times 10 = 120$
- $1280 + 120 + 100 + 40 = 1540$

For Tier 2, the pledge would need to be a minimum of $740:

- $640 + 40 = 680$
- $6 \times 10 = 60$
- $680 + 60 = 740$

For Tier 3, the pledge would need to be a minimum of $680, that is, $640 + 40 = 680$.

Now we have to figure out how many of each tier would be available for purchase. But first, let's round out some of these numbers: Tier 1 will be $1,600, Tier 2 will be $800, and Tier 3 will be $700.

We can write an algorithm to let us know how many pledges we need for each tier. But before we do, let's talk about Python and the functionalities that we could use. We can create the Tier-3 pledge first, making it a parent group. Then we can create two child classes, Tiers 1 and 2, that have slightly different characteristics and costs. The question we ask ourselves is whether we need to be that detailed on the algorithm or can we just add each tier's cost/value.

That's a good question. And the answer is that it depends on what you need this algorithm for. If you're looking to do the early yet simple calculations and need an algorithm you can run every so often based on current sales/pledges, then you can create something simple with a dictionary or a function. But if you want to use the algorithm for multiple purposes or need to be able to adjust some costs later on, then you may want to code that into the algorithm with all the details.

Why would you need that? Take some of the successful campaigns available. Oftentimes, the original pledges sell out for those campaigns. New pledge tiers then become available and you may want to adjust the price point of those tiers. For example, rather than a 20% discount on the item, the pledge may be for a 15% discount. Other changes may happen, such as selling swag instead of the accessories due to inventory limitations and so on.

The more detailed your algorithm is, the more you can do with it in the long run. But if you do not need all that information, then you may not want to spend as much time creating the algorithms.

Let's get to work on this algorithm. Take a look at the snippet shown as follows:

ch11_pledgesA.py

```
import math
tier1 = 1600
tier2 = 800
tier3 = 700
perc = int(input("What percentage of the 100,000 do you
    expect will be Tier 1? Type a number between 1 and
    100. "))
percentage = perc/100
Tier1 = (100000*percentage)/1600
totalTier1 = math.ceil(Tier1)
print("You will need to sell %s Tier 1 pledges if you want
    %s percent of your sales to be in Tier 1."
    %(totalTier1,perc))
```

Let's take a look at some of the things we used in the preceding snippet. We had to import the `math` module so we could use the `math.ceil()` math function. We used this function in order to round the number of Tier-1 pledges needed up. That's because if we rounded down, we would not be able to cover the desired percentage. By doing this, we are finding the smallest integer we'll need to cover the percentage.

In addition, we used the `%s` placeholders in our `print` statement to call our strings, then stated the strings to be used.

When we run that program, this is what the output looks like:

```
What percentage of the 100,000 do you expect will be Tier 1?
Type a number between 1 and 100. 45
```

```
You will need to sell 29 Tier 1 pledges if you want 45 percent
of your sales to be in Tier 1.
```

Notice that our `print` statement uses the percentage entered by the user, in part to ensure that the information matches what was expected. For 45% of the funding to come from Tier-1 pledges, we'd need to sell at least 29 Tier-1 pledges. If we run the math just to verify that this is correct, we see that this information is true:

$$29 \times 1600 = 46{,}400$$
$$100{,}000 \times 0.45 = 45{,}000$$

As you can see, if we sell 29 Tier-1 pledges, we'll make slightly more than 45% of 100,000.

But let's say that you also want the algorithm to tell you how many items you need based on the number of Tier-1 pledges. Let's take a look at the adjusted algorithm:

ch11_pledgesB

```
import math
tier1 = 1600
tier2 = 800
tier3 = 700
perc = int(input("What percentage of the 100,000 do you
    expect will be Tier 1? Type a number between 1 and
    100. "))
percentage = perc/100
Tier1 = (100000*percentage)/1600
totalTier1 = math.ceil(Tier1)
print("You will need to sell %s Tier 1 pledges if you want
    %s percent of your sales to be in Tier 1."
    %(totalTier1,perc))
print("You will need %s items, %s accessories kits, %s
    carrying cases, and %s year-long memberships."
    %(totalTier1*2, totalTier1, totalTier1, totalTier1))
```

Notice that I only added one more `print` statement. There are pros and cons to using this algorithm in this way. In this case, I'm only outputting the numbers for each of the tiers. I'm not saving how many items I need anywhere in the algorithm. If we want to do that for future reference, we'll need to adjust how we get that information and how we save it in the algorithm.

For this algorithm, the output is as follows:

```
What percentage of the 100,000 do you expect will be Tier 1?
Type a number between 1 and 100. 45
You will need to sell 29 Tier 1 pledges if you want 45 percent
of your sales to be in Tier 1.
You will need 58 items, 29 accessories kits, 29 carrying cases,
and 29 year-long memberships.
```

Notice that we got the information we wanted. We'll need `58 items`, `29 accessories kits`, `29 carrying cases`, and `29 year-long memberships`. Again, this would be helpful if we were doing a one-off thing or if we didn't expect any changes. But let's be clear, that's almost never the case. We'll want to make changes. We'll also need to know information based on Tier-2 and Tier-3 selections. *So what can we do?*

First, we'll want to save our numbers. So, we'll need to add some variables for the items, the accessories kits, the carrying cases, and two variables for the memberships, one for the year-long and one for the 6-month-long memberships. We'll also need to make a decision about how we want the rest of the pledges broken up. *Do we want the other percentage equally split between Tiers 2 and 3? Do we want one-third of what's left to be Tier 2 and two-thirds to be Tier 3?* Let's go with those numbers. Here's where we stand now:

- The Tier-1 percentage is chosen by the user as the program is run.
- The Tier-2 percentage will be one-third of the remaining percentage.
- Tier 3 will be two-thirds of the remaining percentage.

Let's teach this to the algorithm. The following file contains the full, uninterrupted code. We added some text to explain certain sections, as follows:

ch11_pledgesC.py

```
tier1 = 1600
tier2 = 800
tier3 = 700
perc = int(input("What percentage of the 100,000 do you
    expect will be Tier 1? Type a number between
    1 and 100."))
percTier1 = perc/100
percTier2 = (100-perc)/3/100
percTier3 = (100-perc-percTier2)/100
```

Notice in the following snippet we are adding some variables, such as `totalTier1`, `itemsTier1`, `accTier1`, and `cases1`. These variables will help us save the numbers of each tier ordered. We'll do the same for Tiers 2 and 3:

```
Tier1 = (100000*percTier1)/1600
totalTier1 = math.ceil(Tier1)
itemsTier1 = totalTier1*2
accTier1 = totalTier1
cases1 = totalTier1
yearMemb = totalTier1
Tier2 = (100000*percTier2)/800
totalTier2 = math.ceil(Tier2)
itemsTier2 = totalTier2
cases2 = totalTier2
sixMemb = totalTier2
Tier3 = (100000*percTier3)/700
totalTier3 = math.ceil(Tier3)
itemsTier3 = totalTier3
cases3 = totalTier3
totalItems = itemsTier1 + itemsTier2 + itemsTier3
totalAccessories = accTier1
totalCases = cases1 + cases2 + cases3

print("You will need to sell %s Tier 1 pledges if you want
    %s percent of your sales to be in Tier 1."
    %(totalTier1, perc))
print("You will need %s Tier 2 pledges and %s Tier 3
    pledges to meet or exceed your $100,000 funding goal."
    %(totalTier2, totalTier3))
```

While we haven't printed the details yet for the number of total items or the total cases, we now have them saved into variables. This is what our output looks like now:

```
What percentage of the 100,000 do you expect will be Tier 1?
Type a number between 1 and 100. 50
You will need to sell 32 Tier 1 pledges if you want 50 percent
of your sales to be in Tier 1.
You will need 21 Tier 2 pledges and 72 Tier 3 pledges to meet
or exceed your $100,000 funding goal.
```

We should note that we exceed our funding goal because we've always been rounding up. That is, rather than using $1,540 for Tier 1, we used $1,600. For the percentage, we've been rounding up. All of these will add up to give us a total above $100,000.

Let's extend the algorithm a little more. The following is only the new snippet from the algorithm we've already seen, which contains the total for the items we'll need:

```
print("These percentages are equivalent to %s total items,
    %s total cases, %s accessories kits, %s year-long
    memberships, and %s six-month memberships." \
      %(totalItems, totalCases, totalAccessories,
      yearMemb, sixMemb))
```

Notice that we can now call those variables we added in our `print` function to get the counts we need for our inventory. We would not be able to get those details if we had not defined those items in our algorithm.

Also notice that in our previous snippet, some of the items have the exact same value. However, we still defined them with different variables. Take a look, for example, at `cases2 = totalTier2` and `sixMemb = totalTier2`. Although both have the same values, we want to identify them separately. And maybe that's not important now, but later on, maybe we'll run out of cases. Then we'd only want to change the value for the cases and not the 6-month memberships.

Since they're already split, we can change one without affecting the other. Let's take a look at what the output looks like for the new `print` statement:

```
These percentages are equivalent to 157 total items, 125 total
cases, 32 accessories kits, 32 year-long memberships, and 21
six-month memberships.
```

Looking at this, you may realize that you only get one carrying case but two items in Tier 1, which is why there's a different count for those. The accessories and year-long memberships only happen in Tier 1, so it makes sense that those two numbers are the same. The 6-month memberships are only for Tier 2, so that number matches the number of Tier-2 pledges.

As we consider this problem further, we may realize that we may want to save information differently. Maybe rather than asking the user for the percentage of Tier-1 pledges they want, we could ask how many total items they have and then break down the tiers based on that. All of that is possible, so how we define the problem is critical. How we save the information or request input from the user is also just as important. Decomposing a problem is just part of the process of creating and designing the algorithms we need.

In this section, we learned how to address problems with multiple solutions depending on our goals. As we define our problems, we are often also identifying the variables we'll need and determining what kinds of functions are most useful depending on what we'll want to get out of the algorithm. Unless we have a very simple and straightforward problem, the decomposition and definition of the problem are critical in order to successfully define an algorithm.

Now let's take a look at generalizing the problem in the next section.

Generalizing the problem and planning Python algorithms

In the previous section, we were working with an algorithm designed for use in a funding campaign. The algorithm we looked at was problem-specific.

Now let's try to generalize this problem and understand how we could potentially design a different algorithm. *Why would we need that?* Think of it as a template. If you run multiple funding campaigns for start-ups, you may want to create a general algorithm that you can then adapt based on the needs of each campaign rather than having to start each campaign over.

You would need to set up some clear parameters and make some decisions. In order to keep this scenario manageable for the book's purposes, we're going to limit our choices a bit:

- Every campaign will have between 3 and 5 tiers of pledges, not including donation-only.
- Every tier will ask for the items needed for each tier.
- Each tier option will have a set percentage built in.

If there are three tiers, Tier 1 will be 50% of pledges, Tier 2 will be 30%, and Tier 3 will be 20%. If there are four tiers, Tier 1 will be 40% of pledges, Tier 2 will be 30%, Tier 3 will be 20%, and Tier 4 will be 10%. If there are five tiers, Tier 1 will be 40% of pledges, Tier 2 will be 30%, Tier 3 will be 15%, Tier 4 will be 10%, and Tier 5 will be 5%.

Take a look at the following diagram, which shows the breakdown of the tiers:

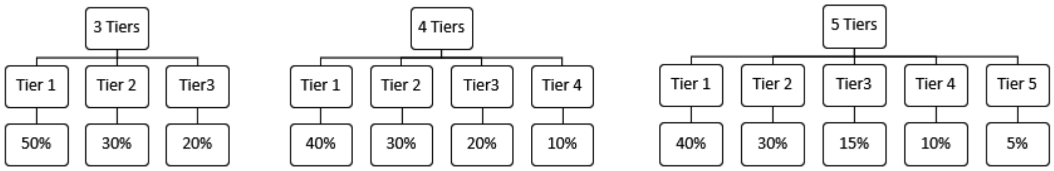


Figure 11.1 – Tier percentage breakdown

Because we are using one algorithm for many possible scenarios, we'll break down that algorithm piece by piece. The full algorithm can be found in the `ch11_pledgesTemplate.py` file on GitHub.

In this first snippet, we ask for the initial input, then save our percentages:

```
import math
numberTiers = int(input("How many tiers of pledges will you
    offer? Enter a number between 3 and 5 inclusive. "))
#Number of tiers percentages
if numberTiers == 3:
    tier1 = .50
    tier2 = .30
    tier3 = .20
elif numberTiers == 4:
    tier1 = .40
    tier2 = .30
    tier3 = .20
    tier4 = .10
elif numberTiers == 5:
    tier1 = .40
    tier2 = .30
    tier3 = .15
    tier4 = .10
    tier5 = .05
else:
    print("Please try again and enter the numbers 3, 4,
        or 5. ")
```

As you can see, there are three conditions after we ask for input. Notice that we converted the input to an integer. This is important, otherwise the conditions will run but the `else` condition will also run.

The preceding snippet won't give us any output except for asking for that input. So, we'll show more output after our next snippet.

Important Note:
Notice the use of comments in this code. Because of the length of the code, we'll be using comments in order to find spots in our code where we need to edit it. It is always a good idea to use comments in all code, but especially in lengthy code. Otherwise, finding the particular lines we want to change can be very tricky.

Once we have the number of tiers, we'll need to know the number of items in each tier. We'll need to ask how many items are chosen for each tier. Let's take a look at the continuation of the preceding code:

```
#Number of items per tier
if numberTiers == 3:
    numTier1Items = int(input("How many items will be
        provided for a Tier 1 pledge? "))
    numTier2Items = int(input("How many items will be
        provided for a Tier 2 pledge? "))
    numTier3Items = int(input("How many items will be
        provided for a Tier 3 pledge? "))
elif numberTiers == 4:
    numTier1Items = int(input("How many items will be
        provided for a Tier 1 pledge? "))
    numTier2Items = int(input("How many items will be
        provided for a Tier 2 pledge? "))
    numTier3Items = int(input("How many items will be
        provided for a Tier 3 pledge? "))
    numTier4Items = int(input("How many items will be
        provided for a Tier 4 pledge? "))
```

Notice that we're only showing the conditions for when the tiers are 3 or 4. The code file will also have the information for the 5 tiers, but it follows what is shown in the preceding code. Notice that the algorithm asks for input for each of the tiers. This will be important when we need to work with numbers and percentages.

In addition, we can include an `else` statement that allows us to ask the questions again if there is an error in the input. You can add these kinds of conditions to the existing template if you wish. For now, we're going to continue with the next pieces of information we'll need to get from the user, such as the price point for each tier.

Now let's think back to what we could need. We'll need price points for each tier, which will also be the input requests for a template kind of algorithm. Since each of those will be different for each campaign, we'll need to leave that up to the user to enter. The input lines will look very similar to the previously shown snippets. Here is what it looks like for 3 tiers:

```
#Price points for each Tier
if numberTiers == 3:
    priceTier1 = int(input("What is the price point of Tier
        1? Enter dollar amount without dollar sign. "))
    priceTier2 = int(input("What is the price point of Tier
        2? Enter dollar amount without dollar sign. "))
    priceTier3 = int(input("What is the price point of Tier
        3? Enter dollar amount without dollar sign. "))
```

Again, notice that we're using comments to separate each section of the code. As you can see, we are adding information about how much we are charging for each pledge level. The code then continues to do this for the number of tiers, 3, 4, or 5.

As was previously discussed, you may also want to test for errors or provide an alternative to continue running the code after an error is added by the user. We are not addressing those errors in this code but they can certainly be added to improve the user experience with this algorithm. As you can see, we've started working on how to generalize this algorithm for multiple scenarios.

In this case, we're generalizing for multiple uses. But we've used a lot of algorithms and seen a lot of scenarios in this book where the generalization of the patterns is much simpler. Generalization can be something as simple as writing an equation with one variable. Or it can be creating an algorithm for a number of circumstances and conditions. That's why it's important to identify what our problem is and what exactly we want to accomplish.

In this section, we looked at how and when to get input from the user. We also worked through defining variables in order to store the input and use it in our equations for the algorithm to output the necessary information.

To complete the algorithm, we'll go to the next section, which is focused on the design and testing of the algorithm.

Designing and testing the algorithm

In the previous section, we got a lot of input from the user because we were creating an algorithm that was used as a template for many campaigns. Now we need to use that input in a variety of ways. Here's what we have so far:

- The number of tiers of pledges
- The percentage breakdown of the tiers
- The number of items for each tier
- The cost of each tier

Now we can do something with all that information. First, let's talk about what we could want. We could want to test how much we'd make selling a specific quantity of the tiers. We could also break down how many of each tier we'd need based on a funding goal, much as we did in the first part of this chapter.

What would be most helpful? Well, that really depends on what you need. I'm going to say that I want the breakdown based on the funding goal. I want to know how many of each pledge type I will need to make available. So now I have to figure out how I'm going to get that information from the variables I've already defined.

We'll need to have three conditions here as well. And because the variable and how many there are of each variable differs per tier type, we'll need to make sure we're accounting for that information. Let's think about three tiers first. Given the funding goal, here are some of the outputs that can be helpful:

- The number of each tier of pledges to be made available
- The number of items per tier that will need to be in inventory

So how do we figure that out?

Let's say that we had $50,000 as our funding goal and let's assume that the Tier-1 cost is $500. Then, we can take the following steps to find the number of Tier-1 pledges needed:

1. Multiply the funding goal by the percentage, that is, 50,000 × 0.50 = 25,000.
2. Divide the resulting number by the cost of the pledge, that is, 25,000 ÷ 500 = 50.

That means that we'll need to post 50 pledges for Tier 1. Now let's assume that the user entered that there were `3` items in Tier 1. Then, this means 50 × 3 = 150 items.

Now let's see that in our code. Remember, this is the same file as the previous snippets (`ch11_pledgesTemplate.py`). We're continuing with the discussion using pieces of the code:

```
#Breakdown of number of Tiers based on funding goal
fundGoal = int(input("What is the funding goal for this
    campaign? Enter dollar amount with no symbols. "))

if numberTiers == 3:
    Tier1Total = math.ceil(fundGoal*tier1/priceTier1)
    Tier2Total = math.ceil(fundGoal*tier2/priceTier2)
    Tier3Total = math.ceil(fundGoal*tier3/priceTier3)
    print("For a funding goal of %s with %s tiers, you'll
        need %s Tier 1 pledges, %s Tier 2 pledges, and %s
        Tier 3 pledges. " % (fundGoal, numberTiers,
        Tier1Total, Tier2Total, Tier3Total))
```

In the preceding snippet, we have a `print` function with the number of pledges for each tier, but they're also saved as functions in our conditional statement. Notice that we'll now have some output here. We'll get the number of pledges we'll need from this snippet, but not the number of items per tier. We'll add that shortly. For now, here's what that output looks like when we run the program:

```
How many tiers of pledges will you offer? Enter a number
between 3 and 5 inclusive. 3
How many items will be provided for a Tier 1 pledge? Enter a
number between 1 and 3 inclusive. 3
How many items will be provided for a Tier 2 pledge? Enter a
number between 1 and 3 inclusive. 2
How many items will be provided for a Tier 3 pledge? Enter a
number between 1 and 3 inclusive. 1
What is the price point of Tier 1? Enter dollar amount without
dollar sign. 500
What is the price point of Tier 2? Enter dollar amount without
dollar sign. 400
What is the price point of Tier 3? Enter dollar amount without
dollar sign. 350
What is the funding goal for this campaign? Enter dollar amount
with no symbols. 50000
For a funding goal of 50000 with 3 tiers, you'll need 50 Tier 1
pledges, 38 Tier 2 pledges, and 29 Tier 3 pledges.
```

As you can see, we now know that we need to list 50 Tier-1 pledges at $500, 38 Tier-2 pledges at $400, and 29 Tier-3 pledges at $350 to reach our funding goal. Now we have to figure out how many items we need for each tier given the number of items per tier provided. Here's what that code will look like:

```
if numberTiers == 3:
    Tier1Total = math.ceil(fundGoal*tier1/priceTier1)
    Tier2Total = math.ceil(fundGoal*tier2/priceTier2)
    Tier3Total = math.ceil(fundGoal*tier3/priceTier3)
    print("For a funding goal of %s with %s tiers, you'll
        need %s Tier 1 pledges, %s Tier 2 pledges, and %s
        Tier 3 pledges. " % (fundGoal, numberTiers,
        Tier1Total, Tier2Total, Tier3Total))
    Tier1Items = numTier1Items*Tier1Total
    Tier2Items = numTier2Items*Tier2Total
    Tier3Items = numTier3Items*Tier3Total
    print("For %s Tier 1 pledges, you'll need %s items. For
        %s Tier 2 pledges, you'll need %s items. For %s
        Tier 3 pledges, you'll need %s items. "
        %(Tier1Total, Tier1Items, Tier2Total, Tier2Items,
        Tier3Total, Tier3Items))
```

As you can see, now we have another three math equations and a `print` statement that breaks down the information for us. We'll get the number of pledges for each tier as well as the number of items we'll need for each of the tiers. If you wanted even more information from this template, you could include pieces from the first example in this chapter, where we broke down the types of items per pledge. We'll leave that up to you as a challenge.

For now, here's what our final output would look like for three tiers and a funding goal of $50,000:

```
How many tiers of pledges will you offer? Enter a number
between 3 and 5 inclusive. 3
How many items will be provided for a Tier 1 pledge? 3
How many items will be provided for a Tier 2 pledge? 2
How many items will be provided for a Tier 3 pledge? 1
What is the price point of Tier 1? Enter dollar amount without
dollar sign. 500
What is the price point of Tier 2? Enter dollar amount without
dollar sign. 400
What is the price point of Tier 3? Enter dollar amount without
dollar sign. 350
What is the funding goal for this campaign? Enter dollar amount
with no symbols. 50000
For a funding goal of 50000 with 3 tiers, you'll need 50 Tier 1
pledges, 38 Tier 2 pledges, and 29 Tier 3 pledges.
For 50 Tier 1 pledges, you'll need 150 items. For 38 Tier 2
pledges, you'll need 76 items. For 29 Tier 3 pledges, you'll
need 29 items.
```

As you can see, we not only have the information we needed, but we also have variables set up to use if we need to adapt this information. Thinking back to previous chapters and notes that we've been discussing, let's try to determine how else we could now save the information.

The first thing that comes to mind is that we could create a dictionary that stores the information for us. If we did that, then we could recall the information we needed from that dictionary, such as the number of items for one tier, for example. We could also adjust the key-value pairs if we needed to without having to enter the entire thing all over again. Say our initial cost for Tier 1 was $500 but we now need it to be $600, yet the other tiers wouldn't change. Then we could just adjust that one value.

This scenario would allow you to explore many of the functionalities of the Python programming language we've discussed. Take some time to study the code, then make some adjustments and try to use some of your knowledge to improve upon it based on different conditions.

Remember, we are always given problem situations that can be interpreted in different ways. It is up to us to write algorithms that meet our needs and the needs of our clients. Sometimes, we'll get clarification from our stakeholders directly. Other times, we'll have to ask for that clarification and/or make some of the assumptions ourselves. What is critical is that we design the algorithms and document our progress so that we can adjust, adapt, and change pieces of our work without having to start over if we don't get what we need.

Summary

In this chapter, we went over the computational thinking process one more time by working through a more complex scenario and interpretations of that scenario. We learned how to decompose the problem provided, then identify the patterns, generalize them, and design the algorithms. We used some of what we've learned throughout the book to write an algorithm that provided the information we needed.

The computational thinking process helps us develop skills that make our algorithm planning much easier. By walking through that process, we learn more about what Python capabilities and functions may help us in particular scenarios. We also learned how to generalize patterns, sometimes in simple equations for a problem, but other times in creating algorithms that can help us in multiple scenarios without having to recreate them each time. As we got to learn more about Python, we got more comfortable with the computational thinking process in this last chapter of *Section 2*, *Applying Python and Computational Thinking*.

In *Section 3*, *Data Processing, Analysis, and Applications Using Computational Thinking and Python*, we will move on to other capabilities of Python in order to deal with data processing, analysis, and applications using computational thinking elements. In the next chapter, we'll start to look at data and how we can use Python to analyze data, create visual representations, and write algorithms that work with experimental data.

Section 3: Data Processing, Analysis, and Applications Using Computational Thinking and Python

Data is all around us. We use data to inform decisions about policies, such as how many resources are provided for a school district, how large the budget for a county or state's Medicare program will be, how much we pay for housing in an area, and the trends of the real estate market. Data is embedded in the way that we interact with advertisements too. Simply put, understanding more about data and how to use Python to analyze this data is a really important skillset.

In this section, you will be introduced to advanced functionalities of the Python programming language associated with data analysis and other applications, such as cryptography, experimental data, data mining, and machine learning. We will use computational thinking to design solutions and algorithms for many real-world, applied problems, from those in language and historical analysis to some machine learning applications.

This section comprises the following chapters:

- *Chapter 15, Applied Computational Thinking Problems*
- *Chapter 16, Advanced Applied Computational Thinking Problems*

12
Using Python in Experimental and Data Analysis Problems

In this chapter, we will take a look at how Python can help us understand and analyze data using algorithms and libraries created specifically for data analysis and data science. We will first go through experimental data and then move on to algorithms that use two main libraries: **NumPy** and **pandas**.

In this chapter, we will cover the following topics:

- Defining experimental data
- Using data libraries in Python
- Understanding data analysis with Python
- Using additional libraries for plotting and analysis

By the end of this chapter, you will be able to define types of experiments, data gathering, and how computational thinking helps when designing models and solutions. You will also learn how to use data libraries, particularly, **NumPy**, **pandas**, and **Matplotlib**, to help in analyzing and displaying data. Finally, you'll be able to design algorithms that help with data analysis in order to learn from existing data.

Technical requirements

You will need to install the latest version of Python for running the code in this chapter.

You will also need to have a few libraries installed, including **NumPy**, **pandas**, **Matplotlib**, and **Seaborn**.

You may choose to use an integrated environment to run Python, such as **Anaconda**, which can simplify the library dependencies and is helpful in organizing your algorithm in notebooks.

The source code used in this chapter can be found in the GitHub repository here: `https://github.com/PacktPublishing/Applied-Computational-Thinking-with-Python/tree/master/Chapter12`

Defining experimental data

We're finally at the data chapter of this book. We all have our biases and areas where we just love to live in. This is one of mine. There are many reasons why data is so important, but let's start with the fact that data, how we collect it, how we analyze it, and how we present it has a massive impact on our daily lives.

When writing algorithms to display information, we have a responsibility to share that data in the least biased way possible, making sure that our data is inclusive and representative of our communities and our people. I wanted to make sure I said that before we talk about the topic in as much depth as a chapter will allow me to. For me, this is how I fell in love with code and Python.

In this section, we're going to go over experimental data, defining what it is as well as key terms used when working with experimental data.

Now, let's get started. **Experimental data** is a term that gets its use from science and engineering. However, we also use experimental data in other areas, such as education, civics, and much more. Experimental data includes the methodology, how we measure data, the design of the experiments we are conducting, and how we present the data we gather and analyze from those designs. For the purposes of this chapter, we're not designing the experiments. We're going to be focusing on how we use Python to analyze existing data. But it is important to note that, if we have a say in the experiment, we need to make sure we are designing it fairly.

If the experimental design and the data collection methods are sound, then the data we use and analyze will be much more helpful than if we begin with a biased experiment. While we can never eliminate bias completely, as researchers and developers, it is our responsibility to present data with the least amount of bias possible. *Why is this important?* Think of all the policy changes that happen based on data, graphs, and information presented from experiments. If the experiments are biased to begin with, the results will lead to policies that may not adequately encompass the needs of a community, for example.

When working with data, we are always using computational thinking elements. As we tackle problems, we have to define what the problems are, what we want to study, how we want to measure it, how we'll be able to create and generalize the patterns, and what algorithm we'll need to use to produce the best representations of our data. Everything we do in data analysis benefits from us using computational thinking elements.

Data science is a growing field in **STEM**. In 2017, it was named the fastest-growing field in the US. The US Bureau of Labor Statistics stated that there is an estimated 11.5 million new jobs expected in data science and statistics by 2026. Currently, there are more jobs available than qualified candidates.

Now, let's look at how Python lets us tackle data.

In experimental data, we want to gather information using independent, dependent, and control variables:

- **Independent variables** are variables that are changed or controlled by the researcher.
- **Dependent variables** are variables that are being measured or tested by the researcher; dependent variables *depend* on independent variables.
- **Control variables** are variables or factors in the experiment that must remain the same.

Let's look at a simple example of these variables in an experiment. A common example involves the study of plant growth. The independent variables can be one or multiple variables. For our example, our independent variable will be the amount of fertilizer we add to the plants. The plant growth will be a dependent variable. The **control** will not get any fertilizer, just water. So, in our experiment, let's say we have five plants that we will be measuring. One plant will only get water. That's our **control**. The other four will have varying levels of fertilizer added to them. Those are our **experimental** plants. The growth is **dependent** on the amount of fertilizer.

When we design experiments, there are three things we want to be true: that they are reliable, valid, and can be generalized. Here's what each of these things means:

- **Reliable** relates to the consistency of the measurements. This means that if we mimic the conditions, our results should be reliably similar.
- **Valid** relates to whether or not the experiment measured what it intended to measure.
- **Generalizable** relates to the results being something that can be generalized and applied to other settings.

There is a lot more detail and depth we could go into about experiments, but we are only looking at what happens once we have the data for this chapter. It is important to understand these basic terms for when we are involved in the design of the experiments. As developers, depending on our roles, that may be the case. For example, in a start-up, everyone may be involved in all aspects of product development. So, in those instances, we could be working on the experiment and also the subsequent analysis algorithms.

For now, let's move on to what we can do with data analysis and how Python libraries can help us achieve what we need when analyzing the experiment results.

Using data libraries in Python

In this section, we're going to take a look at some libraries and packages that we can use with the Python programming language. We refer to *packages* and *libraries*, using both terms interchangeably sometimes, but for clarity, a package contains modules, while a library is a collection of packages.

We use Python libraries much like we use modules that are built-in, such as the `math` module we first used in *Chapter 4*, *Understanding Logical Reasoning*. In our source code, we imported the `math` module by using `import math` before creating the algorithm. In the example under the *Applying inductive reasoning* section of *Chapter 4*, *Understanding Logical Reasoning*, we used the `math.floor()` function of the module, which allowed us to round a number down, regardless of what the decimal value was. When we import modules or libraries in Python, we are tapping into additional functions and capabilities that allow us to take the programming language much further.

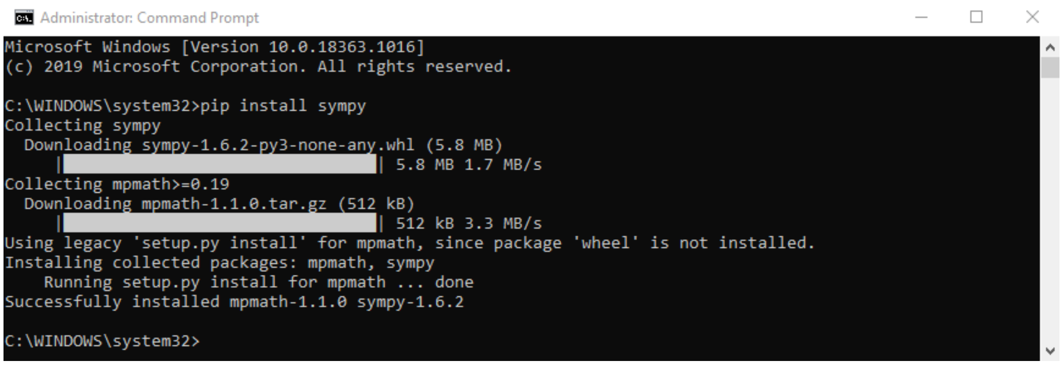
So, what is a library? In Python, a **library** refers to chunks of code that can be reused. Libraries contain a collection of modules. There are many libraries available for Python, and like Python itself, a large number of the libraries are **open source**, which means they can be downloaded and used by anyone. Because we're going to be working with data in this chapter, we're going to stick with pandas, NumPy, and Matplotlib for now. However, there are many other libraries and many types of libraries. For example, there are **Graphical User Interface** (**GUI**) frameworks, such as **Kivy**, **tkinter**, **PyQt**, **PySimpleGUI**, and others. For gaming, there are other libraries, such as **Pygame** and **Pyglet**. In machine learning, the **TensorFlow** library is a popular tool developed by **Google** in collaboration with the **Brain** team. But these are only some examples of the libraries available and the areas that use them.

Installing libraries

Although the `math` module is built into the Python language, libraries need to be installed. In **Python 3.9**, the program we use to install libraries is the **pip installer**. This is a command built into Python and run from the **Command Prompt** window. One caveat I will mention here is that permissions and where we install Python matter, so if your computer belongs to your employer, make sure that the Python program paths are adapted as needed so that you can access all modules and install libraries. The permissions may vary.

In my case, while my main computer does not belong to me, I have administrator access to it, so I can run my Command Prompt as an administrator and run `pip` from there.

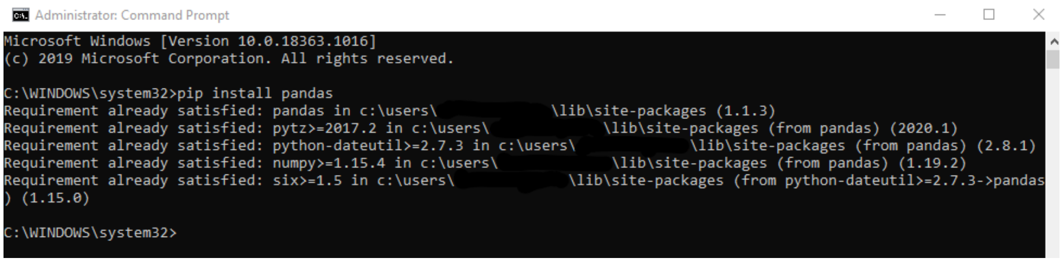
The following screenshot shows an installation of the `sympy` Python library. As you can see, using the `pip install sympy` command installs the library onto our system. For the record, `sympy` is a symbolic math library available for Python. Since I have the other libraries we'll be using already installed, I had to show the installation for a package I hadn't yet installed on my machine:



```
Microsoft Windows [Version 10.0.18363.1016]
(c) 2019 Microsoft Corporation. All rights reserved.

C:\WINDOWS\system32>pip install sympy
Collecting sympy
  Downloading sympy-1.6.2-py3-none-any.whl (5.8 MB)
     |████████████████████████████████| 5.8 MB 1.7 MB/s
Collecting mpmath>=0.19
  Downloading mpmath-1.1.0.tar.gz (512 kB)
     |████████████████████████████████| 512 kB 3.3 MB/s
Using legacy 'setup.py install' for mpmath, since package 'wheel' is not installed.
Installing collected packages: mpmath, sympy
    Running setup.py install for mpmath ... done
Successfully installed mpmath-1.1.0 sympy-1.6.2

C:\WINDOWS\system32>
```

Figure 12.1 – Installing a Python library

If you're trying to install a library that you've already installed—for example, if I tried to install pandas again—you'll get a **Requirement already satisfied** message, such as the one shown in the following screenshot. Notice that the user information will be filled in with your user, not blacked out as shown in the screenshot, and this also shows the location where the library package can be found on your system:



```
Microsoft Windows [Version 10.0.18363.1016]
(c) 2019 Microsoft Corporation. All rights reserved.

C:\WINDOWS\system32>pip install pandas
Requirement already satisfied: pandas in c:\users\[redacted]\lib\site-packages (1.1.3)
Requirement already satisfied: pytz>=2017.2 in c:\users\[redacted]\lib\site-packages (from pandas) (2020.1)
Requirement already satisfied: python-dateutil>=2.7.3 in c:\users\[redacted]\lib\site-packages (from pandas) (2.8.1)
Requirement already satisfied: numpy>=1.15.4 in c:\users\[redacted]\lib\site-packages (from pandas) (1.19.2)
Requirement already satisfied: six>=1.5 in c:\users\[redacted]\lib\site-packages (from python-dateutil>=2.7.3->pandas
) (1.15.0)

C:\WINDOWS\system32>
```

Figure 12.2 – Requirement already satisfied message

You may also wish to use Anaconda, which is an open source distribution of the Python and R programming languages. When Anaconda is installed, you can use the `CMD.exe` prompt window to install your libraries using `conda install` or `pip install`. The Jupyter notebook included in the Anaconda navigator can run and save your Python programs.

There are more programs and packages included in the Anaconda distribution program and they can simplify how we interact with the Python programming language. It's a great resource if you are doing extensive coding.

If you haven't done so already, this would be a good time to install the NumPy, pandas, and Matplotlib libraries before we start using them in analyzing and displaying data and creating models.

Using NumPy and pandas

NumPy, like so many of the libraries in Python and the Python programming language itself, is an open source library. NumPy is used for multi-dimensional arrays and matrix data structures. Python itself doesn't have arrays; it has lists. So, libraries can be used to provide that capability for our algorithms. When we have multiple elements of the same type, we can use a data structure to save them – that is, an **array**.

The `pandas` library is used to analyze data and is built on the `numpy` package. The pandas and NumPy libraries are often used together. When we need graphical models, we add a third library, Matplotlib, or another similar library. In this chapter, we're going to stick with Matplotlib.

When we import libraries, we can import them as the whole name, `numpy` in this case, or we can shorten them for ease of use. If we want to import the library, we can use `import numpy`. Let's say we wanted to create an array of the numbers from `0` to `11`. We can use `numpy` to organize that by combining the `arange` and `reshape` functions. Let's take a look at the code snippet:

ch12_abbreviate.py

```
import numpy as np
myArray = np.arange(0, 12).reshape(2, 6)
print(myArray)
```

Notice that we imported `numpy as np` instead of just `numpy`. That means that I can now call the NumPy functions using `np` instead of having to type `numpy` each time.

Tip:

Please note that `np` is a standard abbreviation of NumPy, so you may see it often. You can import NumPy as anything, any name, but `np` is the standard convention.

In the preceding snippet, we are asking the algorithm to split the list of numbers from `0` to `11` into `2` rows of `6` elements each. Then, we print the array to see our result. Take a look at the output:

```
[[ 0  1  2  3  4  5]
 [ 6  7  8  9 10 11]]
```

As you can see, we start at `0`, which is our lower bound in the range `0` to `12`. We have two rows now, with six numbers in each of the two rows. *We do not include 12.* If we had tried to do `0` to `13`, we wouldn't be able to reshape our array because we cannot evenly split 13 numbers. We would have gotten the following traceback error (most recent call last):

```
  File "C:/Users/… /ch12_abbreviate.py", line 3, in <module>
    myArray = np.arange(0, 13).reshape(2, 6)
ValueError: cannot reshape array of size 13 into shape (2,6)
```

Recall that your information will be different where you see the ellipses (…). It will have your file location, or path, based on how your Python is set up. For example, your file location may be something like `C:/Users/JohnSmith/Documents/ch12_abbreviate.py`. This should replace the file location in the preceding code.

As you can see from the `ValueError` message, we cannot reshape an array into the `(2, 6)` shape because we'd have one number left over.

Now, let's take a look at a pandas `DataFrame`. We use `DataFrames` so that we can manipulate our data in rows and columns. Let's take a look at an example:

ch12_pdDataFrame.py

```
import pandas as pd

myAddressBook = {'names': ['Susan', 'Malena', 'Isabel',
                           'Juan', 'Mike'],
                 'phoneNumbers':['333-333-3333',
                                 '444-444-4444',
                                 '555-555-5555',
                                 '777-777-7777',
                                 '888-888-8888']}
addressBook = pd.DataFrame(myAddressBook)
print(addressBook)
```

Notice that we created a dictionary with value pairs for the names and numbers of our contacts. After doing so, we saved our address book as `DataFrame`, which will organize our information in tabular form. Finally, we printed our address book. Take a look at the output:

```
     names  phoneNumbers
0    Susan  333-333-3333
1   Malena  444-444-4444
2   Isabel  555-555-5555
3     Juan  777-777-7777
4     Mike  888-888-8888
```

Notice that pandas used the information from the dictionary to create the labels for our columns. We did not give it information for labeling our rows, so the numbers `0` to `4` were automatically used by Python and pandas. The algorithm resulted in a table that provides the names and numbers in our address book.

As you can see, pandas and NumPy simply add more capabilities to Python. Before we move on to Matplotlib, notice that we haven't really seen actual data analysis yet. We'll get to that soon. Right now, we know how to use our libraries and that we can use them to organize and analyze data. Let's talk quickly about Matplotlib and then move on to an example where we can use a data file to do some analysis.

Using Matplotlib

As we mentioned in the previous section, pandas and NumPy do not provide a visual representation or visual model for our data. In order to create graphs from our data, we can use the Matplotlib library in Python. Much like we imported the pandas and NumPy libraries as `pd` and `np`, respectively, we can also shorten Matplotlib. The full library contains multiple modules. We'll use the `pyplot` module a lot. The most commonly used abbreviation for `Matplotlib.pyplot` is `plt`. If we're calling Matplotlib, we usually abbreviate that as `mlt`. Let's take a look at a quick sample:

ch12_matplotlib.py

```
import matplotlib.pyplot as plt
plt.plot([0, 3, 6], [4, 5, 6])
plt.show()
```

Notice that this particular library has an even longer name than `matplotlib`. To import the library, we have to import it using `matplotlib.pyplot`. Imagine having to write all that every time we need it in an algorithm. *No thanks!* Instead, we import it as `plt`.

Now, let's take a look at the second line in the code snippet. We're creating a plot that matches each of the numbers in the first list to the numbers in the second list. So, we have three coordinate pairs: `(0, 4)`, `(3, 5)`, and `(6, 6)`. The output when using this code isn't just a line of output; it's a plot. Now, the algorithm creates that plot in the second line of the preceding code snippet.

But it won't show it to you unless you tell it that's what you want. Think of the `show()` function as our `print()` function in Python. Unless we tell the algorithm that we want to see something, the algorithm will perform the task in the background but won't show it to us. The following graph shows the output for this algorithm:

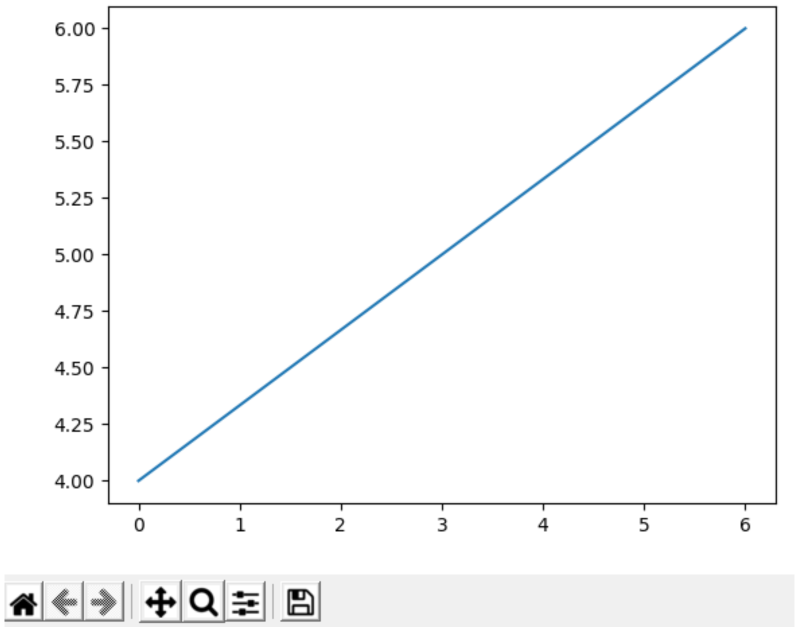
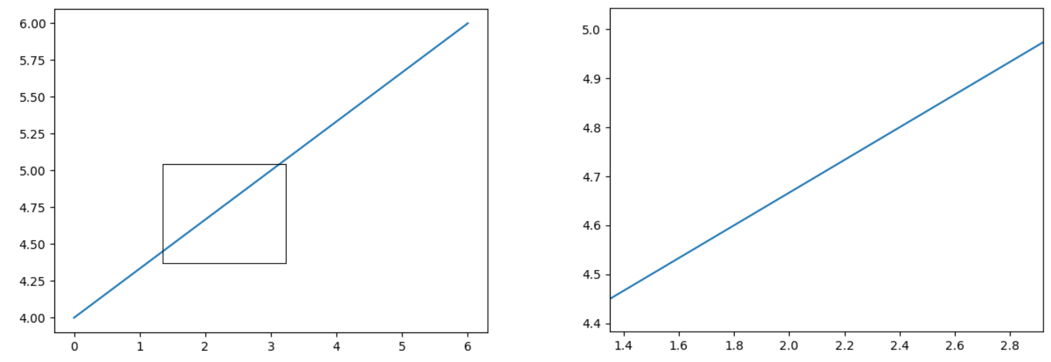


Figure 12.3 – Matplotlib sample graph

Now, as you can see, the graph itself is already helpful. We can see that this is a linear relationship defined by those points. But notice that the bottom of the screen gives you some options. The home icon at the bottom left will reset your graph at any time to its original window. The **Zoom** function, shown in the following screenshot, allows us to select a piece of the graph that we want to look at more closely:



Selection of the graph area before zooming   Selection of the graph area after zooming

Figure 12.4 – Zoom feature of Matplotlib

Notice that the left graph shows our selection for the piece of the graph we wanted to take a closer look at. Then, the graph on the right is only showing us the values we selected. Matplotlib also allows us to configure subplots and navigate with the arrows back and forth between multiple representations. For example, if I were to click on the left arrow on the graph after zooming, it would take me back to the previous representation. Clicking the **Home** button takes me back to the original graph.

Of course, this is not the only type of graph or representation Matplotlib allows. This is just a tiny glimpse into visual representations.

> **Tip:**
> An additional resource for the types of plots available is available here: `https://matplotlib.org/3.3.2/tutorials/introductory/sample_plots.html`.

In this section, we learned about the Matplotlib library. We learned how to create a simple plot and how we can zoom into sections of our graphs from the outputs when we create our graphics.

Now that we know how to access our libraries and what each can do for us, let's take a look at how we use them to analyze data.

Understanding data analysis with Python

In the previous section, we introduced some of the libraries that we can use to analyze data in Python. In this section, we will be looking at one example and multiple code snippets to build a bar graph using real data and Matplotlib, but before we do so, let's review why Python is so important with regard to data analysis.

As Python is object-oriented, it allows us to streamline really complex and/or large datasets. This allows great readability of the data and using the libraries can produce data representations such as tables and visual models that allow us to predict where our data is going, create regression analyses, and much more. As mentioned in the introduction of this chapter, data analysis is also critical for decision-making. A well-designed experiment produces data that we can rely on and that is generalizable. Data analysis can be a tool for more equality and equity in our society.

All that being said, we are going to look more at the mechanical aspects of what we can do with Python rather than how we interpret it so that we can understand how Python does data analysis and presents results using our libraries. That said, we'll use more examples in *Chapter 15*, *Applied Computational Thinking Problems*, and *Chapter 16*, *Advanced Applied Computational Thinking Problems*, in various areas that will not only use some of these tools but will also provide us with an opportunity to explore what data analysis means in the context of those samples.

It's time to take a look at an example that can help us further understand the capabilities of these tools and how we can write algorithms that solve some of the presented problems.

Before we begin, we'll be using a data file, `ch12_data.csv`, which can be found in the GitHub repository. This file contains graduation rates for degree-seeking students from 1996 through 2012 by race/ethnicity, time to completion, sex, control of institution, and the percentage of applications accepted. The data was downloaded from the **National Center for Education Statistics** here: `https://nces.ed.gov/programs/digest/d19/tables/dt19_326.10.asp`.

The data file in our repository only contains the data for all four-year institutions rather than the entire file. Some of the headings are also simplified.

When we want to work with a data file, it is important that we tell Python where it can locate it so that it knows what to use when running the algorithm. To do so, we use the `os` module, which allows our algorithm to interact with our **operating system** (**OS**). Notice that our snippet already includes the other libraries as well (we'll use them later):

ch12_csvAnalysis.py

```
import pandas as pd
import matplotlib as mlp
import matplotlib.pyplot as plt
import os

#Let Python know correct directory for file.
```

```
os.chdir('C:\\Users\\...\\Program Files\\Chapter 12')
print('Current directory', os.getcwd())
```

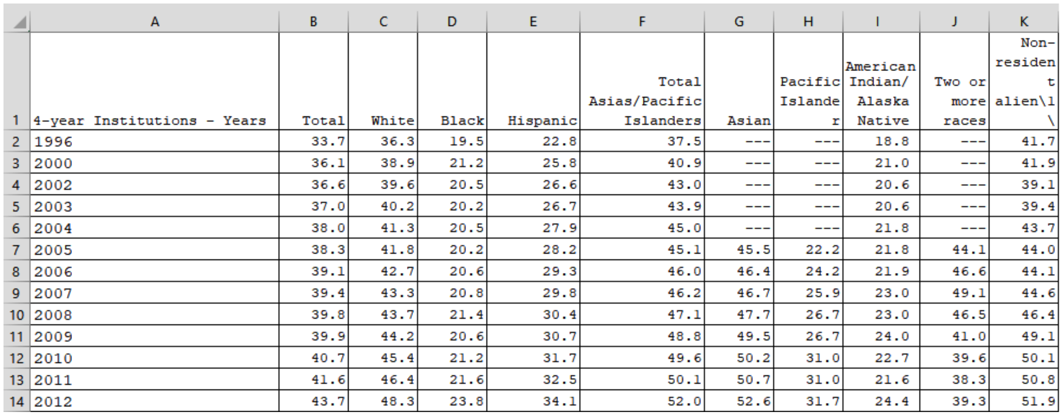
As we've done previously, make sure you replace the ellipses in the user information with your own location. In the algorithm provided in the repository, you'll also need to adjust that location in order to run this algorithm, as the location for that algorithm would be for my own path.

Now that we've told Python where to find the file, we can use it in the algorithm. If we run this algorithm, we'll see that the output should match the path we noted in the line that starts with `os.chdir`:

```
Current directory C:\Users\...\Program Files\Chapter 12
```

Notice again that our paths will not match. That will all depend on where you have your files saved.

The following figure shows the same data and file as our `.csv` file in `.xls` format, as it is easier to point out what we need from it. Note that we will use the `.csv` file for the analysis:



	A	B	C	D	E	F	G	H	I	J	K
1	4-year Institutions - Years	Total	White	Black	Hispanic	Total Asias/Pacific Islanders	Asian	Pacific Islande r	American Indian/ Alaska Native	Two or more races	Non-residen t alien\1 \
2	1996	33.7	36.3	19.5	22.8	37.5	---	---	18.8	---	41.7
3	2000	36.1	38.9	21.2	25.8	40.9	---	---	21.0	---	41.9
4	2002	36.6	39.6	20.5	26.6	43.0	---	---	20.6	---	39.1
5	2003	37.0	40.2	20.2	26.7	43.9	---	---	20.6	---	39.4
6	2004	38.0	41.3	20.5	27.9	45.0	---	---	21.8	---	43.7
7	2005	38.3	41.8	20.2	28.2	45.1	45.5	22.2	21.8	44.1	44.0
8	2006	39.1	42.7	20.6	29.3	46.0	46.4	24.2	21.9	46.6	44.1
9	2007	39.4	43.3	20.8	29.8	46.2	46.7	25.9	23.0	49.1	44.6
10	2008	39.8	43.7	21.4	30.4	47.1	47.7	26.7	23.0	46.5	46.4
11	2009	39.9	44.2	20.6	30.7	48.8	49.5	26.7	24.0	41.0	49.1
12	2010	40.7	45.4	21.2	31.7	49.6	50.2	31.0	22.7	39.6	50.1
13	2011	41.6	46.4	21.6	32.5	50.1	50.7	31.0	21.6	38.3	50.8
14	2012	43.7	48.3	23.8	34.1	52.0	52.6	31.7	24.4	39.3	51.9

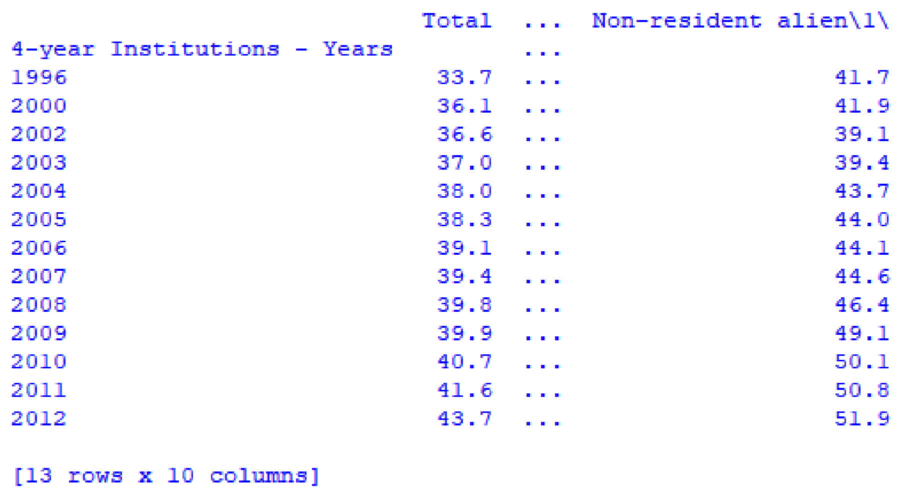
Figure 12.5 – Data to be used in Python formatted as an .xls file

If we only want to pull the rows shown, we can use the following code snippet to get that information from our `.csv` file:

ch12_csvAnalysis_2.py

```
import pandas as pd
graduates = pd.read_csv('ch12_data.csv', index_col = 0)

print(graduates[0: 16])
```

Notice that our algorithm imported the pandas library. The `read_csv()` function tells Python that we'll be using the filename and the index of the column we want to start working with. That index gives us the values we'll use as row headers. If the index wasn't in the first column, we could change that to a different value. Then, we print the rows we want to see. Because our data is wide, our output is shown as follows:



```
                             Total  ...  Non-resident alien\1\
4-year Institutions - Years         ...
1996                          33.7  ...                   41.7
2000                          36.1  ...                   41.9
2002                          36.6  ...                   39.1
2003                          37.0  ...                   39.4
2004                          38.0  ...                   43.7
2005                          38.3  ...                   44.0
2006                          39.1  ...                   44.1
2007                          39.4  ...                   44.6
2008                          39.8  ...                   46.4
2009                          39.9  ...                   49.1
2010                          40.7  ...                   50.1
2011                          41.6  ...                   50.8
2012                          43.7  ...                   51.9

[13 rows x 10 columns]
```

Figure 12.6 – Output of ch12_csvAnalysis.py algorithm

Notice that we are only seeing the first two columns, then we have ellipses. That shows us that there are more columns between the second column and the third shown. That doesn't mean that Python didn't read the rest of the information, it's just not visible to us. The last output line, `[13 rows x 10 columns]`, actually tells us how many rows and columns are in the data we pulled.

Let's say I wanted to see data for one group, say `Hispanic`, as a comparison using a bar chart for the years between 1996 and 2012. Then, I can use the **bar chart plot** from the Matplotlib library. Let's take a look at the adjustments made to the following algorithm:

ch12_csvAnalysis_3.py

```
import pandas as pd
import matplotlib as mlp
import matplotlib.pyplot as plt
import os

#Let Python know correct directory for file.
os.chdir('C:\\Users\\sofia.dejesus\\OneDrive - Hawken
    School\\03_book\\Program Files\\Chapter 12')

graduates = pd.read_csv('ch12_data.csv', index_col = 0)
print(graduates[0:13])

fig, ax = plt.subplots()
ax.bar(graduates.index, graduates['Hispanic'])
ax.set_xticks(graduates.index)
ax.set_xticklabels(graduates.index, rotation = 60,
    horizontalalignment = 'right')
ax.set_title('Number of Hispanic graduates from 1996-2012',
    fontsize = 16)
ax.set_ylabel('Number of Graduates')

plt.show()
```

There are some things we should look at closely from our preceding code. The first thing we did was add the `fig, ax = plt.subplots()` line. This line allows us to create the plot, but also allows us to create multiple plots in a figure if we wanted to. If we were adding four plots, we'd use `fig, ax = plt.subplots(2, 2)`, which would tell the algorithm that we're creating four plots in two rows and two columns. If we leave the parentheses empty, as we did in the preceding code, we have just one subplot.

Next, we are identifying the figure we'll create, which is a bar graph. We want to just compare the numbers for the Hispanic population over the years, so we identify that in our `ax.bar(graduates.index, graduates['Hispanic'])` line.

I should note here that many developers use `dataframe` as their variable. I prefer to name my `DataFrame` descriptively instead, so instead of calling my `DataFrame` `df`, I've called it `graduates` in the algorithm. Whatever your preference is, this is the `DataFrame` that we are currently working with to create a readable plot.

Notice the ticks and tick labels in the code; we first identify where we'll get the data from for the ticks and what the labels will be. Then, we can add more formatting to our plot by adding rotation (if we want to tilt our labels), alignment, and so on. We could also change the font size here. Finally, we set the y -axis title and the bar graph title before we show the plot. The following screenshot shows the plot for this algorithm:

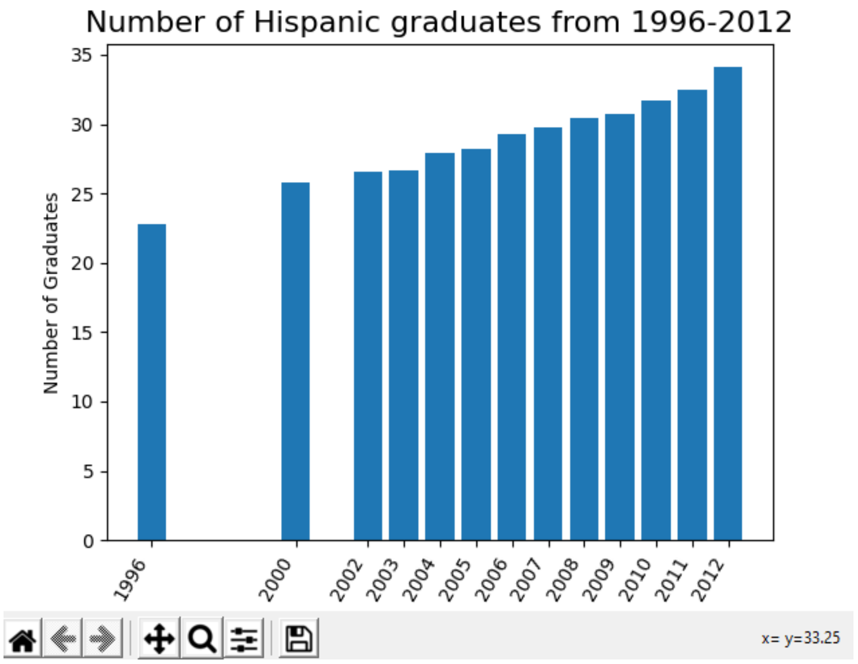


Figure 12.7 – Plot of Hispanic graduates for ch12_csvAnalysis_3.py

Notice that our graph has labels and shows the data clearly, and we were able to modify our formatting so that it was legible. As we analyze the information, we also notice the gap between the first year and the year **2000**, and between **2000** and **2002**. The data for those years is not included in the data file.

This is only one of the plots possible using Matplotlib. We'll have a chance to explore a few more in *Chapter 15*, *Applied Computational Thinking Problems*, and *Chapter 16*, *Advanced Applied Computational Thinking Problems*, which are solely dedicated to samples in multiple areas using everything we've discussed throughout earlier chapters of this book. For now, let's move on to some other applications of data and Python libraries.

Using additional libraries for plotting and analysis

Before we end this chapter on experimental data, the use of libraries, and plotting and analyzing data, let's look at three more libraries that are helpful in data analysis and plotting. These are not the only libraries for analysis and plotting, nor will they be the only ones we explore throughout the rest of this book:

- **Seaborn** is a library used for data visualization; built on top of Matplotlib.
- **SciPy** is a library used for linear algebra, optimization, statistics, and more; built on top of NumPy.
- **Scikit-Learn** is a library used in machine learning; part of the SciPy stack.

In the following chapters, we'll go deeper into the use of some of these libraries as we tackle some of the application problems that require their use. For now, let's take a quick look at what each of these libraries can help us with when looking at datasets.

Using the Seaborn library

The Seaborn library provides us with more features on top of the Matplotlib visualization features. There are many things we can do with the Seaborn library, which we usually import as `sns` for simplification of the code. Here are some of the common uses of the library:

- Correlations
- Aggregate statistics (observation of categorical values)
- Linear regression plots for dependent variables
- Creating abstractions and grids with multiple plots

One of the greatest things about Seaborn is that it also works well with pandas. Creating statistical representations of data and visualizations is easy when combining with pandas DataFrames.

Seaborn has some sample datasets that can be accessed – that is, they are built-in – if we know the name of the dataset. We can then call the built-in dataset with a simple line of code. Let's take a look at the following snippet of code and resulting graphics:

ch12_seabornSample.py

```
# Import seaborn
import seaborn as sns
sns.set_style('darkgrid')
# Load an example dataset
exercise = sns.load_dataset("exercise")
#Create the plot
sns.relplot(
    data=exercise,
    x='id', y="pulse", col="time",
)
```

From the preceding code, you can see that we've set a style for our plots. We are adding a `'darkgrid'` style to our plots and called it after importing the library. Seaborn has a few built-in styles: `white`, `whitegrid`, `dark`, `darkgrid`, and `ticks`. The following screenshot shows the resulting plot from Seaborn:

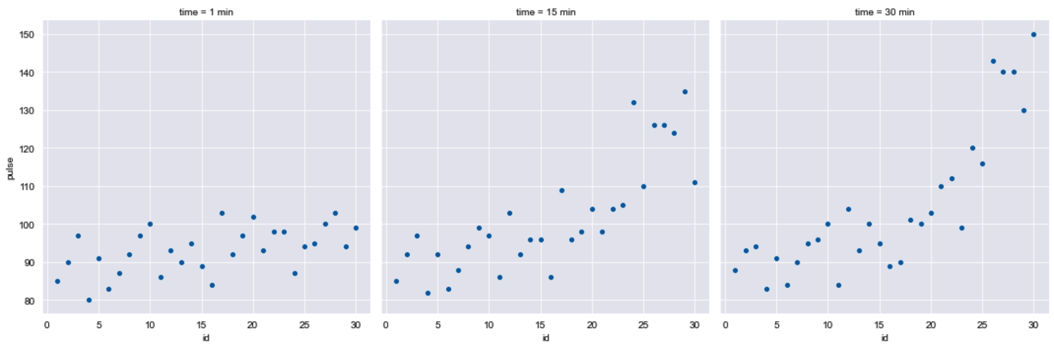


Figure 12.8 – Graph of exercise sample dataset with darkgrid style

As you can see from the graphs, we would be able to analyze whether or not there are correlations between pulse and time or any other variables.

We can also use pair plotting to show whether there are correlations between the variables. Let's use another built-in dataset, flights, to see what the pair plotting does:

ch12_pairplotSNS.py

```
# Import seaborn
import seaborn as sns

sns.set_style('darkgrid')
# Load an example dataset
flights = sns.load_dataset("flights")

#Create the plot
sns.pairplot(
    data=flights,
)
```

The preceding snippet of code is very similar to the previous one for the exercise data. The difference is that we call the `pairplot()` function in this case. The following screenshot shows our resulting grid of plots:

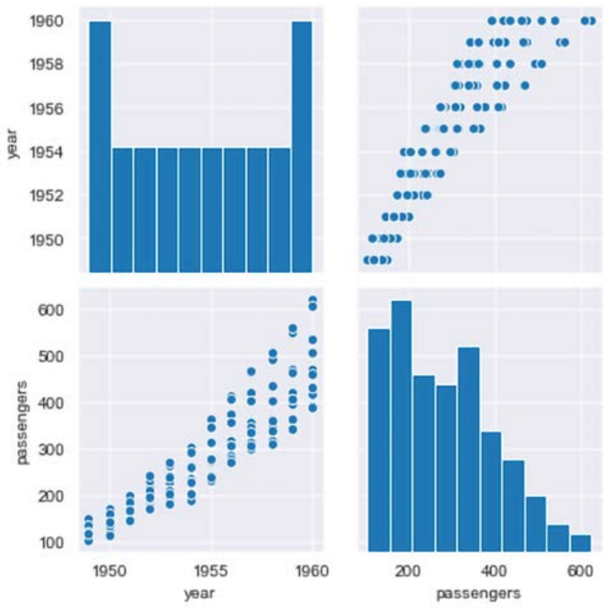


Figure 12.9 – Pair plotting using Seaborn

Notice that the number of passengers and the years seem to be positively correlated. That is, there are more passengers in later years than in the early years of flight. When analyzing this dataset, we would probably want to make a prediction, such as stating that more passengers would be flying in modern times.

We could use the graphs to help us make predictions. That said, this data is fairly old, so we'd need more updated data to make accurate predictions. The more data we have, the better. For massively large amounts of data, we can also use machine learning, which we will briefly explore in *Chapter 13*, *Using Classification and Clusters*, and *Chapter 14*, *Computational Thinking and Python in Statistical Analysis*, so that we can learn how to manipulate and learn from the data.

Before we move on to another library, one more thing about the pair plots—if there are 10 columns of data, the pair plot will compare each column with itself, then every other column available, creating a pretty large display with all the compared variables. We will look at some more pair plotting in *Chapter 14*, *Computational Thinking and Python in Statistical Analysis*. There are also problems in *Chapter 15*, *Applied Computational Thinking Problems*, and *Chapter 16*, *Advanced Applied Computational Thinking Problems*, that will allow us to practice more with those graphs and what we can get from them.

The Seaborn library is helpful to visualize statistical data, much like Matplotlib. The ease of some of the built-in functions in Seaborn makes it a great tool for visualization and analysis.

Now, let's look at the SciPy library.

Using the SciPy library

The SciPy library is mostly used to solve scientific and mathematical problems. Some of the helpful sub-packages and their uses are as follows:

- `cluster` is used for clustering algorithms.
- `constants` contains physical and mathematical constants and units, such as `golden` (the golden ratio) and `mu_0` (the magnetic constant).
- `fftpack` utilizes fast Fourier transform routines.
- `integrate` is used for differential equation solvers.
- `interpolate` contains interpolation and smoothing splines.
- `io` relates to input and output.
- `linalg` is associated with linear algebra.
- `ndimage` is used for processing *N*-dimensional images.

- `odr` is used for orthogonal distance regression.
- `optimize` is used for optimization and root-finding routines.
- `signal` is used for signal processing.
- `sparse` is used for sparse matrices and associated routines.
- `spatial` is used for spatial data structures and algorithms.
- `special` is used for special functions (such as elliptic functions and integrals).
- `stats` is used for statistical distributions and functions.

Within each of the sub-packages, SciPy contains many functions to aid in the visualization and optimization of scientific data. Because it was created specifically with that purpose, it is a common tool used in the scientific area. That said, the statistical packages are also robust, so the library is helpful to use even in non-scientific statistical analyses.

Let's now look at the Scikit-Learn library, which we'll use in the following chapters of this book.

Using the Scikit-Learn library

Scikit-Learn is probably the most important library for machine learning available in Python. We will explore this library in samples in the following chapters as we explore some of the problems that would be suitable for machine learning, so we won't go too deep into the functionalities here. That said, here are some of the functionalities that Scikit-Learn gives us:

- **Clustering** helps with grouping data that is unlabeled.
- **Regression** measures the relationship between the variable (the mean of the variable) and the values of the other variables.
- **Classification** has multiple classifiers within Scikit-Learn, similar to regression. Some of the classifiers are **linear discrimination analysis**, **bagging classifiers**, **K-nearest neighbors classifiers**, and more.
- **Model selection** has tools for creating training and testing models in machine learning.
- **Preprocessing** contains tools to standardize the dataset (more details on data preprocessing can be found in *Chapter 14, Computational Thinking and Python in Statistical Analysis).*

The Scikit-Learn library is something we will become fairly familiar with in the upcoming examples in *Chapter 14*, *Computational Thinking and Python in Statistical Analysis*, *Chapter 15*, *Applied Computational Thinking Problems*, and *Chapter 16*, *Advanced Applied Computational Thinking Problems*, so we'll leave some of that discussion for those chapters.

The libraries and packages that we have available in Python allow us to perform detailed analysis for our dataset and create a wide array of useful plots that aid in data analysis.

Summary

In this chapter, we went over the definitions of experimental data and validity, reliability, and generalizability in the context of experiments. We also discussed how to install and use the pandas, NumPy, and Matplotlib libraries so that we could use them to organize and display data. Some of the skills you learned include defining an experiment, data gathering, and how computational thinking helps us define problems and design what we'd use to display our results.

In addition, we learned about data analysis and data science and its growth and importance in our current world. We were able to use the libraries to produce a plot that represented a subset of a data file using a Matplotlib bar chart.

In the next chapter, we'll be learning more about data and other applications of data science and data analysis.

13 Using Classification and Clusters

In this chapter, we will use the classification and clustering capabilities of the Python programming language. We will use the computational thinking elements to define the necessary components for problems when working with clusters and classification.

In this chapter, we will cover the following topics:

- Defining training and testing
- Implementing data clustering

By the end of this chapter, you will be able to design algorithms that are the best fit for the scenarios presented. You will also be able to identify the Python functions that are the most aligned with the problems presented and generalize your solutions.

Technical requirements

We will need the latest version of Python and **Scikit-Learn** to execute the code in this chapter.

You will find the code used in this chapter here: `https://github.com/PacktPublishing/Applied-Computational-Thinking-with-Python/tree/master/Chapter13`

You will find the **Pima Indians Diabetes Database** from **Kaggle** here: `https://www.kaggle.com/uciml/pima-indians-diabetes-database`

Data training and testing

In this section, we're going to learn how to create models for training and testing data using Python tools and libraries. When working with data and data science, we sometimes want to train the algorithm to continue gathering and learning from the data. Data training is then used for data mining and machine learning. First, let's define the **training dataset**:

- It is a sample of data used to fit the model.
- It is an actual dataset that is used to train the model (weights and biases, in the case of a **neural network**). The training model *sees* and *learns* from this data.

In computing, a neural network is a system of computing that is created using biological neural networks in human and animal brains as inspiration. When using training datasets, such as when we are creating machine learning models, the models depend heavily on the data. *But what is machine learning?* **Machine learning**, or **ML**, is an application of **artificial intelligence** (**AI**) that allows a machine to learn automatically using a program without explicitly being programmed to do so.

Without a top-quality foundation of data training, an algorithm is useless. Data training in ML refers to the initial data used to develop models. They find relationships and develop understanding and discover patterns and trends. Input data is fed into the ML algorithm and also to all the techniques associated to produce an output. That output is also fed back into the models as updated feedback, which in turn provides feedback data that is used as input again. The process is cyclical, continuously adapting and learning. The following diagram shows a simple ML graphic with the processes that happen:

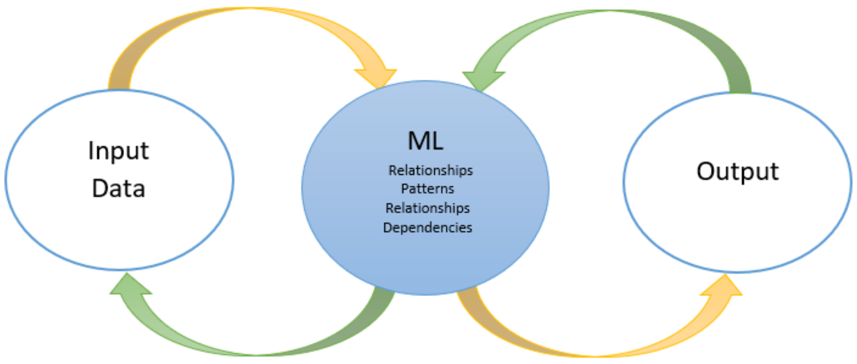


Figure 13.1 – Machine Learning and its relationships with input and output

In ML, data is combined with statistical tools to predict an output. The machines receive input data and use an algorithm to construct answers. This is similar to **data mining**, where large datasets are used to find anomalies, correlations, patterns, and so on to predict an outcome.

In data mining, we extract information but using methods that pull the necessary, relevant, error-free data points. That is, with data mining, we extract what we need from the dataset without extracting those anomalies while also looking at correlations and patterns within our data. The difference between data mining and ML is that ML analyzes both input data and output data. So once the output is processed, it goes back to the algorithm, where it's fed back to the input data, and reprocessed. The cycle is ongoing, as you can see from the preceding diagram.

Important Note:

There are four groups of ML algorithms, but in this book, we are only going to introduce two. While this is not an ML book, Python's applications in the ML arena continue to grow, so it is relevant to our goal of understanding the applications of the programming language and how we use computational thinking to solve problems.

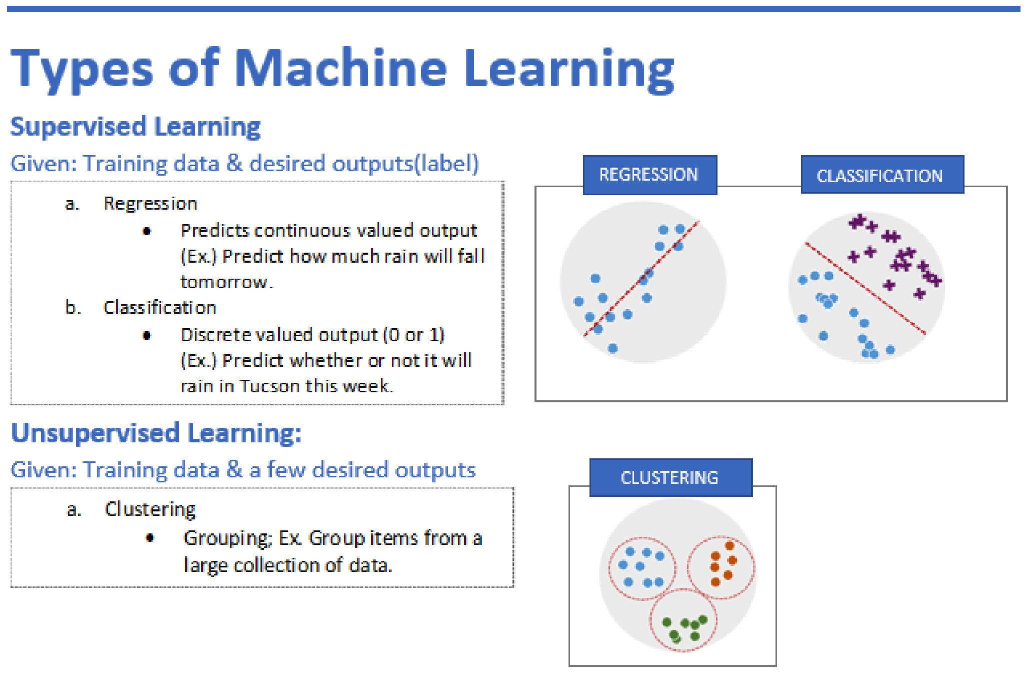
In ML, we use two types of important techniques:

- **Supervised learning** maps data pairs, using input data and expected output (training data) so that the model can find underlying patterns.
- **Unsupervised learning** uses unlabeled training data to make conclusions.

In addition, there are two other techniques we won't go into in this book: semi-supervised learning and reinforcement learning.

In supervised learning, the learning algorithm is presented with a set of inputs along with their desired outputs (also called labels). The goal is to discover a rule that enables the computer to re-create the outputs, or in other words, map the input and output. On the other hand, unsupervised learning allows us to approach problems with little or no idea what our results should look like. The output variables are unlabeled. With unsupervised learning, an algorithm is presented with a set of inputs but no desired outputs, which means the algorithm must find structures and patterns on its own.

The following diagram shows a roadmap to follow for supervised and unsupervised learning:



Types of Machine Learning

Supervised Learning

Given: Training data & desired outputs(label)

a. Regression
   - Predicts continuous valued output (Ex.) Predict how much rain will fall tomorrow.

b. Classification
   - Discrete valued output (0 or 1) (Ex.) Predict whether or not it will rain in Tucson this week.

Unsupervised Learning:

Given: Training data & a few desired outputs

a. Clustering
   - Grouping; Ex. Group items from a large collection of data.

Figure 13.2 – Types of machine learning

As you can see from the preceding figure, we have two types of **supervised learning**. When we are given the training data and desired outputs, we use **regression** or **classification**. With **regression**, we predict a continuous-valued output. With **classification**, we get a discrete-valued output (0 or 1). An example of regression would be predicting how much rainfall we'd get on a given day, while with classification, we'd be looking to know whether it would rain or not.

For **unsupervised learning**, the preceding figure contains the example of **clustering**. In **clustering**, we get the training data, but only a few desired outputs. An example of clustering is **grouping**, which would take items from large collections of data and group them.

We can apply these types of learning styles to **artificial neural networks**. The training of a neural network is usually conducted by determining the difference between the processed output of the network (often a prediction) and a target output. This is the error, so the network then adjusts its weighted associations according to a learning rule and using this error value to adapt. Neural networks are typically organized in layers. Layers are made up of several interconnected nodes that contain an activation function.

An activation function is used to make decisions about whether a neuron is activated or not. To do so, a weighted sum is calculated and further bias is added. We use activation functions to provide outputs of a neuron with non-linearity. The three most common activation functions are as follows:

- **Sigmoid** can be formulated as $\frac{1}{1 + e^{-x}}$, where the input of the real number is *x*. It returns a value between –1 and 1.
- **Tanh** is given by *tanh(x)*. It is a hyperbolic tangent function with real number input *x*; it returns a value between –1 and 1.
- **Rectified Linear Unit** (**ReLU**) is a piecewise linear function. Its output is the same as the input if the input is positive or 0 otherwise.

The following diagram shows the graphs associated with each of the aforementioned activation functions:

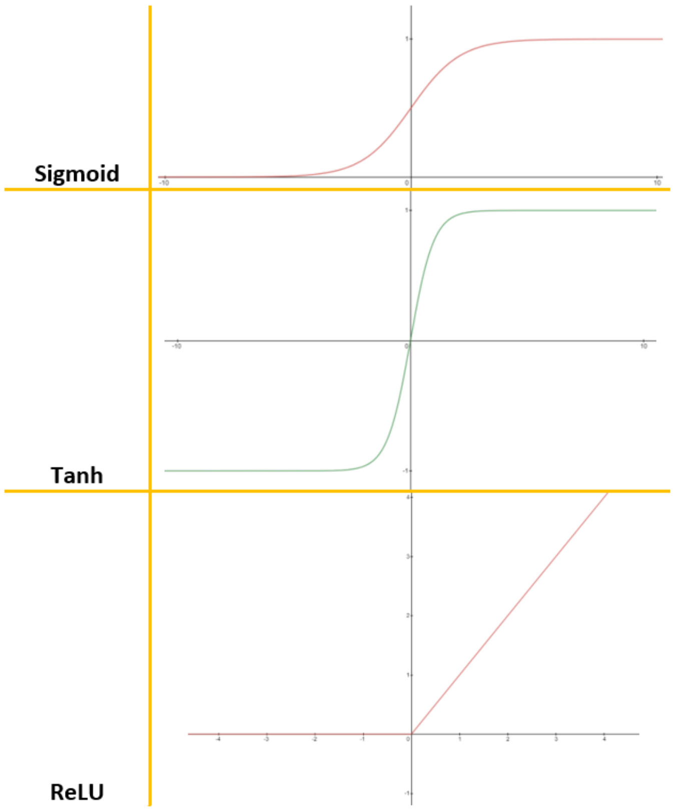


Figure 13.3 – Activation functions

Patterns are presented to the network via the **input layer**, which communicates to one or more **hidden layers**, where the actual processing is done via a system of weighted **connections**. The hidden layers then link to an **output layer** where the answer is output as shown in the following diagram:

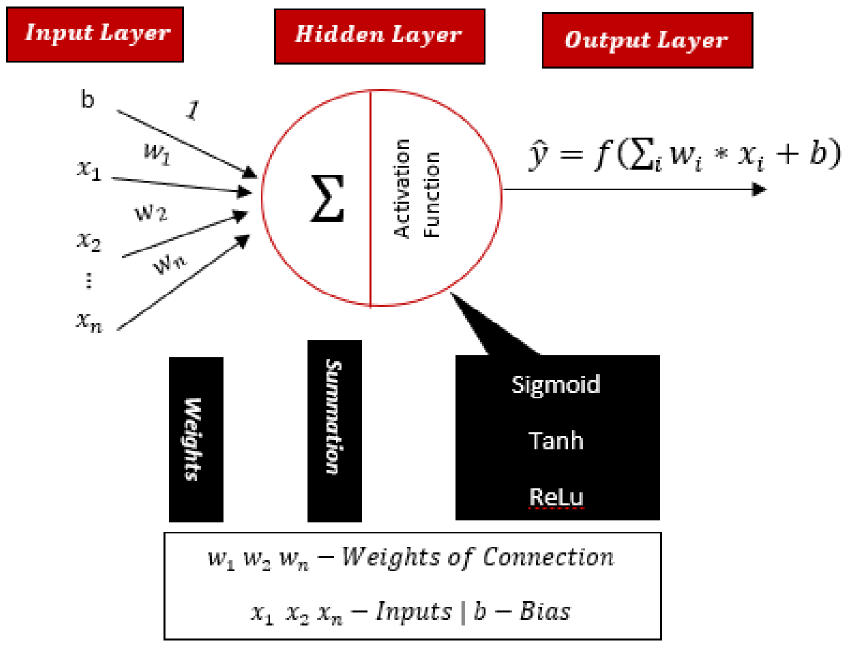


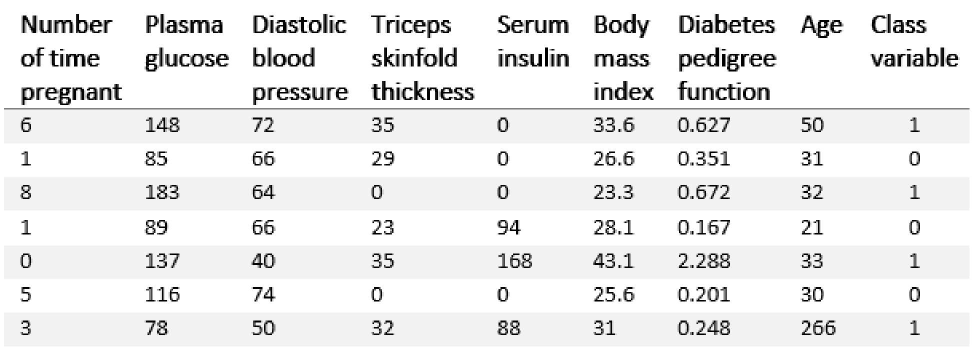
Figure 13.4 – Artificial Neural Network (ANN) model

As you can see, the summation operator (Σ) takes the input values and moves through the network and creates an output. They must be summed and return a single value when entering a new node. An activation function essentially *squashes* the input and transforms it into an output value that represents how much a node should contribute (that is, when a node should fire). A node is considered to be *fired up* when it is activated. It takes the output value and converts it so the next node can take it as input. This is called **firing up**.

Now let's take a look at how we classify data using pandas.

Classifying data example

Now let's take a look at an example where we are classifying data. The following screenshot shows an example of using supervised learning. To produce the output that can be seen in the screenshot, we used an existing dataset from `www.kaggle.com`. The dataset is called **Pima Indians Diabetes Database**. It describes whether or not a Pima Indian patient was diagnosed with diabetes:



Number of time pregnant	Plasma glucose	Diastolic blood pressure	Triceps skinfold thickness	Serum insulin	Body mass index	Diabetes pedigree function	Age	Class variable
6	148	72	35	0	33.6	0.627	50	1
1	85	66	29	0	26.6	0.351	31	0
8	183	64	0	0	23.3	0.672	32	1
1	89	66	23	94	28.1	0.167	21	0
0	137	40	35	168	43.1	2.288	33	1
5	116	74	0	0	25.6	0.201	30	0
3	78	50	32	88	31	0.248	266	1

Figure 13.5 – Sample of unsupervised learning

As you can see, the attributes from the table, also known as the *input variables* (x), are as follows:

- Number of times pregnant
- Plasma glucose concentration for 2 hours in an oral glucose tolerance test
- Diastolic blood pressure (mm Hg)
- Triceps skinfold thickness (mm)
- 2-hour serum insulin (mu U/ml)
- Body mass index ($\frac{\text{weight in kg}}{\text{height in m}^2}$)
- Diabetes pedigree function
- Age (years)

For the *output variable* (y), we have the class variable (0 or 1). From the dataset, each row represents a patient and whether or not that person received a diagnosis of diabetes in the past 5 years. As you can see, there are eight input variables and one output variable (the last column) as shown in the preceding figure.

We will be using the binary classification model (1 or 0), which maps the rows of input variables (x) to the output variable (y). This will summarize $y = f(x)$. The following code snippet uses this information to get our outputs. Please note we will be discussing the full file in snippets throughout this example:

ch13_diabetesA.py

```
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np

dataset = pd.read_csv('diabetes.csv')
```

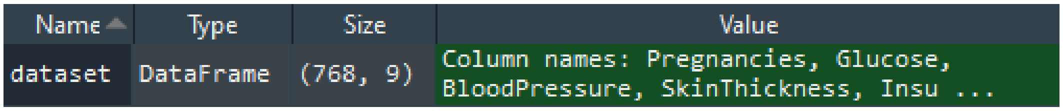
As you can see from the preceding snippet, we are uploading the dataset called `diabetes.csv` (from Kaggle). If you need a reminder of how to save the file and locate the path needed, take a look at *Chapter 12*, *Using Python in Experimental and Data Analysis Problems*, in the *Understanding data analysis with Python* section. There are many ways to upload datasets.

As we did in *Chapter 12*, *Using Python in Experimental and Data Analysis Problems*, we are using the very popular **pandas** and importing it as `pd`. Pandas is used for data manipulation and analysis. It offers data structures and operations for manipulating numerical tables and time series. The `read_csv()` function from pandas deals with importing data from **Comma-Separated Values** (**CSV**) values.

Important Note:

We need to find the correct directory. When you're calling the `.csv` file make sure you're in the correct directory (where the `.csv` file is located) to avoid error codes. Use `os.chdir()` after using `import os`, then use `print('Current directory', os.getcwd())`. See *Chapter 12, Using Python in Experimental and Data Analysis Problems*, for more information.

Once you run the preceding snippet of code, you can look at your variable explorer to see the item shown in the following screenshot. Note that the **variable explorer** is a tool that allows you to browse and manage objects associated with and used in your code. This tool is part of the **Spyder** environment, which runs Python with additional functionalities and editing tools, such as the variable explorer. The variable explorer is found in Spyder at the top on the right-hand side of our environment. The following screenshot shows the view of our database in the variable explorer:



Name	Type	Size	Value
dataset	DataFrame	(768, 9)	Column names: Pregnancies, Glucose, BloodPressure, SkinThickness, Insu ...

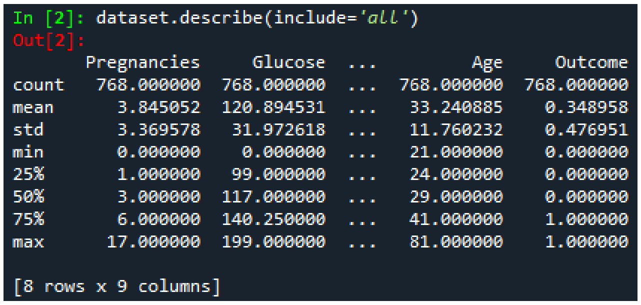
Figure 13.6 – Variable explorer sample

As you can see, **Size** describes the dataset. It shows the number of patients, **786**, and the total number of variables, **9**. Now we have a better understanding of our dataset.

But let's say you don't know what type of learning you will be needing. You can type this function in the console to get a full picture of the data and outputs:

```
dataset.describe(include='all')
```

The following screenshot shows the information we receive after using the preceding line of code in our algorithm:



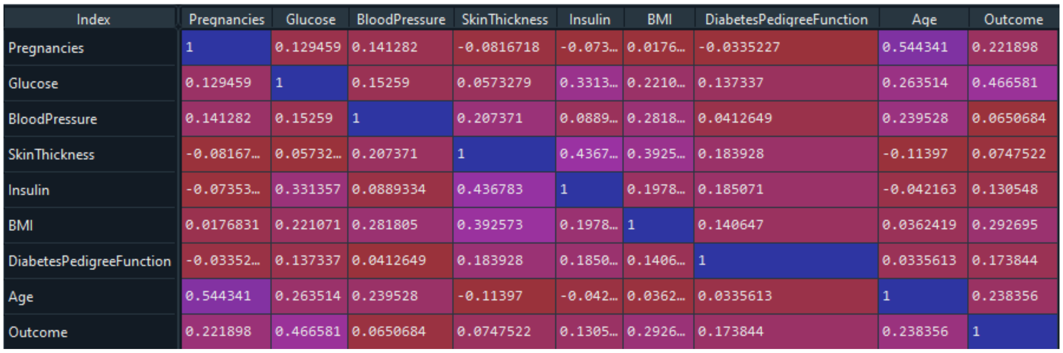
```
In [2]: dataset.describe(include='all')
Out[2]:
       Pregnancies     Glucose  ...         Age     Outcome
count   768.000000  768.000000  ...  768.000000  768.000000
mean      3.845052  120.894531  ...   33.240885    0.348958
std       3.369578   31.972618  ...   11.760232    0.476951
min       0.000000    0.000000  ...   21.000000    0.000000
25%       1.000000   99.000000  ...   24.000000    0.000000
50%       3.000000  117.000000  ...   29.000000    0.000000
75%       6.000000  140.250000  ...   41.000000    1.000000
max      17.000000  199.000000  ...   81.000000    1.000000

[8 rows x 9 columns]
```

Figure 13.7 – Display of information after running a description algorithm

As you can see from the preceding figure, we are able to get all the numerical features and know that there is no categorical data. We want that information, so the following line of code can be used to see the correlation between the variables:

```
dataset.corr()
```

This simple line of code helps us get the information shown in the following screenshot. Please note that the following screenshot may look different depending on the environment you are using. When running this code, using environments like **Spyder** or **Jupyter**, depending on your theme settings and choices, the table may look different, with different color schemes (or no color schemes):



Index	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age	Outcome
Pregnancies	1	0.129459	0.141282	-0.0816718	-0.073…	0.0176…	-0.0335227	0.544341	0.221898
Glucose	0.129459	1	0.15259	0.0573279	0.3313…	0.2210…	0.137337	0.263514	0.466581
BloodPressure	0.141282	0.15259	1	0.207371	0.0889…	0.2818…	0.0412649	0.239528	0.0650684
SkinThickness	-0.08167…	0.05732…	0.207371	1	0.4367…	0.3925…	0.183928	-0.11397	0.0747522
Insulin	-0.07353…	0.331357	0.0889334	0.436783	1	0.1978…	0.185071	-0.042163	0.130548
BMI	0.0176831	0.221071	0.281805	0.392573	0.1978…	1	0.140647	0.0362419	0.292695
DiabetesPedigreeFunction	-0.03352…	0.137337	0.0412649	0.183928	0.1850…	0.1406…	1	0.0335613	0.173844
Age	0.544341	0.263514	0.239528	-0.11397	-0.042…	0.0362…	0.0335613	1	0.238356
Outcome	0.221898	0.466581	0.0650684	0.0747522	0.1305…	0.2926…	0.173844	0.238356	1

Figure 13.8 – Dataset correlation graphic

We can see the correlation between all the variables with the outcome (output (*y*)). The preceding screenshot shows us that plasma glucose has the strongest correlation with the outcome, and insulin has the lowest.

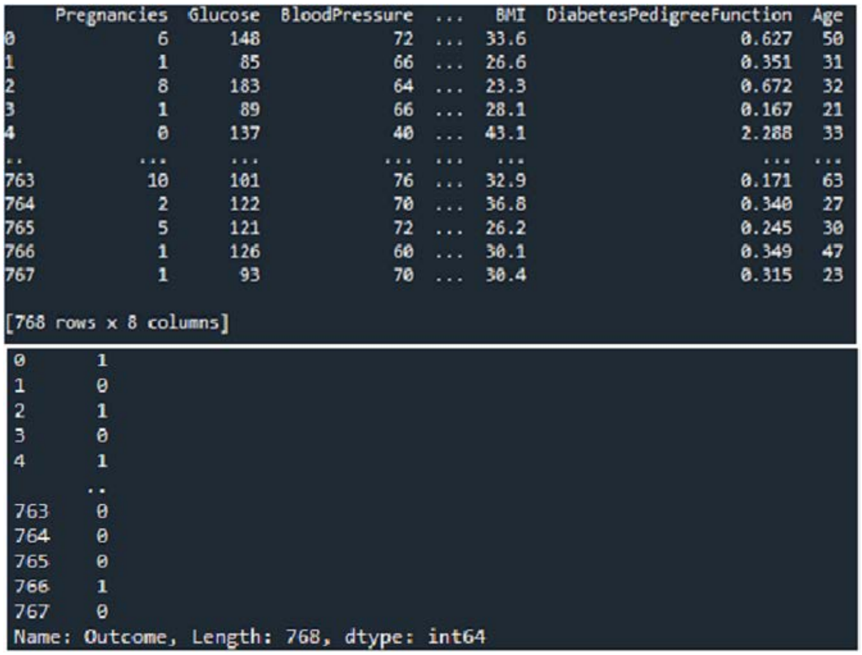
Now that we have a better understanding of the dataset, let's separate the input variables and output variables to put in the model. Let's take a look at the following code snippet from our `ch13_diabetesA.py` file, which exemplifies this for us:

```
#Split dataset into input(x) and output(y) variables
x_variables = dataset.iloc[:,0:8]
y_variable = dataset.iloc[:,8]
```

We use the `print` function to check our values:

```
print(x_variables)
print(y_variable)
```

Once you have run the preceding snippet of code, the output data will look as shown in the following screenshot. Note that the result shows what we've defined as the variables `x_variables` and `y_variable`, which are in turn defined as parts of the dataset, as noted in the preceding code:



```
     Pregnancies  Glucose  BloodPressure  ...   BMI  DiabetesPedigreeFunction  Age
0              6      148             72  ...  33.6                     0.627   50
1              1       85             66  ...  26.6                     0.351   31
2              8      183             64  ...  23.3                     0.672   32
3              1       89             66  ...  28.1                     0.167   21
4              0      137             40  ...  43.1                     2.288   33
..           ...      ...            ...  ...   ...                       ...  ...
763           10      101             76  ...  32.9                     0.171   63
764            2      122             70  ...  36.8                     0.340   27
765            5      121             72  ...  26.2                     0.245   30
766            1      126             60  ...  30.1                     0.349   47
767            1       93             70  ...  30.4                     0.315   23

[768 rows x 8 columns]
```

```
0      1
1      0
2      1
3      0
4      1
      ..
763    0
764    0
765    0
766    1
767    0
Name: Outcome, Length: 768, dtype: int64
```

Figure 13.9 – Training the dataset output for algorithm printing input and output values

Now we have to split the data into a training dataset and a test dataset. The purpose of the split technique is to evaluate the performance of an ML algorithm. It is only meant for any type of supervised learning algorithm. The first set (the training dataset) is used for the purpose of fitting the model.

The main goal is to fit it on available data with known inputs and outputs, then make predictions on new examples in the future where we do not have the expected output or target values.

Using the Scikit-Learn library

Another important library when working with data and ML is the `scikit-learn` (`sklearn`) library. This library is particularly useful for classification, regression, clustering, model selection, dimensionality reduction, and more. You may recall from *Chapter 12*, *Using Python in Experimental and Data Analysis Problems*, under the *Using data libraries in Python section*, that you can use `pip install` from your **Command Prompt** window to install the required libraries. Once you have the library, you can import it into the code, as shown in the following snippet, which uses `sklearn` to split the data. As a note, this snippet is part of the larger `ch13_diabetesA.py` file:

```
from sklearn.model_selection import train_test_split
X_train,X_test, y_train,y_test = train_test_split(
    x_variables, y_variable, test_size = 0.20,
    random_state = 10)
```

Here are the known parameters:

- `x_variable` and `y_variable` as previously defined.
- `test_size`: The test size will be 20% of the dataset.
- `random_state`: It sets a seed to the random generator so your train and test splits are always deterministic. If it is set to none, then a randomly initialized `RandomState` object is returned.

The following diagram shows the process and how each element interacts with others in the cycle:

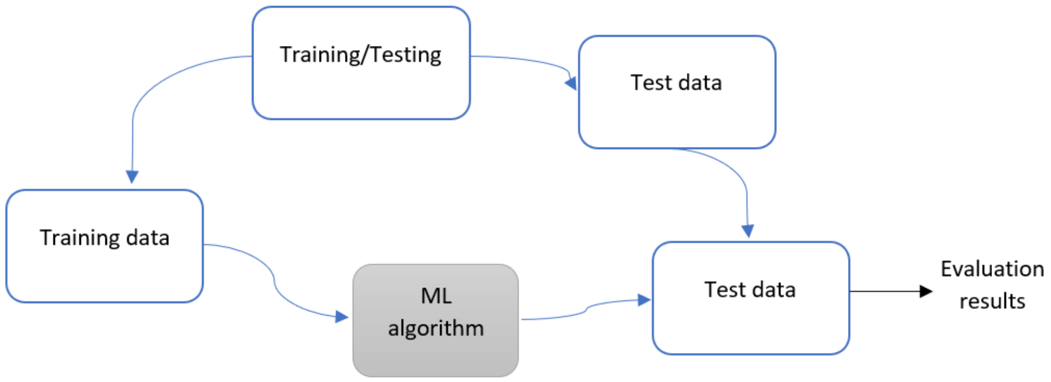


Figure 13.10 – Data cycle in ML

Note that we'll use a sequential model in the algorithm. In addition, we are using the `keras` library, which is used with Python so we can run deep learning models with our algorithms. Make sure you have the `keras` library available to use for this algorithm. If you have TensorFlow installed, you should already have access to the Keras library.

When working with machine learning problems and algorithms, you'll have a choice of libraries. We chose Keras for this particular problem. Keras is open source, and is useful in creating artificial neural networks. TensorFlow is a platform that contains many machine learning components and tasks.

Keras works on top of TensorFlow and makes it easier to interact with it in the Python programming language. Keras is a higher-level API, which we'll discuss further. Because of its capacity, it can sometimes be slower than usual. **PyTorch** is another library used for artificial neural networks. It's a lower-level API, and therefore runs faster. Keras is supported by **Google**, while PyTorch is supported by **Facebook**. Both are helpful, so deciding on which to use is typically a developer's preference. Personally, I prefer the Keras library.

The sequential API allows you to create models layer by layer in a step-by-step fashion. There are two other available models, the **functional API** and **model sub-classing**. We'll use the sequential API because it's the easiest architecture, while the functional API is used for deep learning (complex models) and model sub-classing.

There are a few things we should note about using Keras:

- The model class is the root class and is used to define the architecture for the model.
- Like Python itself, Keras uses object-oriented programming, which means we can add subclasses.
- The subclasses in the model are customizable.

All that said, it should also be noted that sub-classing is more challenging than if we were to use the sequential or functional API. Now let's take a look at an updated algorithm using our Keras library. Remember to include the directory of your file or save the `.csv` file to the necessary directory to run the algorithm correctly:

ch13_diabetesB.py

```
from sklearn.model_selection import train_test_split
from keras import Sequential
from keras.layers import Dense
#Defining the Model
model = Sequential()
model.add(Dense(12, input_dim=8, activation='relu'))
model.add(Dense(15, activation='relu'))
model.add(Dense(8, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
```

From the preceding snippet, we can see that we added four layers that are densely connected.

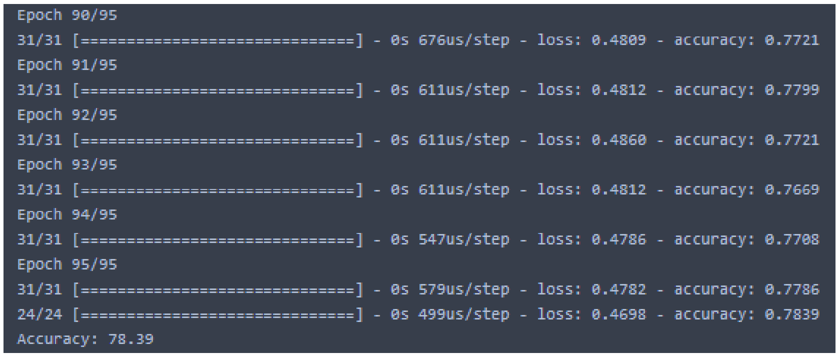
The first layer is built as follows:

- 12 neurons
- `input_dim = 8` (that is, input values that are coming into the network)
- `activation 'relu'`

As you can see, we have added multiple models and defined them. To compile the model, we use the following snippet of code, contained in the same code file. We can also set `model.fit` to use our libraries and the following code, which is part of our `ch13_diabetesB.py` file:

```
#Compile the model
model.compile(loss='binary_crossentropy', optimizer='adam',
    metrics=['accuracy'])
#Fit the model on the dataset
model.fit(x_variables, y_variable, epochs=95,
    batch_size=25)
#Evaluate the model
_, accuracy = model.evaluate(x_variables, y_variable)
print('Accuracy: %.2f' % (accuracy*100))
model.summary()
```

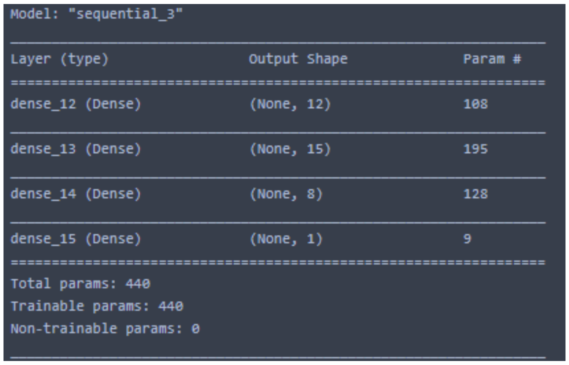
The preceding code compiles the Adam optimizer. The Adam optimizer is used for stochastic gradient descent and updates network weights iteratively using the training data. Once we run our code, our output provides the information shown as follows:



```
Epoch 90/95
31/31 [==============================] - 0s 676us/step - loss: 0.4809 - accuracy: 0.7721
Epoch 91/95
31/31 [==============================] - 0s 611us/step - loss: 0.4812 - accuracy: 0.7799
Epoch 92/95
31/31 [==============================] - 0s 611us/step - loss: 0.4860 - accuracy: 0.7721
Epoch 93/95
31/31 [==============================] - 0s 611us/step - loss: 0.4812 - accuracy: 0.7669
Epoch 94/95
31/31 [==============================] - 0s 547us/step - loss: 0.4786 - accuracy: 0.7708
Epoch 95/95
31/31 [==============================] - 0s 579us/step - loss: 0.4782 - accuracy: 0.7786
24/24 [==============================] - 0s 499us/step - loss: 0.4698 - accuracy: 0.7839
Accuracy: 78.39
```

Figure 13.11 – Output of model run using the Keras library

Note that the accuracy may be different as you run the algorithm. Test the algorithm a few times to see changes in your window. After we've run our accuracy model, we print the model summary using `model.summary()`, shown as follows:



```
Model: "sequential_3"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_12 (Dense)             (None, 12)                108
_________________________________________________________________
dense_13 (Dense)             (None, 15)                195
_________________________________________________________________
dense_14 (Dense)             (None, 8)                 128
_________________________________________________________________
dense_15 (Dense)             (None, 1)                 9
=================================================================
Total params: 440
Trainable params: 440
Non-trainable params: 0
_________________________________________________________________
```

Figure 13.12 – Model summary of the algorithm

Now that we've seen how to run our algorithms using the diabetes data file, let's look briefly at some optimization models that will help us evaluate the algorithm. We will not be going into these algorithms, but we do want to mention a few of the various tools available to us. When we are working with modeling, we use optimization models, such as **binary cross-entropy**, the **Adam optimization algorithm**, and **gradient descent**.

Defining optimization models

Let's take a look at the types of models. Note that we are not diving deep into the use of these models, but further exploration into their application to our algorithm is recommended.

The binary cross-entropy model

In binary classification, we use cross-entropy as the default **loss function**. The loss function is a method that helps us evaluate how our algorithm models the data. With the loss function, we can use optimization in order to produce more accurate results, that is, it helps reduce the **prediction error**. We use the loss function when target values are in the binary set.

Cross-entropy is a loss function that calculates the difference between two probability distributions. We can use cross-entropy when optimizing classification models using **logistic regression** and **artificial neural networks**.

The Adam optimization algorithm

The Adam algorithm is a method for **stochastic optimization**. Stochastic optimization is used when there is randomness in a function, to maximize or minimize the value of the function. The Adam optimization algorithm is appropriate for some simple optimization problems that are *non-convex*. It is efficient and uses little memory but can be applied to large datasets.

The gradient descent model

The gradient descent algorithm is a first-order optimization algorithm. First-order refers to linear local errors. We use gradient descent on functions that can be differentiated to find a local minimum.

The confusion matrix model

The confusion matrix is also known as the **error matrix**. The confusion matrix is visually helpful, as it presents the performance of the algorithm in a table format, which allows better visualization of that performance. It is typically used in supervised learning.

As stated, this is just some base information as you start working on the optimization of your algorithms. Additional information on ML can be found in other Packt books, such as *Python Machine Learning* and *Exploratory Data Analysis with Python*.

Before we move on to an introduction to clusters, let's do a quick recap of what we learned in this section on using the Keras package and model:

- Loading data
- Defining a neural network in Keras
- Compiling a Keras model using the efficient numerical backend
- Training a model on data
- Evaluating a model on data
- Making predictions with the model

Now let's move on to data clusters.

Implementing data clustering

In this section, we're going to take a look at how to approach data clustering. First, let's define what we mean by **data clustering**. Data clustering refers to how we partition data into groups or clusters. Clusters can be meaningful if they provide an expanded understanding of domain knowledge. We use clustering for many applications, such as medicine, where clustering can help identify how a group of patients responds to treatment, or market research, where clustering is used to group consumers in order to appeal to that group based on that particular group's characteristics.

For the purpose of this discussion, we are going to look at synthetic clusters rather than applied clusters. In *Chapter 16*, *Advanced Applied Computational Thinking Problems*, you'll see some examples of clusters in context. A **synthetic cluster** is made from a synthetic dataset. That is, we generate the dataset using an algorithm. Let's take a look at the following code snippet:

ch13_syntheticDataset.py

```
from numpy import where
from sklearn.datasets import make_classification
from matplotlib import pyplot

#Create a synthetic dataset
```

```
X, y = make_classification(n_samples = 1800,
    n_features = 2, n_informative = 2, n_redundant = 0,
    n_clusters_per_class = 1, random_state=4)
#Scatterplot
for class_value in range(2):
     row_ix = where(y == class_value)
     pyplot.scatter(X[row_ix, 0], X[row_ix, 1])
#Display plot
pyplot.xlabel('variable 1')
pyplot.ylabel('variable 2')
pyplot.title('Synthetic data graph')
pyplot.show()
```

Notice from the preceding snippet that we establish the number of samples, the number of features, and the number of clusters, among other things. In addition, we then create a scatterplot of the synthetic data and plot the result. The following graph shows the results of our synthetic dataset plotted as a scatterplot:

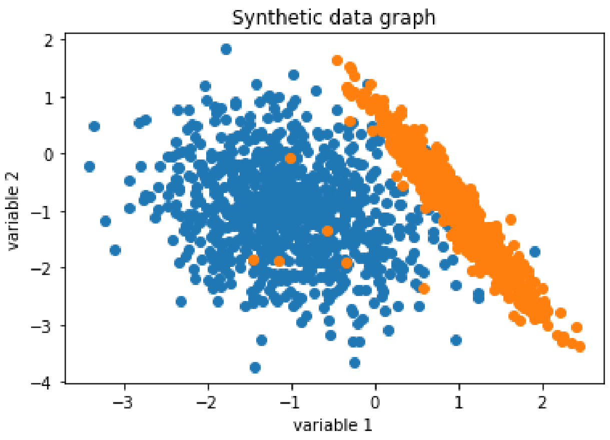


Figure 13.13 – Synthetic dataset scatterplot

Note that the number of samples for this synthetic dataset was `1800`. Try to change the number of samples to see the changes in the scatterplot. Now that we have a dataset, we can start applying clustering algorithms. Here are some of the more common clustering algorithms:

- **The BIRCH algorithm**
- **The K-means clustering algorithm**

We will look at the aforementioned algorithms in the following sections.

Using the BIRCH algorithm

Balanced Iterative Reducing and Clustering using Hierarchies (**BIRCH**) is a clustering algorithm that uses the cluster centroids as long as there is enough available memory and time. For the BIRCH and K-means clustering algorithms, we will share an algorithm and the corresponding plot in order to better understand them.

The following snippet shows us the BIRCH algorithm:

ch13_BIRCH.py

```
from numpy import unique
from numpy import where
from sklearn.datasets import make_classification
from sklearn.cluster import Birch
from matplotlib import pyplot
#Synthetic dataset definition
X, _ = make_classification(n_samples = 1800,
    n_features = 2, n_informative = 2, n_redundant = 0,
    n_clusters_per_class = 1, random_state = 4)
#Define the BIRCH model
model = Birch(threshold = 0.01, n_clusters = 2)
model.fit(X)
yhat = model.predict(X)
#Clusters
clusters = unique(yhat)
#Display
for cluster in clusters:
     row_ix = where(yhat == cluster)
     pyplot.scatter(X[row_ix, 0], X[row_ix, 1])
pyplot.show()
```

Note that we arbitrarily chose two clusters in this sample, as you can see in the line `model = Birch(threshold = 0.01, n_clusters = 2)`. We are sticking with our sample of 1800 so that we can compare our output figures. The following screenshot shows two sample BIRCH models. The first (*on the left*) shows the algorithm run as provided in the preceding snippet. The second (*on the right*) shows the same algorithm run but for three clusters.

To run the second graph, we changed the model line code to the following:

```
model = Birch(threshold = 0.01, n_clusters = 3)
```

Take a look at the following figure with both plots shown, `n_clusters = 2` and `n_clusters = 3`:

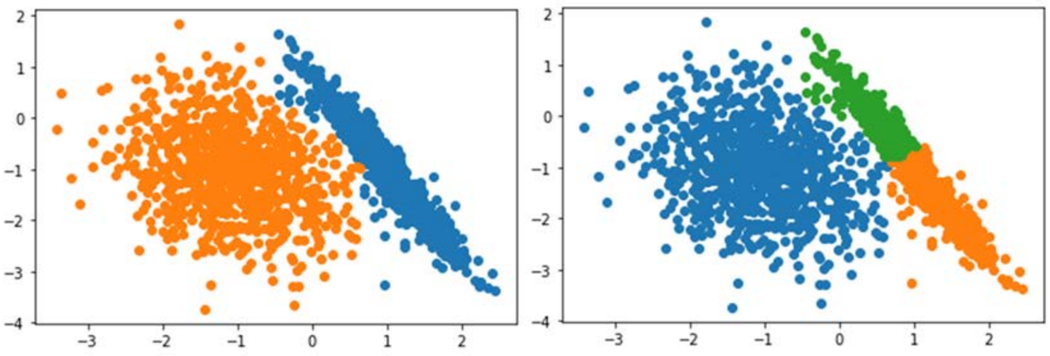


Figure 13.14 – BIRCH models using 2 and 3 clusters, respectively

Notice that on the *left* of the preceding screenshot, two clusters are clearly shown. When compared to *Figure 13.13*, you can see some data points have been converted to fit each of the clusters identified. The scatterplot on the *right* of the preceding screenshot divides the data into three distinct clusters. To get more familiar with the clustering algorithms, change the parameters to see what happens when you change the number of clusters, the sample size, and other parameters.

Now let's take a look at the K-means clustering algorithm.

Using the K-means clustering algorithm

The K-means clustering algorithm is one of the most widely used clustering algorithms. The algorithm assigns examples so that the variance is minimized in each of the identified clusters. Much like with the BIRCH algorithm, we set the number of clusters within the algorithm. Let's take a look at the K-means code snippet:

ch13_KMeans.py

```
from numpy import unique
from numpy import where
from sklearn.datasets import make_classification
from sklearn.cluster import KMeans
from matplotlib import pyplot

#Dataset definition
X, _ = make_classification(n_samples = 1800,
    n_features = 2, n_informative = 2, n_redundant = 0,
    n_clusters_per_class = 1, random_state = 4)
#Model identification and fit
model = KMeans(n_clusters = 2)
model.fit(X)
#Clusters
yhat = model.predict(X)
clusters = unique(yhat)
#Display
for cluster in clusters:
     row_ix = where(yhat == cluster)
     pyplot.scatter(X[row_ix, 0], X[row_ix, 1])
pyplot.show()
```

Again, notice that we're using the same number of clusters (`2`) and the number of samples (`1800`) so that we can compare our displays. The following screenshot shows the K-means scatterplot output resulting from the preceding algorithm:

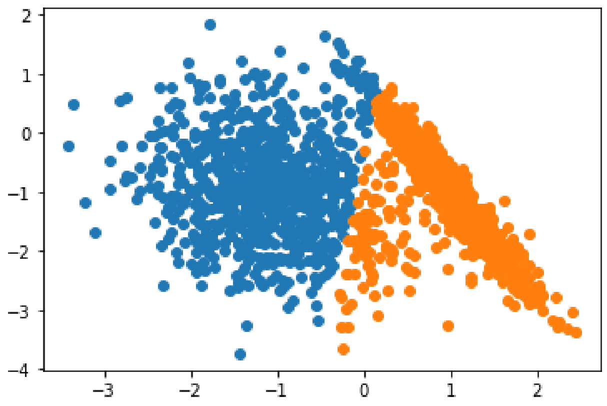


Figure 13.15 – K-means algorithm output

Notice that the data is still exactly the same, however, when we compare the displays we got from the BIRCH algorithm and the K-means algorithm, you can see that our algorithms produced very different results for our clusters.

There are many other clustering algorithms we can use and test. Learning about them and comparing the results is imperative in determining which ones to use based on real datasets. The results from the K-means algorithm, in this case, do not really fit the model well. The BIRCH model seems more suited when using two clusters because the variance in the K-means algorithm is unequal.

As we move on from the clustering examples, please note that, as for much of data science and ML, the more we use the models and algorithms, the more we understand their uses and when the models are appropriate, and we can learn to visually identify whether an algorithm fits our data or not.

Summary

In this chapter, we learned how to use some of the packages available for the Python programming language to create models of large sets of data. We used packages, such as Keras, to upload data and define neural networks. We trained the model and evaluated the model. We used the model to make predictions. We also learned about the classification and testing of data and how to work with data clusters. After reading this chapter, you can now define data training and how Python is used in data training. You can also define and use a clustering algorithm.

We will continue to explore some of the topics discussed here in the next chapter. We will also see some of these applications in the examples provided in *Chapter 16*, *Advanced Applied Computational Thinking Problems*.

14 Using Computational Thinking and Python in Statistical Analysis

In this chapter, we will use Python and the elements of computational thinking to solve problems that require statistical analysis algorithms. We will use **pandas DataFrames** to create statistical analysis algorithms within the Python environment. Additional packages in Python will be needed to create statistical analyses, such as **NumPy**, **pytz**, and more. We will use those packages when they are needed for the code we will work with and when learning what the libraries help us do, such as organizing data with pandas, for example.

In this chapter, we will cover the following topics:

- Defining the problem and Python data selection
- Preprocessing data
- Processing, analyzing, and summarizing data using visualizations

By the end of this chapter, you will be able to design algorithms that best fit the scenarios you are presented with. You will also be able to identify Python functions that best align with the problems presented and generalize your solutions.

Technical requirements

You will need the latest version of Python for running the code in this chapter.

You will need to have the **pandas**, **NumPy**, **SciPy**, and **Scikit-Learn** packages installed for the problems in this chapter.

You can find the full source used in this chapter here: `https://github.com/PacktPublishing/Applied-Computational-Thinking-with-Python/tree/master/Chapter14`

Defining the problem and Python data selection

Before we look at the pandas library, let's define what data analysis is. When we talk about data analysis, we are talking about the process of inspecting, cleansing, transforming, and modeling data with the intent of discovering useful data, notifying conclusions, and supporting decision-making. Decision-making is critical. We don't just want to see what the data says has happened in the past. We want to use data in order to make informed decisions for the future.

Take a look at some of the uses of data analysis:

- **Business**: It helps when making decisions based on customer trends and behavior prediction, increasing business productivity, and driving effective decision-making.
- **Weather forecasting**: Data about the atmosphere (temperature, humidity, wind, and more) is collected and analyzed to understand atmospheric processes (meteorology) to determine how the atmosphere will evolve in the future.

- **Transportation**: Data can be used to determine trends, including traffic, accidents, and more, helping us make decisions about traffic patterns, traffic light durations, and much more.

The aforementioned uses are only some of the possible applications, of course. Data analytics is used for a very wide range of things, and by businesses and educational organizations to make critical decisions, provide resources to the community, fund our schools and colleges, and so much more.

So, let's take a look at what tools we have available to analyze that data. One of the main libraries used in data analysis in Python is the pandas package. The greatness of pandas lies in its ease of use, easy data structure, and high performance. Using pandas simplifies our work in data analysis.

Defining pandas

One thing that's important to note is that pandas is built on top of NumPy. **NumPy** is a package that helps us work with arrays. Python doesn't have arrays per se, so the packages allow us to create them, use them, and then build upon that capability.

Pandas provides a flexible and easy data structure to simplify your work in data analysis. It is a great tool for **big data**. When we talk about big data, we're talking about structured and unstructured datasets that are analyzed so that we can get better insights and aid in decision making or strategies for businesses and organizations. pandas can handle importing different formats, such as `.csv` files, **SQL**, and **JSON**, as well as all sorts of manipulation, such as selecting data, merging, reshaping, and cleaning the data.

There are two different ways to store data in pandas—series and DataFrames:

- **Series** are one-dimensional arrays that hold any data type (integer, string, or float); series represent one column of data.
- **DataFrames** are two-dimensional objects that can have multiple columns and data types. They take inputs such as dictionaries, series, lists, and other DataFrames.

Let's now learn when to use pandas.

Determining when to use pandas

Pandas is extremely useful in general, but it is really a great tool when we are looking at large data and working with **comma-separated values** (**CSV**) files. These files are stored as tables, such as spreadsheets. The other thing is that we can establish *chunks* in pandas. Yes, there's a `chunksize` parameter in pandas that helps us break down our data. Let's say we have 5,000,000 rows. We can decide to use `chunksize` to break that down by 1,000,000 rows.

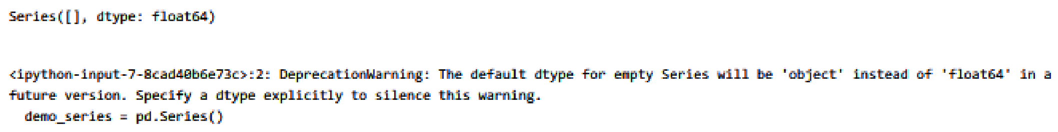
In addition, we sometimes have massive data files but only want to look at some of the components. Pandas allows us to identify the columns we want to include and those we want to ignore.

Working with pandas series

As mentioned in the previous section—*Defining pandas*—series are one-dimensional. We can create an empty pandas series using some simple code. Note that we are importing the pandas library first, as is usual when working with packages and libraries, as follows:

```
import pandas as pd
demo_series = pd.Series()
print(demo_series)
```

The default series created will have a type of `float` because we didn't establish any other type in our algorithm. However, the console prints a warning that in the future, empty series **dtypes** will be set as an object rather than `float`. `dtype` stands for data type; in this case, it's a float. Take a look at the following screenshot, which shows the output when we run our algorithm:



```
Series([], dtype: float64)


<ipython-input-7-8cad40b6e73c>:2: DeprecationWarning: The default dtype for empty Series will be 'object' instead of 'float64' in a
future version. Specify a dtype explicitly to silence this warning.
  demo_series = pd.Series()
```

Figure 14.1 – Output when creating empty series in pandas without identifying dtype

As you can see, there really isn't an error, just a warning about how the data was stored.

Now, let's create a series with defined elements. To do that, we'll need to import both pandas and `numpy` so that we can create the arrays and then the series. Take a look at the following snippet of code:

```
ch14_seriesDemo.py
import pandas as pd
import numpy as np
dataset = np.array(['yellow', 'green', 'blue', 'violet',
'red'])
data_series = pd.Series(dataset)
print(data_series)
```

As you can see from the preceding code, we stored our array using `numpy`, then created a series using that array. Finally, we printed the series. The output is a table, as follows:

```
0    yellow
1     green
2      blue
3    violet
4       red
dtype: object
```

We can also get the exact same thing if we created a list first. Take a look at this snippet of code:

ch14_seriesDemo2.py

```
import pandas as pd
import numpy as np
myList = ['yellow', 'green', 'blue', 'violet', 'red']
data_series = pd.Series(myList)
print(data_series)
```

Notice that we created the series directly from the list. We're not going to show the output for this particular snippet of code because it's exactly the same as the previous snippet's output.

We can also get a series from a dictionary. Let's take a look at that in the following snippet of code:

ch14_seriesDemo3.py

```
import pandas as pd
myDictionary = {
    'Name' : 'Miguel',
    'Number' : 42,
    'Age' : 'unknown'
    }
mySeries = pd.Series(myDictionary)
print(mySeries)
```

The preceding code is just a demo, but we do get a table series that contains the values of our dictionary when we run the algorithm. Let's take a look at that output:

```
Name       Miguel
Number         42
Age       unknown
dtype: object
```

As you can see, we have the two columns based on the key-value pairs, and the type, which is `object`. One thing that will become important is that series don't have column titles. For that, we'll need to use DataFrames.

What if we want to access a specific element in a series? Well, to access the first element, let's use the following snippet of code:

ch14_demo4.py

```
import pandas as pd
myDictionary = {
    'Name' : 'Miguel',
    'Number' : 42,
    'Age' : 'unknown'
    }
mySeries = pd.Series(myDictionary)
print(mySeries[0])
```

The output of the preceding code is simply `Miguel`. That's because we used index 0 to identify what we wanted from that dictionary, so it'll give us the value for the first key-value pair. If we wanted the value pairs for the first two key-value pair elements, we'd replace `(mySeries[0])` with `(mySeries[:2])`. Then, the output would be as follows:

```
Name      Miguel
Number        42
dtype: object
```

There are many other things that can be done with series, so play around with the indexes and creating different types of series using lists, dictionaries, or NumPy arrays. For now, let's move on to DataFrames.

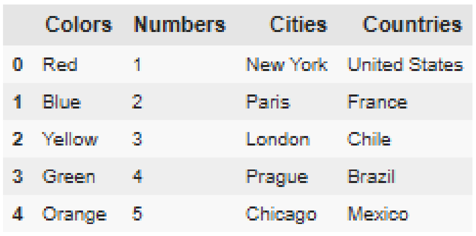
Working with pandas DataFrames

Now, let's take a look at how we work with DataFrames. First, let's take a look at a `.csv` file with pandas. We're going to use the `demo.csv` file in the snippet of code that follows. Please replace the location of the file to match the location where you have saved the file, which can be found in the GitHub repository:

ch14_csvDemo.py

```
import pandas as pd
data_set = pd.read_csv('C:\\...\\demo.csv')
data_set.head()
```

The preceding code does three things. It imports the pandas package so that we can use the data capabilities we need, it tells the program to open the data file we'll be working with, and then it gives us the first few rows of data in the file so that we can see what we're working with. The following screenshot shows the result, or output, from the preceding code:



	Colors	Numbers	Cities	Countries
0	Red	1	New York	United States
1	Blue	2	Paris	France
2	Yellow	3	London	Chile
3	Green	4	Prague	Brazil
4	Orange	5	Chicago	Mexico

Figure 14.2 – Output showing the first few rows of the dataset

As you can see from the preceding screenshot, the table does not show all the values included in our file. While our file is not full of big data, it does include more rows of information. This is just a way for us to get a preview of our data.

But this is a *clean* dataset. *What happens if we have a dataset with rows or columns that are missing information?* Well, pandas allows us to work with the file to prepare it for us. DataFrames also prepare our data so that we can create visual representations. These visuals, or plots, will allow us to see trends, make predictions, determine what values we can use for training, and much more. The DataFrame is really just the backbone of everything else we can then do with a dataset.

In this section, we learned about problems, how to work with pandas, and some of the capabilities of pandas series and DataFrames.

As a note, for the rest of this chapter, we'll be a lot more focused on DataFrames than series. But before we get into an application, let's take a look at how we can avoid errors and pitfalls by preprocessing our DataFrame.

Preprocessing data

Preprocessing data is a technique that transforms raw data into a useable and efficient format. It is, in fact, the most important step in the data mining and machine learning process.

When we are preprocessing data, we are really cleaning it, transforming it, or doing a data reduction. In this section, we will take a look at what these all mean.

Data cleaning

Data cleaning refers to the process of making our dataset more efficient. If we go through data cleaning in really large datasets, we can expedite the algorithm, avoid errors, and get better results. There are two things we deal with when data cleaning:

- **Missing data**: This can be fixed by ignoring the data or manually entering a value for the missing data.
- **Noisy data**: This can be fixed/improved by using binning, regression, or clustering, among other processes.

We're going to look at each of these things in more detail.

Working with missing data

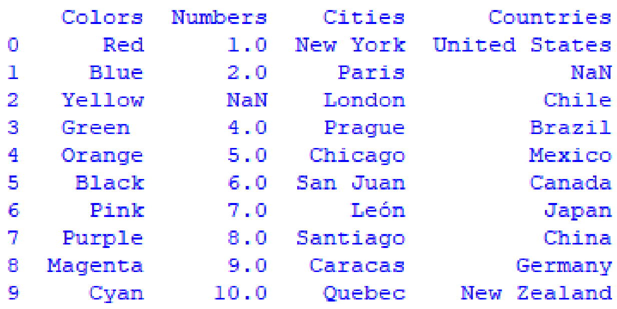
Let's take a look at how we deal with missing data. First, we're going to learn how to ignore missing data. We can use pandas to find rows with missing values. When we do that, we're cleaning our dataset. Now, we're not going to go through every method we can use, just one where we get rid of rows with missing values. As always, the dataset used is available in our GitHub repository and you'll need to update your file location. Let's look at the following code snippet:

ch14_cleaningDemo1.py

```
import pandas as pd
myData = pd.read_csv('C:\\...\\demo_missing.csv')
```

```
print(myData)
cleanData = myData.dropna(axis = 0, how = 'any')
print(cleanData)
```

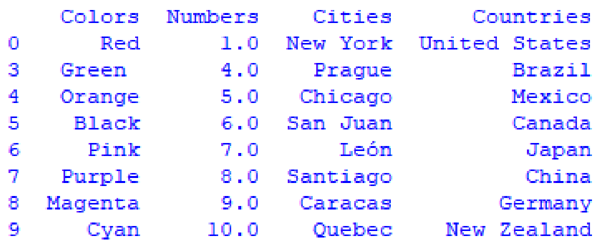
In the preceding code, the first `print` statement is for our own sake so that we can see what our dataset looks like. *You'll never want to do that with huge files!* The following screenshot shows the first `print` output:



```
    Colors  Numbers    Cities      Countries
0      Red      1.0  New York  United States
1     Blue      2.0     Paris            NaN
2   Yellow      NaN    London          Chile
3   Green       4.0    Prague         Brazil
4   Orange      5.0   Chicago         Mexico
5    Black      6.0  San Juan         Canada
6     Pink      7.0      León          Japan
7   Purple      8.0  Santiago          China
8  Magenta      9.0   Caracas        Germany
9     Cyan     10.0    Quebec    New Zealand
```

Figure 14.3 – First print statement, the original dataset

Notice that the **1**, **Blue** column has a value of **NaN** under **Countries** and the next column (**2**, **Yellow**) has a missing value under the **Numbers** column. When we use `dropna()`, the algorithm will drop the rows with missing values. The following screenshot shows the printed statement with the altered dataset:



```
    Colors  Numbers    Cities      Countries
0      Red      1.0  New York  United States
3   Green       4.0    Prague         Brazil
4   Orange      5.0   Chicago         Mexico
5    Black      6.0  San Juan         Canada
6     Pink      7.0      León          Japan
7   Purple      8.0  Santiago          China
8  Magenta      9.0   Caracas        Germany
9     Cyan     10.0    Quebec    New Zealand
```

Figure 14.4 – Printed clean dataset

As you can see from the preceding screenshot, the two rows that were missing values were eliminated in our new dataset. Now, we could run whatever analysis we wanted to for this data.

If you wanted to check only one column to verify whether missing values exist, you could use the following code snippet:

ch14_cleaningDemo2.py

```
import pandas as pd
myData = pd.read_csv('C:\\...\\demo_missing.csv')
print(pd.isna(myData['Countries']))
```

Notice in the preceding algorithm that we used the `Countries` column heading to verify that particular column. When we run the algorithm, here's what our output looks like:

```
0    False
1     True
2    False
3    False
4    False
5    False
6    False
7    False
8    False
9    False
Name: Countries, dtype: bool
```

As you can see, the second row in our dataset has a missing value in the `Countries` column.

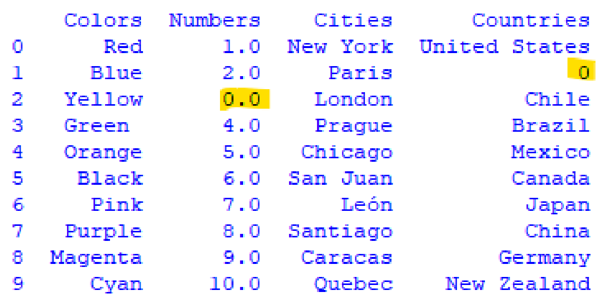
While we're not going to go into every method, you can also remove columns. You can choose to remove rows and/or columns with a certain number of missing values. For example, you could choose to remove only rows or columns that have more than two missing values. If you did that, you'd still need to worry about the values that may still be missing in columns or rows that were not removed because there was only one missing value. For those, you may choose to replace the missing values with something.

Let's say you want to replace a value given the column. To do so, let's take a look at the following code snippet:

ch14_cleaningDemo3.py

```
import pandas as pd
myData = pd.read_csv('C:\\...\\demo_missing.csv')
print(myData.fillna(0))
```

From the preceding code, notice that we are filling each empty cell with the value `0`. When we run the algorithm, we get the following output:



```
    Colors  Numbers    Cities      Countries
0      Red      1.0  New York  United States
1     Blue      2.0     Paris              0
2   Yellow      0.0    London          Chile
3    Green      4.0    Prague         Brazil
4   Orange      5.0   Chicago         Mexico
5    Black      6.0  San Juan         Canada
6     Pink      7.0      León          Japan
7   Purple      8.0  Santiago          China
8  Magenta      9.0   Caracas        Germany
9     Cyan     10.0    Quebec    New Zealand
```

Figure 14.5 – Replaced missing values

Notice the highlighted values in the preceding screenshot. Those are the values replaced by our algorithm. Now, let's take a look at how we deal with noisy data.

Working with noisy data

First, let's define what we mean by **noisy data**. When we have a really large amount of data and some of it is not useful for our analysis, we say it is noisy data. Noisy data is also used to refer to data corruption. *Really, it's just useless data.*

Three of the ways that we deal with noisy data are binning, regression, and clustering:

- **Binning** uses neighboring data to smoothen a sorted data value. The sorted values go in bins, which are groups created within the algorithm.
- The **clustering** method identifies and removes outliers in a dataset.
- The **regression** method smoothens data by fitting it into regression functions.

The purpose of binning is to reduce some errors. In binning, data is divided into small buckets or bins. The data is then replaced with a calculated bin value. When we go through the binning process, we are smoothing the data.

Here's an example with a simple, numerical dataset in the algorithm. The following snippet of code will create bins with equal frequency:

ch14_binning1.py

```
#Binning with equal frequency
def equal_frequency(array1, m):
     l = len(array1)
     n = int(l / m)
     for i in range(0, m):
           array = []
           for j in range(i * n, (i + 1) * n):
                 if j >= l:
                       break
                 array = array + [array1[j]]
           print(array)
#Input dataset
dataset = [3, 6, 7, 9, 11, 14, 10, 15, 19, 35, 38, 45, 48, 49, 
76]
#Input number of bins
m = 5
print("Equal Frequency Binning: ")
equal_frequency(dataset, m)
```

When looking at the preceding code, you can see that the number of bins is defined as `5`, so the data will be binned into five lists. Binning really is a way to group information. We tell the algorithm we want to do it and how many bins we want, and it provides the data in those bins, or groups. In this case, we get those five lists. Take a look at the output:

```
Equal Frequency Binning:
[3, 6, 7]
[9, 11, 14]
[10, 15, 19]
[35, 38, 45]
[48, 49, 76]
```

As you can see, the algorithm created five bins with three values in each of the bins.

> **Important Note:**
> Note that the binning process doesn't organize our data for us. So, if we reordered our values, they would still be binned in the order that the data was entered.

Now, let's take a look at a binning algorithm that uses equal width:

ch14_binning2.py

```
#Binning with equal width
def equal_width(array1, m):
    w = int((max(array1) - min(array1)) / m)
    min1 = min(array1)
    array = []
    for i in range(0, m + 1):
        array = array + [min1 + w * i]
    arrayi=[]
    for i in range(0, m):
        result = []
        for j in array1:
            if j >= array[i] and j <= array[i+1]:
                result += [j]
        arrayi += [result]
    print(arrayi)

#Input dataset
dataset = [3, 6, 7, 9, 11, 14, 10, 15, 19, 35, 38, 45, 48, 49,
76, 81, 208, 221]
#Input number of bins
m = 3

print("\nEqual Width Binning:")
equal_width(dataset, m)
```

The preceding snippet of code breaks down our data into three bins. The goal of equal width binning is to divide the dataset into bins of equal size, which in the case of equal width binning means equal range. The data will be split, but it is important to note that we're talking about range here, so the bins won't have the same number of elements in each of them for this particular dataset. The output for the preceding snippet of code is as follows:

```
Equal Width Binning:
```

```
[[3, 6, 7, 9, 11, 14, 10, 15, 19, 35, 38, 45, 48, 49], [76,
81], [208]]
```

As you can see, the binning produces an output that doesn't look quite as clean as the equal frequency output, but is actually more popular.

Now, let's talk about transforming data.

Transforming data

Pandas allows us to transform data. Here are some of the ways that we can transform it:

- **Normalization** transforms values into new range; the most popular is **min-max normalization**, given as follows:

$$x_{\text{scaled}} = \frac{x - \min(x)}{\max(x) - \min(x)}$$

- **Attribute selection** is the process of transforming data by replacing an attribute with a different attribute or attributes.
- **Concept hierarchy** is actually a transformation done by reducing data. It is done by replacing concepts such as numbers (*10*, *15*, *40*) with higher-level concepts, such as qualifiers (*short*, *lengthy*, *extremely lengthy*).

In the next section, we will glance through the reduction of data.

Reducing data

Data reduction refers to a process that allows us to get similar or even the same results from a dataset but only after having reduced the representation of the data in volume.

We won't go into too much depth with all the concepts here because they are much easier to look at in examples, but here are some of the ways that data reduction is done:

- Removing invalid data from the dataset
- Creating summaries for the data at different levels

Think of removing invalid data as taking care of any outliers. The data may have been entered incorrectly, conditions may have not been optimal, or similar. When we have a data point that doesn't fit the entirety of the dataset, especially where we have a large number of data points to compare it to, we can remove that data point as invalid or as an outlier.

When creating summaries at different levels, we are aggregating the dataset and testing and producing summaries at each of those levels. Say you had 100 data points (datasets will often be in the thousands, but it's easier to explain with smaller numbers). We could create a summary for the first 20 data points. Then, we could do the same for the first 40, then the first 60, and so on. When compared, we could see the trends and use those smaller sub-sections of our dataset to make our predictions if the trends hold true. That's what data reduction helps with, simplifying our dataset while still getting accurate results.

In this section, we learned how to work with data that needs to be cleaned. We learned some of the ways that we can clean data, such as eliminating missing data or replacing missing points. We also learned about noisy data and how to address problems with noisy data in our datasets. Finally, we learned about data reduction and how we can get accurate results by removing invalid data and creating aggregate summaries of our data.

This is just an introduction to the types of things we do when we're working with data. So, let's look at an example so that we can put some of this into context.

Processing, analyzing, and summarizing data using visualizations

We're working in real estate now, and since we want to do well, we really want to build an algorithm that helps us analyze data and predict housing prices. But let's think about that for a second. We can define that problem very broadly or narrowly. We can do a pricing analysis for all houses in a state or houses with three bedrooms or more in a neighborhood. *Does performing the analysis matter? Maybe. But isn't that why we want to look at this problem?*

Let's take a look at how we can process the data first.

Processing data

Let's start by gathering some data. For this problem, we're using the `kv_house_data.csv` dataset, which is available in our GitHub repository. To look at this dataset, we'll need quite a few libraries. We've been talking about pandas mostly, yes, but we want to also do visualizations and perform some analysis, so we'll need **Seaborn**, **SciPy**, and **Scikit-Learn**. The full algorithm can be found in the `ch14_housePrice_prediction.py` file. We'll look at it in snippets to discuss what we're doing along the way:

ch14_housePrice_prediction.py

```
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
```

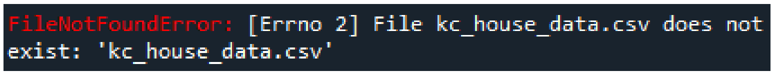
So, what are all these libraries? Pandas we know about. We're using that for organizing our data. NumPy helps us with the arrays. Seaborn and Matplotlib are both used for visualizations. If we were taking this further to create models by training using the dataset, we'd also need Scikit-Learn. For this example, we're going to stick with some of the plots we can get before training.

Now, let's import our dataset. Remember from *Chapter 12*, *Using Python in Experimental and Data Analysis Problems*, that you can set the directory of your file directly. You can also provide the entire path to the data file, as we have done previously in this chapter, such as in the cleaning demo. You can use `os.chdir()` to establish the directory, adding the location of your file in parentheses, then use the following code snippet to read the `.csv` file:

```
housing_data= pd.read_csv("kc_house_data.csv")
```

We are using a pandas function here. *See that* `pd.read_csv()`*?* That `pd` is pandas, since we imported pandas as `pd`, and `read_csv()` is the function that allows the algorithm to get the information in the file.

If you forget to enter your directory or include the wrong location for it, you will receive an error code, as seen in the following screenshot:



```
FileNotFoundError: [Errno 2] File kc_house_data.csv does not
exist: 'kc_house_data.csv'
```

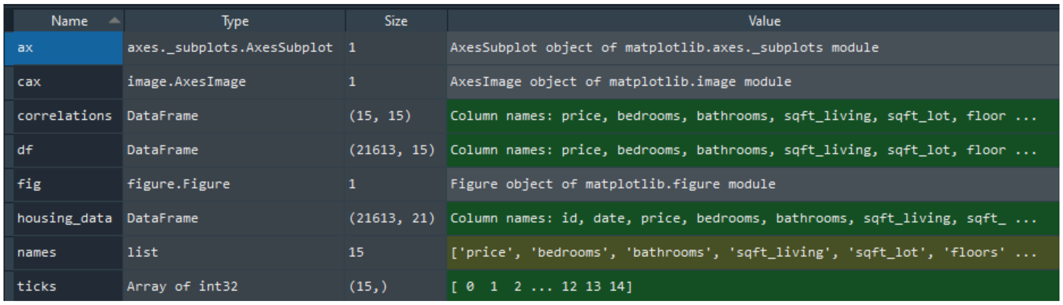
Figure 14.6 – Path to file error

As you can see, Python will make sure you know that you've made a mistake. *That can get aggravating at times, but it is certainly helpful.*

Now that we have our data, we'll need to check it and clean it. This is where all the content we shared previously in the chapter comes into play. Let's see what that looks like in practice.

Analyzing and summarizing data

Now, there's something we haven't really talked about much, and that's Python's variable explorer. More specifically, it's **Spyder's Python variable explorer**. Spyder is an integrated environment and is free of charge. It works with Python, running Python as usual, but also provides us with better editing tools. The following screenshot shows how it should look when you import your dataset from the variable explorer in Python. When we run the `ch14_housePrice_prediction.py` Python algorithm in Spyder, we can see our variables in the variable explorer. The following screenshot shows the data we get from the variable explorer when we run this algorithm:

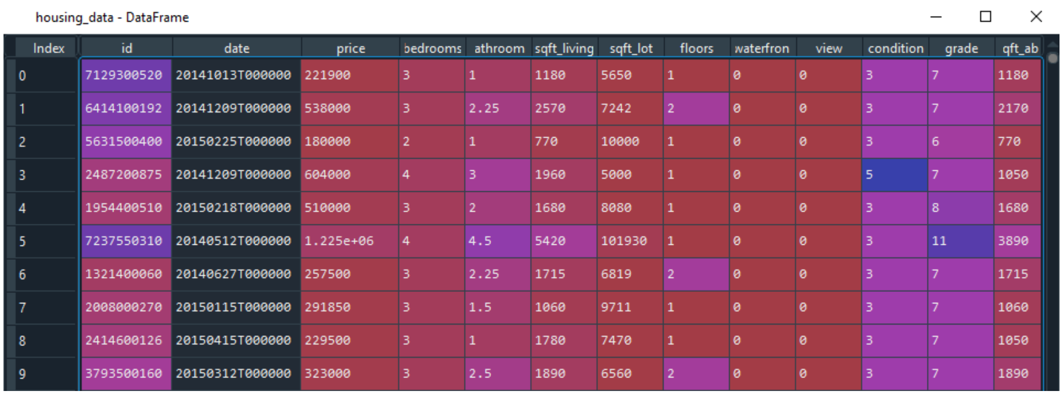


Name	Type	Size	Value
ax	axes._subplots.AxesSubplot	1	AxesSubplot object of matplotlib.axes._subplots module
cax	image.AxesImage	1	AxesImage object of matplotlib.image module
correlations	DataFrame	(15, 15)	Column names: price, bedrooms, bathrooms, sqft_living, sqft_lot, floor ...
df	DataFrame	(21613, 15)	Column names: price, bedrooms, bathrooms, sqft_living, sqft_lot, floor ...
fig	figure.Figure	1	Figure object of matplotlib.figure module
housing_data	DataFrame	(21613, 21)	Column names: id, date, price, bedrooms, bathrooms, sqft_living, sqft_ ...
names	list	15	['price', 'bedrooms', 'bathrooms', 'sqft_living', 'sqft_lot', 'floors' ...
ticks	Array of int32	(15,)	[0 1 2 ... 12 13 14]

Figure 14.7 – Variable explorer view in Spyder

When we are dealing with a lot of data and larger algorithms, this tool becomes really critical. We can get a lot of information from just this tool. For example, let's look at the `housing_data` variable. In *Figure 14.7*, you can see that the type for this variable in our algorithm is **DataFrame** with a size of **(21613, 21)**.

If you double-click on the variable in the variable explorer, you get what's shown in the following screenshot (please note that the screenshot may look different depending on the environment you are using. When running this code using environments such as Spyder or Jupyter, depending on your theme settings and choices, the table may look different, with different color schemes or no color schemes):



housing_data - DataFrame

Index	id	date	price	bedrooms	athroom	sqft_living	sqft_lot	floors	waterfron	view	condition	grade	qft_ab
0	7129300520	20141013T000000	221900	3	1	1180	5650	1	0	0	3	7	1180
1	6414100192	20141209T000000	538000	3	2.25	2570	7242	2	0	0	3	7	2170
2	5631500400	20150225T000000	180000	2	1	770	10000	1	0	0	3	6	770
3	2487200875	20141209T000000	604000	4	3	1960	5000	1	0	0	5	7	1050
4	1954400510	20150218T000000	510000	3	2	1680	8080	1	0	0	3	8	1680
5	7237550310	20140512T000000	1.225e+06	4	4.5	5420	101930	1	0	0	3	11	3890
6	1321400060	20140627T000000	257500	3	2.25	1715	6819	2	0	0	3	7	1715
7	2008000270	20150115T000000	291850	3	1.5	1060	9711	1	0	0	3	7	1060
8	2414600126	20150415T000000	229500	3	1	1780	7470	1	0	0	3	7	1050
9	3793500160	20150312T000000	323000	3	2.5	1890	6560	2	0	0	3	7	1890

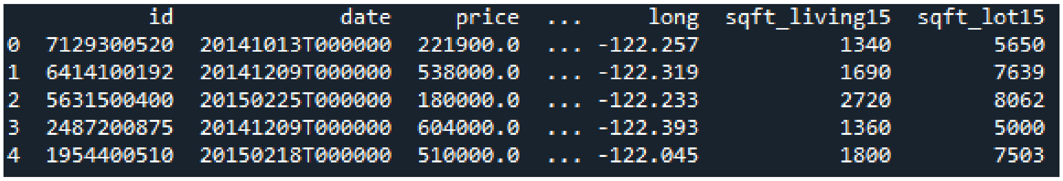
Figure 14.8 – DataFrame variable view in Spyder

This is only one way to get some of the information for our DataFrame. Spyder allows us to resize the window as well, so we can take a look at more columns, scroll through them to find values, and so on. It's not that easy if we're in the Python console. You can get the information, just not as easily.

Here's the code that can give us some of the information:

```
housing_data.head()
```

The preceding code will show us the first five rows of our dataset. Take a look at the following screenshot; we have ellipses (**...**) between **price** and **long**. That's because Python wants to let us know that there are additional columns between those two:



```
           id             date     price  ...     long  sqft_living15  sqft_lot15
0  7129300520  20141013T000000  221900.0  ... -122.257           1340        5650
1  6414100192  20141209T000000  538000.0  ... -122.319           1690        7639
2  5631500400  20150225T000000  180000.0  ... -122.233           2720        8062
3  2487200875  20141209T000000  604000.0  ... -122.393           1360        5000
4  1954400510  20150218T000000  510000.0  ... -122.045           1800        7503
```

Figure 14.9 – First few rows of the DataFrame

As you can see, the rows help us see what our dataset looks like, but nothing else. So, we can also take a look at the size of our DataFrame by using the following code:

```
housing_data.shape
```

When we run the preceding code, we get the following output:

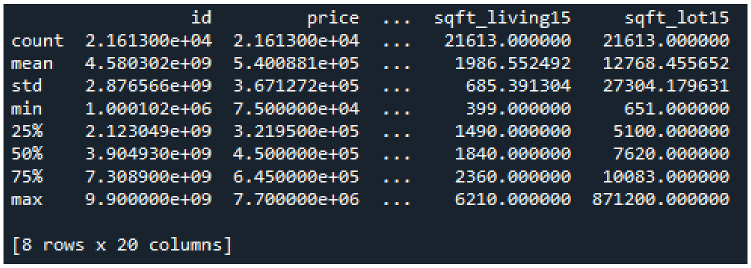
```
(21613, 21)
```

As you can see, now we have the shape or size of our DataFrame. *What does it mean?* It means we have `21,613` rows of data in `21` columns. Regardless of whether you were in Spyder, the Python console, or another environment of your choice, you can see that your data was successfully imported.

Now that we have the raw data imported, let's see whether we can get more information. We can use the following code to get a summary:

```
housing_data.describe()
```

The `describe()` function generates a summary that includes details such as the mean, standard deviation, and percentile. This percentile is a part of your five-number summary for datasets, used to create **boxplots**. While we won't create a boxplot in this problem, that visual representation can be helpful depending on our goals for our algorithms. Take a look at the following screenshot for the results of using the `describe()` function:



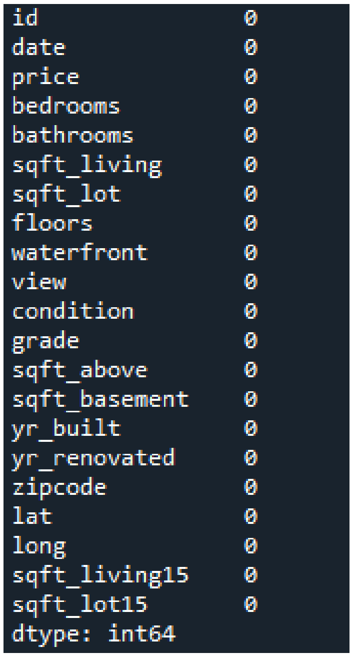
```
                 id         price  ...  sqft_living15     sqft_lot15
count  2.161300e+04  2.161300e+04  ...   21613.000000   21613.000000
mean   4.580302e+09  5.400881e+05  ...    1986.552492   12768.455652
std    2.876566e+09  3.671272e+05  ...     685.391304   27304.179631
min    1.000102e+06  7.500000e+04  ...     399.000000     651.000000
25%    2.123049e+09  3.219500e+05  ...    1490.000000    5100.000000
50%    3.904930e+09  4.500000e+05  ...    1840.000000    7620.000000
75%    7.308900e+09  6.450000e+05  ...    2360.000000   10083.000000
max    9.900000e+09  7.700000e+06  ...    6210.000000  871200.000000

[8 rows x 20 columns]
```

Figure 14.10 – Using the describe() function

Now, we should note that the function analyzes numeric values from the DataFrame. It excludes `NaN` values. `NaN` stands for **Not a Number** values in Python and represents any value that is undefined or unrepresentable. `NaN` can also represent missing numbers. We talked about those missing numbers earlier in the *Working with missing data* section and some of the ways we could tackle them. Let's look at that in context now. We want to find our missing values. For that, we can run the following code snippet:

```
housing_data.isnull().sum()
```

The preceding snippet will let us know if there are missing data points in each of the columns in our dataset. Then, it will aggregate the values as a sum. So, if we had two missing values under the `date` column, we'd expect to see a `2` there. Our results can be seen in the following screenshot:



```
id               0
date             0
price            0
bedrooms         0
bathrooms        0
sqft_living      0
sqft_lot         0
floors           0
waterfront       0
view             0
condition        0
grade            0
sqft_above       0
sqft_basement    0
yr_built         0
yr_renovated     0
zipcode          0
lat              0
long             0
sqft_living15    0
sqft_lot15       0
dtype: int64
```

Figure 14.11 – Results from running the isnull() and .sum() functions

We don't have to clean this dataset! That's a clean dataset. However, learning how to identify whether or not we would need to do so is really important. Now that we know that we have a clean dataset, we can start working on building some visualizations.

Using data visualization

Remember that this is still part of our initial algorithm file, `ch15_housePrice_prediction.py`. If you open that file, the code that follows starts at *line 25*. We have added comments for descriptions between the lines of code:

```
names=['price','bedrooms','bathrooms','sqft_living','sqft_
lot','floors','waterfront','view','condition','grade','sqft_
above','sqft_basement','zipcode','lat','long']
df=housing_data[names]
```

The first thing we're doing is identifying the columns that we'll be using for our plot. After we do that, we'll make a DataFrame and save it as `df`:

```
correlations= df.corr()
```

The preceding code snippet creates the correlations for our DataFrame. In the next code snippet from the file, we'll create our figure; that is, we'll work on the data visualization:

```
fig=plt.figure()
ax=fig.add_subplot(111)
cax=ax.matshow(correlations,vmin=-1,vmax=1)
fig.colorbar(cax)
```

In the preceding code, we named our figure, added the subplot, and identified our colors. Next, we'll have to set some of our properties, such as tick marks, the distance between tick marks, and labels for axes and tick marks:

```
ticks=np.arange(0,15,1)
ax.set_xticks(ticks)
ax.set_yticks(ticks)
ax.set_xticklabels(names, rotation =' 90')
ax.set_yticklabels(names)
```

After we've set some of the properties, we can ask for the plot to use `tight_layout()`. This will help us see all the details of the plot and labels. If we do not use `tight_layout()`, we'll sometimes have some labels that will not be visible in our graphs:

```
plt.tight_layout()
plt.savefig('Correlation_graph.png',dpi = 300)
plt.show()
```

In the preceding snippet, we also created a save file and defined the figure's size. Finally, we asked the algorithm to show us the correlations. The following screenshot shows us the result of the preceding code:

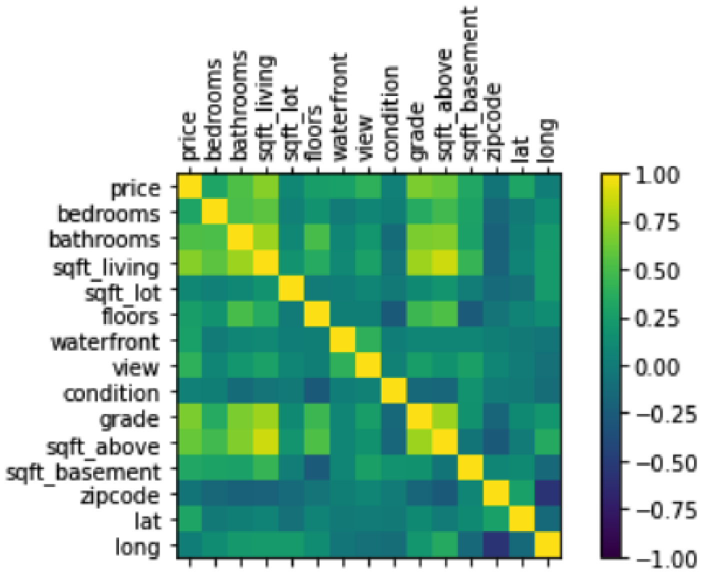


Figure 14.12 – Correlations plot for housing data

As you can see, we just created a **correlation matrix** using Python. And it's a pretty great matrix. *But what is a correlation matrix?* A correlation matrix looks at all the values in our DataFrame and then calculates how closely they are correlated, giving a value between **-1** and **1**. The closer the value is to **1**, the more correlated the values are. Each value is compared to itself, which is a perfect correlation, of course, and seen as the diagonal in our matrix.

The rest of the graph, where all the values are compared to each other, has to be looked at more closely. The closer to the yellow color, the closer the values are correlated. So, take a look at the *y*-axis **sqft_above** value and the corresponding value for **sqft_living** on the *x* axis. That value is close to the yellow value of **1**, but not quite. We've highlighted those values in the following screenshot:

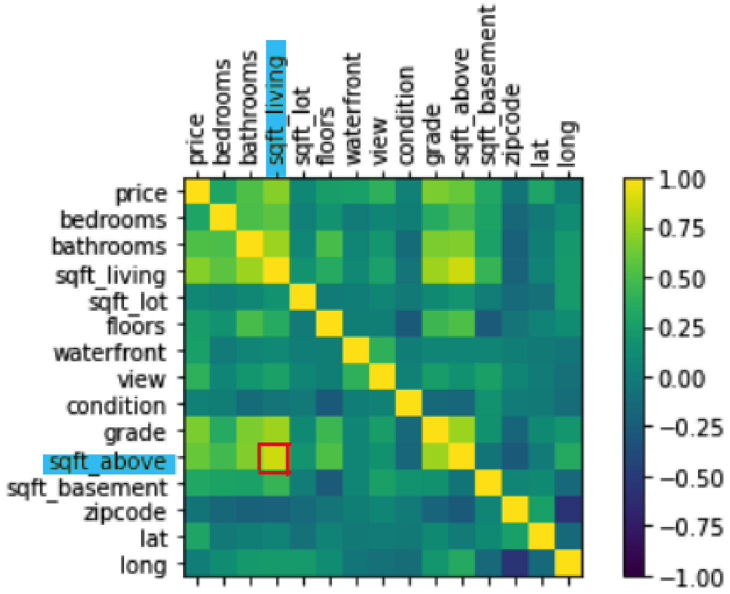


Figure 14.13 – Highlighted correlation of sqft_above and sqft_living

There are other values that show some correlation, but not quite strong enough. This plot helps us then make decisions to maybe take a closer look at that correlation, find more information about it, and so on.

Now, we can also take a look at a pretty intense plot matrix that's called *pair plotting*. A **pairs plot** shows us the distribution of single variables and the relationships between two variables. If we ran a pairs plot for the data that is included in our algorithm, we get a massive graph, as shown in the following screenshot (please note that this plot can take up to a few minutes to generate because of the amount of data being analyzed and plotted):

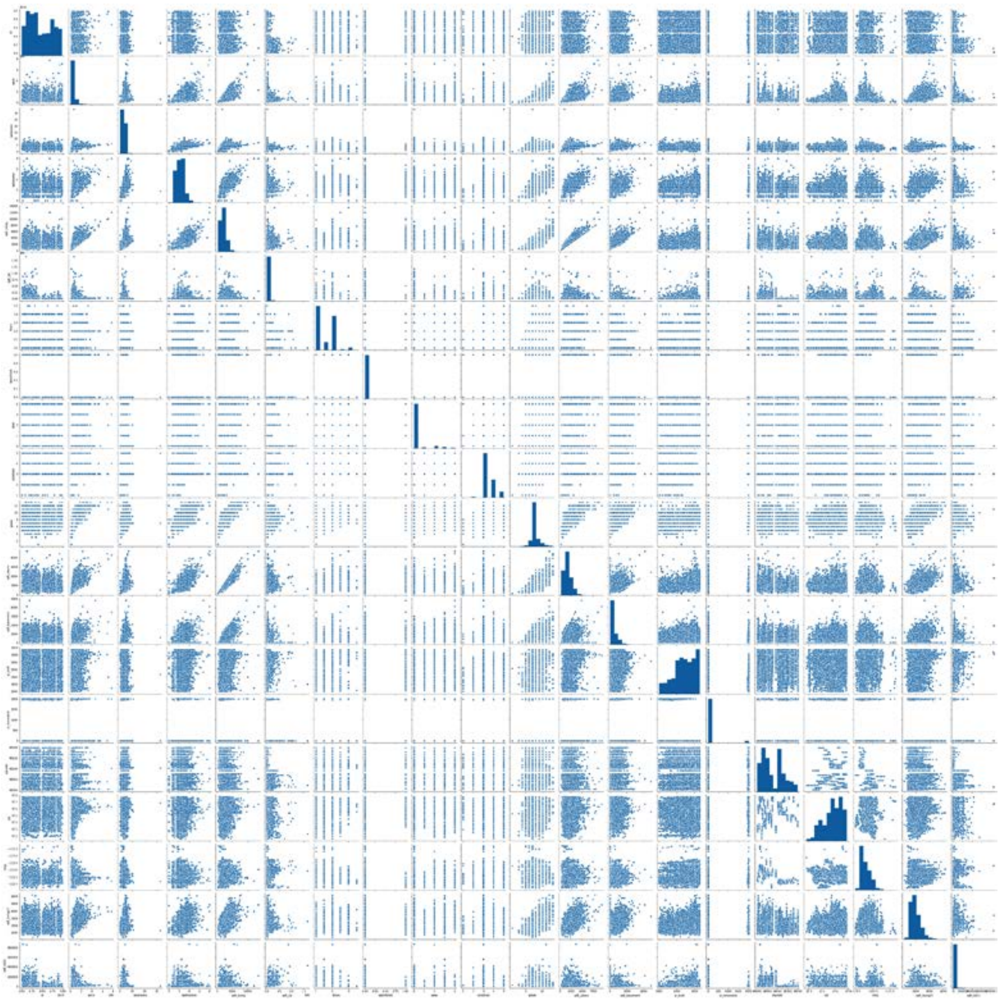


Figure 14.14 – Full DataFrame pairs plot

As you can see, this is a fairly intense, hard-to-read, almost-impossible-to-analyze graph. You'd have to zoom in to each of the paired figures to make some determinations. It should also be mentioned that it takes time for the algorithm to run and produce this graphic. *It takes some processing power to create something so complex from such a large dataset!*

Do we ever want to see a big plot of graphs such as this one? Actually, yes. If we have a dataset with fewer variables, absolutely! We can also make this more friendly by using other color schemes, for example. This may make identifying trends easier. But let's be clear; this isn't very helpful. What we can do is create a pairs plot of some of the variables we may be interested in. Remember we talked about `sqft_living` and `sqft_above` possibly having a strong positive correlation? We also really do want to compare things to pricing, right? So, let's create a pairs plot using just `sqft_living`, `pricing`, and `sqft_above`. Take a look at the relevant code snippet from our file:

```
coln = ['price','sqft_living','sqft_lot']
sns.pairplot(housing_data[coln], height = 3);
plt.savefig('pairplotting.png',dpi =300)
plt.show()
```

Now, when we run this part of the algorithm, we get the graph shown in the following screenshot. This graph provides the correlations for these three values and we can definitely see some positive correlations happening:

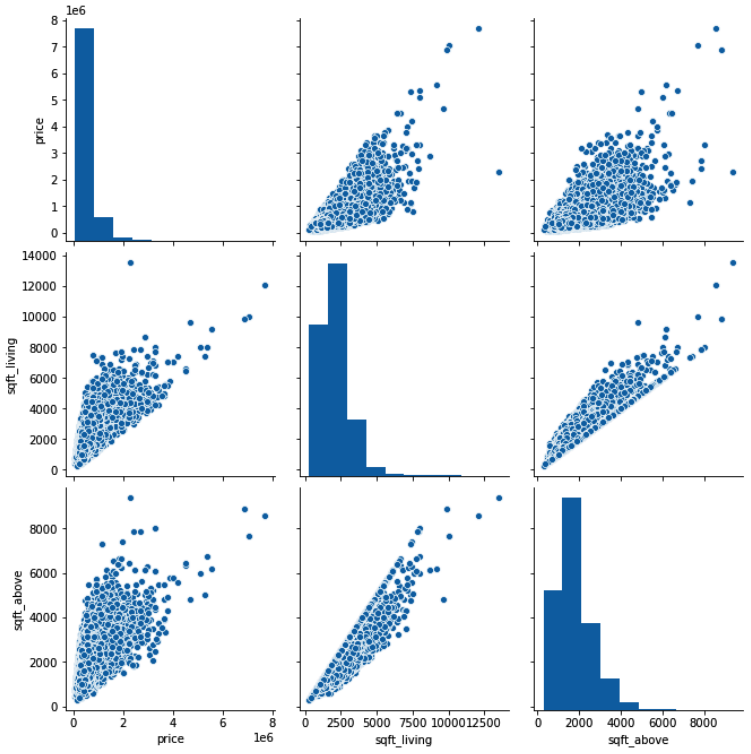


Figure 14.15 – Pairs plot of price, sqft_living, and sqft_above

Notice in particular the graphs that pair **sqft_living** and **sqft_above**. The relationship between those two is fairly linear and positive. This confirms what we observed from *Figure 14.13*, where the correlation was closer to **1** than the other variables.

But it would also help to analyze another three variables so that we can see what happens when the correlation is not strong. We'll keep `price` and `sqft_living` so that we are only changing one of the variables for comparison purposes. Looking at *Figure 14.12*, **sqft_living** and **zipcode** don't seem to have a strong positive correlation at all. So, let's run the algorithm again, swapping `zipcode` with `sqft_above`. Let's take a look at the result shown in the following screenshot:

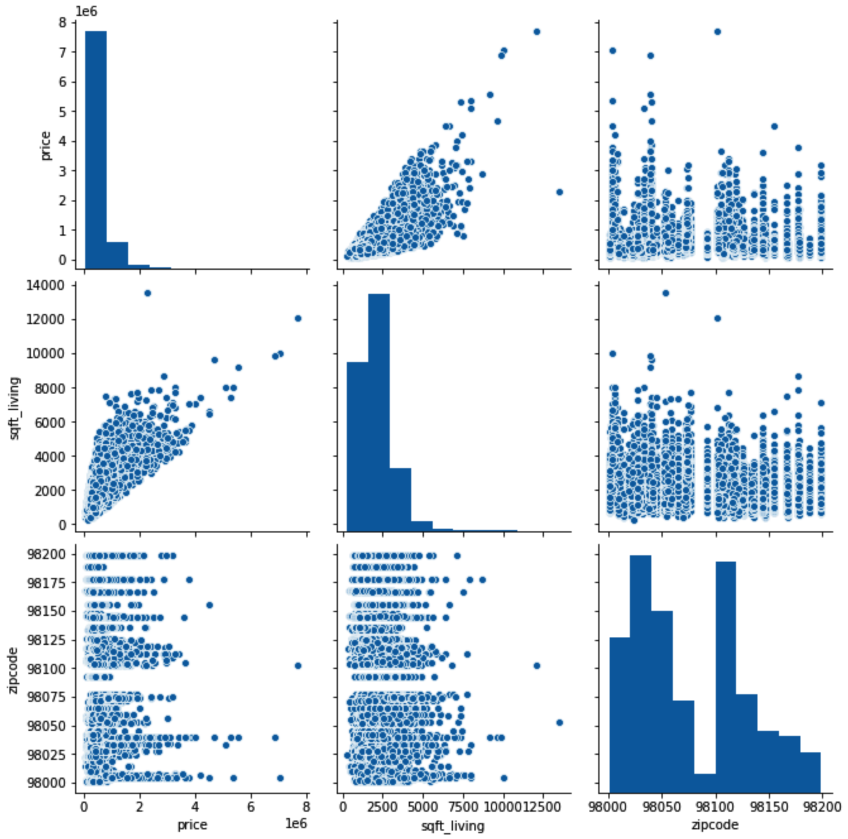


Figure 14.16 – Pairs plot of price, sqft_living, and zipcode

As you can see, **sqft_living** and **zipcode** show no correlation at all. They look more like bar graphs, with no diagonal in sight. Before we move on from these plots, it's worth mentioning that these pairs plots only provide scatterplots for the compared variables and histograms for each of the variables in the plots. If we wanted to go deeper, we could look at other visualization tools, such as those in **Seaborn**.

We're pausing this analysis here. We've used visualization to understand where our data has correlations. If we were to take this problem further, we could use the data to create training models and help us in predictions. Even with the data we have from the graphics, we can see our correlations and make predictions using that. If we had `sqft_living`, we could predict `sqft_above`, for example, because of their strong correlation.

Python allows us to look at data in a wide variety of ways. That's one of the great assets of a tool such as this one.

Summary

In this chapter, we've learned about how to work with data in Python and tackled a housing dataset using some of the concepts we learned about in the chapter. We learned about the pandas package and how it helps us organize and prepare data. We also learned about the need to preprocess datasets, especially in very large datasets. We worked through missing and noisy data, as well as data transformation and the reduction of data. We also learned how to use visualization, creating plots for our datasets that can aid us in identifying correlations and trends.

The topics in this chapter are pretty broad, with entire books written about them. But we felt it important to share some of the capabilities of the Python programming language before moving on to the next two chapters of the book.

In the next chapters, we will focus entirely on applications, using problem scenarios and topics to share some exciting applications of Python and computational thinking in designing algorithms.

15
Applied Computational Thinking Problems

In this chapter, we will be providing examples in multiple areas for applications of the Python programming language and computational thinking. We will be exploring multiple areas, such as humanities, linguistics, cryptography, and more. We will use what we've learned so far about **computational thinking** and the **Python** programming language to do the following:

- Analyze historical speeches
- Write stories
- Calculate text readability
- Find the most efficient route
- Implement cryptography
- Implement cybersecurity
- Create a chatbot

This chapter is unlike the others, as we will be focused exclusively on presenting the problems and providing algorithmic solutions after evaluating each scenario.

Technical requirements

You will need the latest version of Python installed to run the code in this chapter.

You will need the following libraries and packages installed for Python:

- **NLTK**
- **Cairos**
- **Pandas**
- **Matplotlib**
- **Seaborn**

You can find the full source code used in this chapter here: `https://github.com/PacktPublishing/Applied-Computational-Thinking-with-Python/tree/master/Chapter15`

Problem 1 – Using Python to analyze historical speeches

History is quite fascinating, and there are many reasons why we would look at writing algorithms to evaluate historical data and contexts.

For this problem, we want to analyze some historical text. In particular, we are going to take a look at *Abraham Lincoln's second inaugural speech*. Our goal is to find some frequencies of words. There are many reasons why we'd want to perform some straightforward text analysis, especially for historical texts. We may want to compare them, understand underlying themes, and so on.

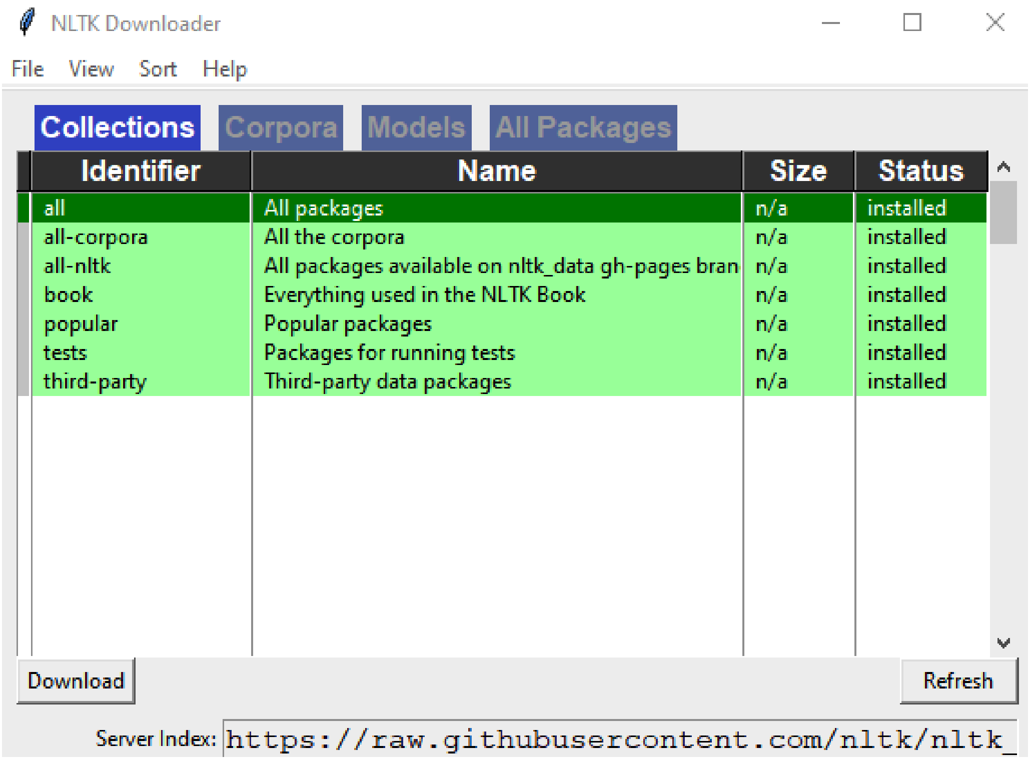
For our algorithm, we are going to use a fairly simple design using the `nltk` package. Because the installation of some of the components is a bit different to what we've done so far, we'll provide some information in case your packages have not been installed.

In the Python shell, if you are in the active console, create a new file and import `nltk` after installing the main package (using `pip install nltk`).

Please note you should not be in the active **Shell** window. If you see >>> at the beginning of the line, click the **File** | **New File** option and then enter the following code instruction line for your empty shell:

```
import nltk
nltk.download()
```

Notice from the preceding code that you'll also open the downloader in `nltk`. The preceding code will bring up a window as shown in the following screenshot (please note that the `nltk` library takes approximately 7 MB of memory, while installing the additional packages will require memory as well, with each ranging from a few KB to 14 to 15 MB):



Identifier	Name	Size	Status
all	All packages	n/a	installed
all-corpora	All the corpora	n/a	installed
all-nltk	All packages available on nltk_data gh-pages bran	n/a	installed
book	Everything used in the NLTK Book	n/a	installed
popular	Popular packages	n/a	installed
tests	Packages for running tests	n/a	installed
third-party	Third-party data packages	n/a	installed

Server Index: https://raw.githubusercontent.com/nltk/nltk_

Figure 15.1 – NLTK Downloader

As you can see, my packages are all installed. If yours are not, select **all**, and then click the **Download** button at the bottom left-hand corner of that window. Once the packages are installed, you can close the window.

Because our problem is fairly simple, we'll skip much of the computational thinking process for this particular section. We are only trying to get frequencies of words used in the speech. So let's dive directly into the algorithm and see how we'll use the `nltk` package to get what we need, including a visual representation of the data:

1. First, we'll need to import `nltk` and the `word_tokenize` function. The `word_tokenize` function allows us to divide the speech into individual words and/or punctuation marks. We'll need the speech text. In this case, the speech is copied into the algorithm. You could potentially import the file into the algorithm and do it that way.

   The `sent_tokenize` function is short for sentence tokenization. In the same way as word tokenization breaks down the text by words and punctuation marks, the sentence tokenization function allows breaking the text into full sentences. The output would contain sentences in a list separated by commas.

   > **Important Note:**
   >
   > It is important to know that all quotation marks have been escaped using `\'` or `\"`, respectively, in order to maintain the original text without creating an error in our code.

   The following algorithm contains everything we need in order to do the analysis of Abraham Lincoln's second inaugural speech:

ch15_historicalTextAnalysis.py

```
import nltk
from nltk.tokenize import sent_tokenize, word_tokenize
```

   The entire text of the speech is contained in the GitHub repository file. For length purposes, we've only included half of that text here. Please note that the output we'll share after the algorithm and explanations will correspond to the truncated speech, but the visual plot will include the data from the entire speech. The `[…]`, seen in the following code at the end of the `text` definition, is simply used to show that there is additional text that goes there:

```
text = 'Fellow-Countrymen: At this second appearing
to take the oath of the Presidential office there is
less occasion for an extended address than there was
at the first. Then a statement somewhat in detail of a
course to be pursued seemed fitting and proper. Now,
at the expiration of four years, during which public
declarations have been constantly called forth on every
```

```
point and phase of the great contest which still absorbs
the attention and engrosses the energies of the nation,
little that is new could be presented. The progress
of our arms, upon which all else chiefly depends, is
as well known to the public as to myself, and it is, I
trust, reasonably satisfactory and encouraging to all.
With high hope for the future, no prediction in regard
to it is ventured. On the occasion corresponding to this
four years ago all thoughts were anxiously directed to
an impending civil war. All dreaded it, all sought to
avert it. While the inaugural address was being delivered
from this place, devoted altogether to saving the Union
without war, urgent agents were in the city seeking to
destroy it without war—seeking to dissolve the Union and
divide effects by negotiation. Both parties deprecated
war, but one of them would make war rather than let the
nation survive, and the other would accept war rather
than let it perish, and the war came. One-eighth of the
whole population were colored slaves, not distributed
generally over the Union, but localized in the southern
part of it. These slaves constituted a peculiar and
powerful interest. All knew that this interest was
somehow the cause of the war. To strengthen, perpetuate,
and extend this interest was the object for which the
insurgents would rend the Union even by war, while the
Government claimed no right to do more than to restrict
the territorial enlargement of it. Neither party expected
for the war the magnitude or the duration which it has
already attained. […]'
```

2. Now that we've defined the text that we want analyzed, as shown in the preceding code snippet, we can tell the algorithm that we want to *tokenize* the text, that is, we want it divided into words. The algorithm will print out a list that contains each individual word or punctuation symbol separated by a comma, as shown in the following code:

```
tokenized_word = word_tokenize(text)
print(tokenized_word)
```

3. After we have our list of words, we want to get the frequency distribution of the words. To do so, we'll import the package from `nltk.probability`, as shown in the following code snippet:

```
from nltk.probability import FreqDist
fdist = FreqDist(tokenized_word)
print(fdist)
fdist.most_common(2)
```

4. Once we have the distribution, we'll want a visual plot of this data, so we'll use `matplotlib` to create our distribution plot, as shown in the following code snippet:

```
import matplotlib.pyplot as plt
fdist.plot(30, cumulative = False)
plt.show()
```

That's the entire algorithm we'll need. When we run the algorithm, this is what our output looks like:

```
['Fellow-Countrymen', ':', 'At', 'this', 'second',
'appearing', 'to', 'take', 'the', 'oath', 'of', 'the',
'Presidential', 'office', 'there', 'is', 'less',
'occasion', 'for', 'an', 'extended', 'address',
'than', 'there', 'was', 'at', 'the', 'first', '.',
'Then', 'a', 'statement', 'somewhat', 'in', 'detail',
'of', 'a', 'course', 'to', 'be', 'pursued', 'seemed',
'fitting', 'and', 'proper', '.', 'Now', ',', 'at',
'the', 'expiration', 'of', 'four', 'years', ',',
'during', 'which', 'public', 'declarations', 'have',
'been', 'constantly', 'called', 'forth', 'on', 'every',
'point', 'and', 'phase', 'of', 'the', 'great', 'contest',
'which', 'still', 'absorbs', 'the', 'attention', 'and',
'engrosses', 'the', 'energies', 'of', 'the', 'nation',
',', 'little', 'that', 'is', 'new', 'could', 'be',
'presented', '.', 'The', 'progress', 'of', 'our',
'arms', ',', 'upon', 'which', 'all', 'else', 'chiefly',
'depends', ',', 'is', 'as', 'well', 'known', 'to', 'the',
'public', 'as', 'to', 'myself', ',', 'and', 'it', 'is',
',', 'I', 'trust', ',', 'reasonably', 'satisfactory',
'and', 'encouraging', 'to', 'all', '.', 'With', 'high',
'hope', 'for', 'the', 'future', ',', 'no', 'prediction',
'in', 'regard', 'to', 'it', 'is', 'ventured', '.', 'On',
'the', 'occasion', 'corresponding', 'to', 'this', 'four',
'years', 'ago', 'all', 'thoughts', 'were', 'anxiously',
'directed', 'to', 'an', 'impending', 'civil', 'war',
'.', 'All', 'dreaded', 'it', ',', 'all', 'sought',
'to', 'avert', 'it', '.', 'While', 'the', 'inaugural',
'address', 'was', 'being', 'delivered', 'from', 'this',
'place', ',', 'devoted', 'altogether', 'to', 'saving',
'the', 'Union', 'without', 'war', ',', 'urgent',
'agents', 'were', 'in', 'the', 'city', 'seeking',
'to', 'destroy', 'it', 'without', 'war—seeking',
'to', 'dissolve', 'the', 'Union', 'and', 'divide',
'effects', 'by', 'negotiation', '.', 'Both', 'parties',
'deprecated', 'war', ',', 'but', 'one', 'of', 'them',
'would', 'make', 'war', 'rather', 'than', 'let', 'the',
'nation', 'survive', ',', 'and', 'the', 'other', 'would',
'accept', 'war', 'rather', 'than', 'let', 'it', 'perish',
',', 'and', 'the', 'war', 'came', '.', 'One-eighth',
'of', 'the', 'whole', 'population', 'were', 'colored',
'slaves', ',', 'not', 'distributed', 'generally', 'over',
'the', 'Union', ',', 'but', 'localized', 'in', 'the',
'southern', 'part', 'of', 'it', '.', 'These', 'slaves',
'constituted', 'a', 'peculiar', 'and', 'powerful',
'interest', '.', 'All', 'knew', 'that', 'this',
'interest', 'was', 'somehow', 'the', 'cause', 'of',
'the', 'war', '.', 'To', 'strengthen', ',', 'perpetuate',
',', 'and', 'extend', 'this', 'interest', 'was', 'the',
'object', 'for', 'which', 'the', 'insurgents', 'would',
'rend', 'the', 'Union', 'even', 'by', 'war', ',',
'while', 'the', 'Government', 'claimed', 'no', 'right',
'to', 'do', 'more', 'than', 'to', 'restrict', 'the',
'territorial', 'enlargement', 'of', 'it', '.', 'Neither',
'party', 'expected', 'for', 'the', 'war', 'the',
'magnitude', 'or', 'the', 'duration', 'which', 'it',
'has', 'already', 'attained', '.']
```

5. Recall that `word tokenization` only included the truncated text. However, the frequency information and the plot that follow are for the entire speech. The `ch15_historicalTextAnalysis.py` GitHub file includes the full speech:

```
<FreqDist with 365 samples and 782 outcomes>
```

The following screenshot shows the frequency distribution visual plot for this algorithm:

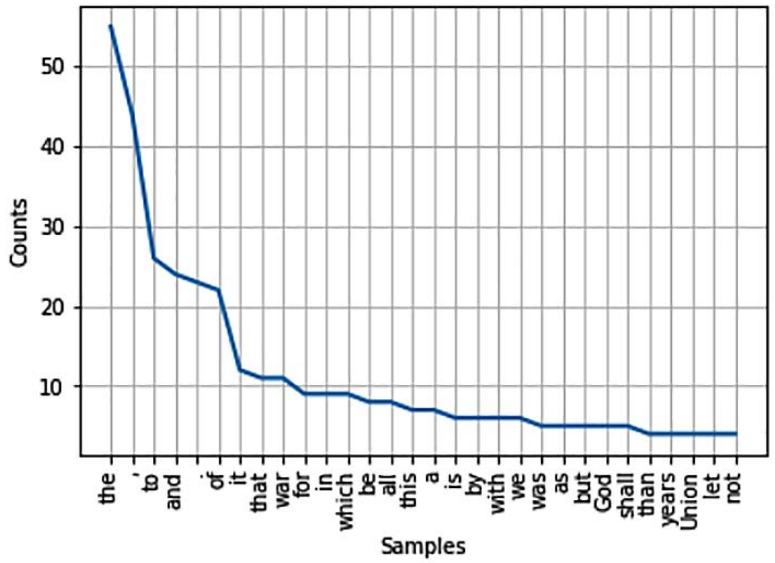


Figure 15.2 – Frequency distribution plot for Abraham Lincoln's second inaugural address

Once we have this information, we can start to look more closely at the words that were used most frequently. When working with this kind of analysis, you may want to consider removing some of the words, such as **to**, **a**, and **the**, as they would be irrelevant to our analysis. However, words such as **years** and **Union** may be relevant to our analysis.

There are many adjustments that can be made for this algorithm, but for now, we've managed to at least get a frequency distribution plot for a historical speech. Now, we'll move on to the next problem.

Problem 2 – Using Python to write stories

Let's look at a fairly simple problem. In this section, we want to create an algorithm that produces a story based on input from a user. We can make this as simple as we want, or add some options. But let's dig into what this is.

Defining, decomposing, and planning a story

First of all, *what is it we're trying to create?* Well, a story. Because of the nature of this problem, we're going to start in reverse, with a sample of the output we want to achieve, that is, a sample story. Let's take a look at a quick story generated by our algorithm before we actually get into the algorithm:

```
There once was a citizen in the town of Narnia, whose name was
Malena. Malena loved to hang with their trusty dog, King Kong.
```

```
You could always see them strolling through the market in the
morning, wearing their favorite blue attire.
```

The preceding output was created by an algorithm that substituted names, locations, time of day, pet, and pet name. It's a short story, but this is something that can be used in much wider applications, such as using input to write social media posts, and filling in information for things such as invitations, forms, and more.

So let's work backward a bit to write our algorithm. *And why did I start at the end this time?* Well, in this case, we know what we want from the end result. You could write your story. You could have an example of that wedding invitation template you need to fill in, or the form. Now we have to figure out how to get the input and then output what we wanted.

From the story shown, here are the things we can get original input for:

- Character name
- Town name
- Type of pet
- Name of pet
- Part of town visited
- Time of day
- Favorite color

When we write our algorithm, we'll need to get all the aforementioned inputs. Let's take a look at the algorithm found in the `ch15_storyTime.py` file:

1. We will need inputs from the user, so we want to use a `print` statement and input requests that include instructions for what is needed:

   ```
   print('Help me write a story by answering some questions.
   ')
   ```

```
name = input('What name would you like to be known by? ')
location = input('What is your favorite city, real or
imaginary? ')
time = input('Is this happening in the morning or
afternoon? ')
color = input('What is your favorite color? ')
town_spot = input('Are you going to the market, the
library, or the park? ')
pet = input('What kind of pet would you like as your
companion? ')
pet_name = input('What is your pet\'s name? ')
```

The preceding code snippet grabs all the inputs from the user so we can write our story.

2. Once we have those, we have to `print` our story. Notice that we wrote it in simple terms, using `%s` so we could replace it with the corresponding inputs. We also used backslashes so that we can see our code on multiple lines, rather than have it in one long line:

```
print('There once was a citizen in the town of %s, whose
name was %s. %s loved to hang \
with their trusty %s, %s.' % (location, name, name, pet,
pet_name))
print('You could always see them strolling through the %s
\
in the %s, wearing their favorite %s attire.' % (town_
spot, time, color))
```

Let's run that code one more time and see what our story says now:

```
Help me write a story by answering some questions.
What name would you like to be known by? Azabache
What is your favorite city, real or imaginary? Rincon
Is this happening in the morning or afternoon? afternoon
What is your favorite color? magenta
Are you going to the market, the library, or the park?
library
What kind of pet would you like as your companion? dog
What is your pet's name? Luna
There once was a citizen in the town of Rincon, whose
name was Azabache. Azabache loved to hang with their
```

```
trusty dog, Luna.
You could always see them strolling through the library
in the afternoon, wearing their favorite magenta attire.
```

Notice that details such as character and settings have changed. In an education learning environment, such a simple algorithm can be a great tool for showing students how to interact with stories and identify key information in them.

While this is a three-sentence story, these algorithms can be much more complex, providing an opportunity to write fantastic original stories with user input. If you wanted to try some of this out, you could even put conditions for which phrases you would use based on some of the input, such as changing the sentences used based on the length of the name entered, for example. *Have some fun with the code and writing some stories!*

Problem 3 – Using Python to calculate text readability

In this section, we'll look at an application relating to linguistics, specifically the readability level of any text. We will be using Martin Luther King's **I Have a Dream** speech in the code snippets that follow. You can substitute this for any text file, so long as you change the location of the file and filename to be accurately reflected in the code. The full code can be found in the `ch15_Readability.py` file.

Before we get into the code, let's talk first about what we're looking for and why it's important. Learning about the readability of texts can help us make decisions about whether or not to include them in a presentation, a school grade level, and much more. The **Flesch-Kincaid score** is used to determine readability and was developed in the 1940s.

Rudolf Flesch created it when working as a consultant with the Associated Press in an effort to improve the readability of newspapers. Originally called the **Flesch Reading Ease**, it was modernized into what is currently used by the US Navy. Rather than getting a score that then had to be converted into a grade level, the Flesch-Kincaid score now provides a grade-level score.

While we won't be using the formula, it is important to know the background of what we're using. The Flesch Reading Ease formula is given by the following:

$$206.835 - 1.015\left(\frac{\text{total words}}{\text{total sentences}}\right) - 84.6\left(\frac{\text{total syllables}}{\text{total words}}\right)$$

The Flesch-Kincaid grade-level formula is given by the following:

$$0.39\left(\frac{\text{total words}}{\text{total sentences}}\right) + 11.8\left(\frac{\text{total syllables}}{\text{total words}}\right) - 15.59$$

The thing about the preceding formulas is that they exist in the readability package available for Python. If we import the package, we are able to perform a readability analysis with a fairly simple bit of code.

Let's take a look at the code we'll need in order to perform a readability analysis for the Martin Luther King speech:

1. Firstly, remember to change the path for your file in the code and then import the necessary packages for the code:

ch15_readability.py

```
from readability import Readability
text = open('C:\\...\\ch15_MLK-IHaveADream.txt')
text_up = text.read()

r = Readability(text_up)
flesch_kincaidR = r.flesch_kincaid()
```

From the preceding code, you'll see that we imported the `readability` package into our program. If you need to install the library/package, you can use `pip install readability` to do so.

Once we have the requisite libraries, we can open the file we want to analyze. We also want to tell the algorithm to read the text, which I've called `text_up` here for text upload, so I don't forget that I am reading an open file. This is the text we've opened from the file location in the preceding code. Finally, we ask the program to analyze the text using the `Readability` function. Notice that we saved that to `r`.

2. After we've done all of that, we can `print` our grade level with the following code snippet:

```
print('The text has a grade '+ flesch_kincaidR.grade_
level + ' readability level.')
```

When we run our algorithm, our output is fairly simple as well. Take a look at the following output:

```
The text has a grade 9 readability level.
```

Now that you know how to verify the readability of any text, try to perform the analysis on other types of texts, including poems, stories, speeches, songs, and more.

Problem 4 – Using Python to find most efficient route

For this problem, when learning about algorithms, we will use a common algorithm—the **Travelling Salesman Problem** (**TSP**). Let's set up the problem itself.

A salesman needs to travel to a set number of cities or locations. Let's say the salesman has 10 locations to go to. They could go to those 10 locations in a lot of different orders. Our goal with this algorithm is to create the best possible, most efficient route to hit those locations.

Note that for this particular scenario, as we will do in the next problem, we'll employ straightforward analysis by using the four elements of the computational thinking process.

Defining the problem (TSP)

This problem is a little more complex than how it appears initially. Think of it this way. If we have 10 destinations and we are calculating round-trip permutations to check the fastest routes, we're left with more than 300,000 possible permutations and combinations. As a reminder, a permutation takes order into consideration, while a combination does not.

For example, the numbers *3344* and *3434* are two different permutations. However, they are only counted as one combination because the order of the numbers does not matter.

But back to our problem. All we need to know is that we want to create an algorithm that will take us to our destinations in the most efficient way. We have to identify the cities to be visited and identify the way we'll travel as follows:

- There are five cities in total, namely, **New York** (**NYC**), **Philadelphia**, **Baltimore**, **Chicago**, and **Cleveland**.
- We will use one vehicle because we're using TSP instead of a **vehicle routing problem** (**VRP**).
- The first city is 0, which is NYC. The distance between NYC and itself is 0.

Now let's look at the pattern.

Recognizing the pattern (TSP)

For each city, there will be a total of five distances, with the distance to itself equal to 0. We are going to need an array, or lists, for all the distances for each city. We will need to create a model in order to access the data in our algorithm. We'll look at that while designing the algorithm. First, let's talk about generalizing the pattern a bit.

Generalizing (TSP)

For this particular problem, we'll enter the cities manually into the algorithm itself. One thing you may want to consider is how to get input from a user in order to create the arrays necessary with the distances.

You could also create a database for the distances between major cities that you can access from a `.csv` file so that the data for cities that a person inputs can be found there, which can then be added to our model. There are many additions to this particular algorithm and this isn't a problem that can be solved in just one way. For now, we're going to stay with a defined set of cities so that we can create our algorithm.

As an aside, we have referred to the source code from `https://developers.google.com/optimization/routing/tsp` for this problem.

Designing the algorithm (TSP)

It's time to see what we've been talking about. Let's start with NYC and construct that array first. The other arrays are created in the same way. All distances are in miles and have been approximated and rounded based on **Google Maps** data, given as follows:

- Distance from NYC to NYC is 0.
- Distance from NYC to Philadelphia is 95.
- Distance from NYC to Baltimore is 192.
- Distance from NYC to Chicago is 789.
- Distance from NYC to Cleveland is 462.

The following table shows the distances from each city to each other and itself:

	NYC	Philadelphia	Baltimore	Chicago	Cleveland
NYC	0	95	192	789	462
Philadelphia	95	0	105	759	431
Baltimore	192	105	0	701	374
Chicago	789	759	701	0	344
Cleveland	462	431	374	344	0

Table 15.1 – Distances from one city to another

So, as you can in the preceding table, if we were to write these distances as an array, we would use the following code:

```
[0, 95, 192, 789, 462]
```

For Philadelphia, we would have the following array:

```
[95, 0, 105, 759, 431]
```

For Baltimore, we would have the following array:

```
[192, 105, 0, 701, 374]
```

For Chicago, we would have the following array:

```
[789, 759, 701, 0, 344]
```

And finally, for Cleveland, we would have the following array:

```
[462, 431, 374, 344, 0]
```

Note that we will give indexes to each of the cities in order to identify them. NYC is *0*, Philadelphia is *1*, Baltimore is *2*, Chicago is *3*, and Cleveland is *4*. Let's see what the algorithm looks like for this problem (note that the **OR-Tools library** is used to optimize vehicle routes, linear programming, constraint programming, and more):

1. First, let's start by importing the packages and libraries we'll need. The full file for this algorithm is `ch15_travel.py` and available on GitHub:

   ```
   from ortools.constraint_solver import routing_enums_pb2
   from ortools.constraint_solver import pywrapcp
   ```

Remember that this algorithm would need to get a new distance matrix if you are planning to visit more cities and/or different cities. That is the only piece of the code you'd need to alter. The snippet of code that you will need to adjust each time is the matrix under `create_data_model()`, as shown in the following code snippet:

```
#Create data model.
def create_data_model():
    data = {}
    data['distance_matrix'] = [
        [0, 95, 192, 789, 462],
        [95, 0, 105, 759, 431],
        [192, 105, 0, 701, 374],
        [789, 759, 701, 0, 344],
        [462, 431, 374, 344, 0],
    ]
    data['num_vehicles'] = 1
    data['depot'] = 0
    return data
```

2. After we've defined our data model, we'll need to print a solution. The following function provides that information:

```
#Provide solution as output - print to console
def print_solution(manager, routing, solution):
    print('Objective: {} miles'.format(solution.
ObjectiveValue()))
    index = routing.Start(0)
    plan_output = 'Route for vehicle 0:\n'
    route_distance = 0
    while not routing.IsEnd(index):
        plan_output += ' {} ->'.format(manager.
IndexToNode(index))
        previous_index = index
        index = solution.Value(routing.NextVar(index))
        route_distance += routing.
GetArcCostForVehicle(previous_index, index, 0)
    plan_output += ' {}\n'.format(manager.
IndexToNode(index))
    print(plan_output)
    plan_output += 'Route distance: {}miles\n'.
format(route_distance)
```

As you can see from the preceding code, we are creating a function so that we can print the solutions based on our arrays and the distances in those arrays. Recall that you will identify the point of origin, that is, the city you're leaving from. Then we run the algorithm to gather the information and create our `print` statement.

3. Finally, we'll need to define our `main()` function in order to run our algorithm. The `main()` function tells the algorithm to go ahead and create that data model we had defined, and then store it as data. We then create the routing model to find our solution. Take a look at the following code snippet:

```
def main():
    data = create_data_model()
    manager = pywrapcp.
RoutingIndexManager(len(data['distance_matrix']),
                                               data['num_
vehicles'], data['depot'])

    # Create Routing Model.
    routing = pywrapcp.RoutingModel(manager)

    def distance_callback(from_index, to_index):
        """Returns the distance between the two nodes."""
        # Convert from routing variable Index to distance
matrix NodeIndex.
        from_node = manager.IndexToNode(from_index)
        to_node = manager.IndexToNode(to_index)
        return data['distance_matrix'][from_node][to_
node]

    transit_callback_index = routing.
RegisterTransitCallback(distance_callback)

    routing.SetArcCostEvaluatorOfAllVehicles(transit_
callback_index)

    search_parameters = pywrapcp.
DefaultRoutingSearchParameters()
```

```
    search_parameters.first_solution_strategy = (
        routing_enums_pb2.FirstSolutionStrategy.PATH_
CHEAPEST_ARC)

    solution = routing.SolveWithParameters(search_
parameters)

    if solution:
        print_solution(manager, routing, solution)

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

The preceding code shows how we define our `main()` function. As a note, a `main()` function can be named anything we want. When using multiple functions, we sometimes use `main()` to identify the function that will be outputting what we originally wanted from the algorithm. For this problem, we're creating a `main()` function that will identify the best route for our travels.

4. Now let's take a look at what we get for our output when we run this code. The code provides us with the `Objective` total of miles and the route we should take for the trip. Here's the output:

```
Objective: 1707 miles
Route for vehicle 0:
 0 -> 1 -> 2 -> 4 -> 3 -> 0
```

As you can see, our trip from NYC and going back to NYC would be most efficient if we followed the route in this order: NYC | Philadelphia | Baltimore | Cleveland | Chicago | NYC.

This is not the only approach for the travel problem. It is also not necessarily the most user-friendly approach if we wanted to run this multiple times a day, for example, for different travelers. To do that, you'd want to automate a lot more of this, as mentioned earlier in the example. Some things you might consider are as follows:

- Being able to input cities
- Having a calculator that grabs information to determine distances
- Using an automated process for creating the distance matrix

But for now, you've seen a TSP in action! We will look at a new problem in the next section.

Problem 5 – Using Python for cryptography

Cryptography is what we use to code and decode messages. We used a simple **Caesar cipher** in *Chapter 9*, *Understanding Input and Output to Design a Solution Algorithm*. For this problem, we're going to use some of the packages available in Python to encrypt and decode information.

Note that for this particular scenario, we'll use the straightforward analysis from using the four elements of the computational thinking process. While we don't always follow them exactly, this particular problem lends itself to a fairly straightforward use.

Defining the problem (cryptography)

You are working on a classified project and need to encrypt your information to maintain its safety.

Recognizing the pattern (cryptography)

Python has a cryptography package that can be installed, much like when we installed other libraries, such as **Pandas** and **NumPy**. In our problem, one of the main things we need to know is that we may need to continue to encrypt messages. We may also want to decode messages that we receive, but we are going to focus on the encryption side of things first.

Generalizing (cryptography)

When we design our algorithm, we'll want something we can continue to use throughout the life of our project with little effort. That is, any time we want to encrypt a new message, we can run the algorithm and input the message rather than add the message itself to the algorithm body every time. This is the generalized pattern for our particular problem. *That means we're ready to design.*

Designing the algorithm (cryptography)

To write our algorithm, let's first take a look at what we'll need to do:

1. Define the letters.
2. Change all the letters to lowercase to run our algorithm.
3. Define the required functions—`encryption`, `decoding`, and `main`.
4. Call the cryptography `main` function.

> **Tip:**
>
> The full algorithm for this problem can be found in the `ch15_cryptographyA.py` file.

We will start designing our algorithm by following these steps:

1. Let's begin by defining our letters. The following code snippet defines our letters, and then renders each letter in lowercase:

```
LETTERS = 
'ABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZ'
LETTERS = LETTERS.lower()
```

2. Next, we define our encryption function. This function will take two arguments—`message` and `key`. The `message` function will be user-defined, which will be done in the `main` function. For now, we'll use an empty message by adding empty quotes (`''`) as the definition of the `encryptedM` variable, as shown in the following code snippet:

```
def encrypt(message, key):
    encryptedM = ''
    for letts in message:
        if letts in LETTERS:
            num = LETTERS.find(letts)
            num += key
            encryptedM +=  LETTERS[num]

    return encryptedM
```

Notice that we iterate over the letters in the message we want to encrypt, and then we use the key that the user defines in the `main` function and encrypt the message. This function then returns the encrypted message.

But why haven't we defined the `main` *function yet if that's where we are getting the inputs?* Because the `main` function will require the other two functions to either encrypt or decode the message. So bear with us; we'll get to that `main` function soon.

3. Now let's take a look at the decoding function. This is what we'll use when we have an encrypted message and want to know what the original message was:

```
def decode(message, key):
    decodedM = ''
    for chars in message:
        if chars in LETTERS:
            num = LETTERS.find(chars)
            num -= key
            decodedM +=  LETTERS[num]

    return decodedM
```

   The preceding code shows the function that we'll use to decode our messages. It uses the characters in the message and the encryption key to then decode the message. Note that you cannot decode a message if you don't have the original key, unless you have the time to sit and try each one of the keys, that is.

4. Finally, we'll need that `main` function we keep referring to. This is the function that takes all the inputs needed in order for this algorithm to run. Here are the three things necessary for it to run correctly—the message to be encrypted or decoded; the key, which can be any number in the range 1 to 26; and whether we are encrypting or decoding.

   Here is the `main` function:

```
def main():
    message = input('What message do you need to encrypt 
or decrypt? ')
    key = int(input('Enter the key, numbered 1-26: '))
    choice = input('Do you want to encrypt or decode? 
Type E for encrypt or D for decode: ')

    if choice.lower().startswith('e'):
        print(encrypt(message, key))
    else:
        print(decode(message, key))

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

Notice from the preceding code that we defined a `main` function. At the end of the code, we called that function. *Don't forget to call that* `main` *function in the algorithm!* That's how you get the algorithm to run.

Here is a sample output when we try to encrypt the input message, `the name of the dog is King Kong,` using a key of `9` to encrypt the message:

```
What message do you need to encrypt or decrypt? the name of the
dog is King Kong
Enter the key, numbered 1-26: 9
Do you want to encrypt or decode? Type E for encrypt or D for
decode: E
cqnwjvnxocqnmxprbrwpxwp
```

As you can see, we obtain the encrypted text, `cqnwjvnxocqnmxprbrwpxwp,` as the cipher text, and now we've created an algorithm that can encrypt or decode any message. Let's now move on to a new problem.

Problem 6 – Using Python in cybersecurity

For this problem, we've decided to perform a fairly short cybersecurity check. First, let's talk about cybersecurity. The market for cybersecurity is expected to grow by 10% by 2027, according to a *Grand View Research* report.

Translating that to the job market is a little tricky. Currently in the United States, for example, there are more cybersecurity needs for the market than people or jobs available. That job market growth is expected to be slightly above 30% between 2018 and 2028. So learning a bit about cybersecurity and cryptography won't hurt.

For this particular problem, we are going to explore a few things. First, let's talk about **hashing**. In cybersecurity, hashing means those really long strings of numbers and letters that replace things such as passwords. For example, if you entered a password of `password1` (please don't do that, never use `password` as a password), the hashing process would replace it with something that looks more like this:

```
27438d623d9e09d7b0f8083b9178b5bb8ff8bc321fee518af
4466f6aadb68a8f:100133bfdbff492cbc8f5d17af46adab
```

When we are creating cryptography algorithms, we have to add random data, which we call **salts**. Salts just provide additional input and help us make passwords more secure when storing them.

When we use hashing in Python, we can use the uuid library. **UUID** stands for **Universal Unique Identifier**. The uuid library is used when we want to generate random, 128-bit objects as IDs. *But what are we really talking about?* Let's take a look at the algorithm found in the ch15_hashing.py file:

1. We will import libraries first:

```
import uuid
import hashlib
```

   We are importing two libraries that will allow us to save our passwords using salting and hashing.

2. In the next code snippet from the file, we define the function to hash our password:

```
def hash_pwd(password):
    salt = uuid.uuid4().hex
    return hashlib.sha1(salt.encode() + password.
encode()).hexdigest() + ':' + salt
```

   We salted the password using our uuid package, and then returned the hash using the secure hash algorithm 1, sha1. This is only one of the algorithms we can use. We could have used others, such as **SHA-256**, **SHA-384**, and more. The sha1 hash has an output size of 160, while sha256 has an output size of 256. Both sha1 and sha256 have block sizes of 512 bits, while sha384 has a block size of 1,024 bits.

   All of this becomes relevant when choosing the hash we'll use, how secure they are, and so on. We used sha1 here more for *nostalgia*, but it is not as secure as sha256 and sha384. When attacked, sha1 will fail against a prolonged attack. The other two will hold out for longer, but are still not the best ones. Hashes such as **shake128** and **shake256** have more success against such attacks.

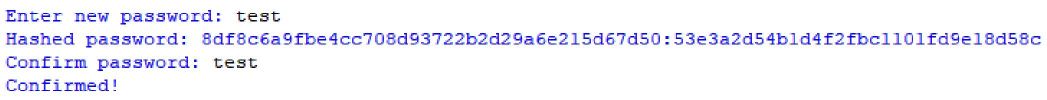
3. Let's now look at the check function. We always want to confirm a password by asking for it to be entered twice. The following snippet of code defines what the algorithm will do when it receives the second password:

```
def check_pwd(hashed_pwd, user_pwd):
    password, salt = hashed_pwd.split(':')
    return password == hashlib.sha1(salt.encode() + user_
pwd.encode()).hexdigest()
```

4. Now let's ask for some input. First, we'll ask for the password. Because we're curious about what the program is doing, we'll print the hashed password, but you can omit that line when we are building this into a site or other application. After that, we ask to verify the password and provide output for the user so they know whether they match, in which case, we'd probably want them to try again. For now, this algorithm either confirms it or lets the user know that it is now confirmed:

```
new_pwd = input('Enter new password: ')
hashed_pwd = hash_pwd(new_pwd)
print('Hashed password: ' + hashed_pwd)
confirm_pwd = input('Confirm password: ')

if check_pwd(hashed_pwd, confirm_pwd):
    print('Confirmed!')
else:
    print('Please try again')
```

After running the program, we get the following output:

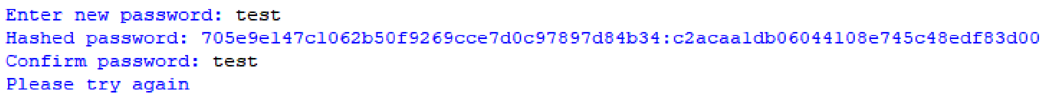


```
Enter new password: test
Hashed password: 8df8c6a9fbe4cc708d93722b2d29a6e215d67d50:53e3a2d54b1d4f2fbc1101fd9e18d58c
Confirm password: test
Confirmed!
```

Figure 15.3 – Output for a salted and hashed password confirmation

As you can see from the preceding screenshot, the password was confirmed by the system.

5. Now let's see what happens when we enter two different passwords. Let's take a look at the following screenshot for that:

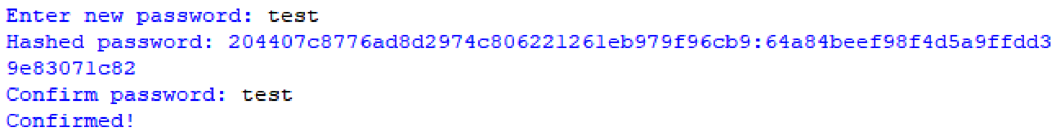


```
Enter new password: test
Hashed password: 705e9e147c1062b50f9269cce7d0c97897d84b34:c2acaa1db06044108e745c48edf83d00
Confirm password: test
Please try again
```

Figure 15.4 – Output for a salted and hashed password with confirmation fail

As you can see, the program asks the user to try again. However, the algorithm doesn't provide a way to do that unless the process is restarted. We can let it live like that or we can add conditions so that the program runs one more time, or two more times, or an infinite number of times, until a confirmation is reached.

6. Now let's take a look at what happens if we run the algorithm using `sha256` instead of `sha1`. The following screenshot shows the result when a password is confirmed using `sha256`:



```
Enter new password: test
Hashed password: 204407c8776ad8d2974c80622126leb979f96cb9:64a84beef98f4d5a9ffdd3
9e83071c82
Confirm password: test
Confirmed!
```

Figure 15.5 – Output when sha1 is replaced with sha256 in the algorithm for confirming a password

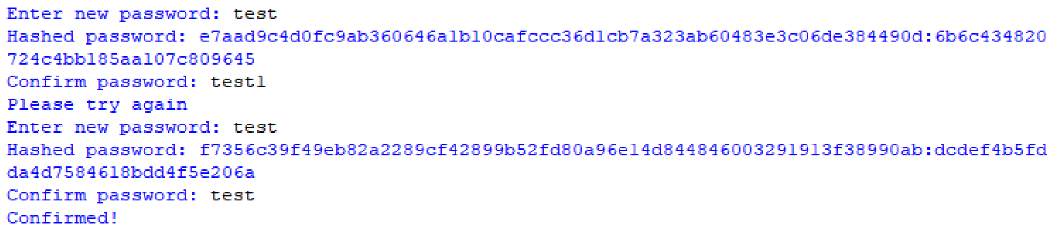
Notice that the hash has a longer length as regards the `sha256` algorithm. When we are working with cryptography, random and long are always helpful. It is easier to crack a password that is not random, such as `password` or `mycat`, than it is to crack a password that is very long and contains random numbers and letters. That's why we try to store data in ways that safeguard it against attacks.

7. Let's take a look at what we could do to provide one more chance for someone to enter the password. At the end of the algorithm, let's add some code after the last line:

```
    new_pwd = input('Enter new password: ')
    hashed_pwd = hash_pwd(new_pwd)
    print('Hashed password: ' + hashed_pwd)
    confirm_pwd = input('Confirm password: ')
    if check_pwd(hashed_pwd, confirm_pwd):
        print('Confirmed!')
    else:
        print('Please try again later')
```

Notice that the last statement in our preceding snippet states `'Please try again later'`. This lets users know that if they want to save a password, they'll have to start the process again. The algorithm has, at that point, stopped.

8. If we place the preceding code after our `else`, `print()` statement, then the algorithm will run one more time. The following screenshot shows the output when a user tries a second time:



```
Enter new password: test
Hashed password: e7aad9c4d0fc9ab360646a1b10cafccc36d1cb7a323ab60483e3c06de384490d:6b6c434820
724c4bb185aa107c809645
Confirm password: test1
Please try again
Enter new password: test
Hashed password: f7356c39f49eb82a2289cf42899b52fd80a96e14d844846003291913f38990ab:dcdef4b5fd
da4d758461 8bdd4f5e206a
Confirm password: test
Confirmed!
```

Figure 15.6 – Output after running an algorithm with an incorrect match first

Before we move on from this example, notice that the hashed passwords provided are different even though our new password entered was `test` on both occasions. As we mentioned, the hashed password is created each time. Otherwise, everyone would know what the passwords were stored because `test` would be the same so long as we'd used the same hash, in this case, `sha256`.

There is a lot more to explore in cybersecurity and cryptography. This is just a taste of how we can encrypt information.

Problem 7 – Using Python to create a chatbot

It's time to create a simple chatbot. You've probably interacted with at least a dozen of these chatbots in the last few years. When you go to some websites, you might get a *person* who wants to chat with you and asks you a few simple questions, such as how you're doing, and what they can help you with. For most of those sites, the *person* is not a person, but a chatbot.

In some instances, the chatbots will then direct you to an actual human being. But most of the time, they'll just answer your questions by pointing you in the direction of the available answers on their website.

We're going to create something like those chatbots here. There are some components we'll need before we get started. One of them is an `intents` file. That file, which should be a `.json` file, contains the greetings and responses that the bot will be using and/or responding to. Here's a sample of what the intents content looks like:

```
{"intents": [
        {"tag": "greeting",
         "patterns": ["Hi", "How are you", "Hello?",
"Welcome!", "Hello"],
         "responses": ["Hello! Thank you for visiting our site!
", "Welcome back!", "Hello, how can I help you?", "What can I
do for you? "],
         "context_set": ""
        },
        {"tag": "goodbye",
         "patterns": ["Bye", "See you later", "Goodbye"],
         "responses": ["See you later, thanks for visiting",
"Thank you and have a wonderful day!", "Bye! See you soon!"]
        }
   ]
}
```

As you can see, this is just a group of possible responses. The more data we give the `.json` file, the more robust and accurate our bot will be.

We should note that the `intents.json` file will need to be edited in a **JSON Editor**. There is an online editor you can use at `https://jsoneditoronline.org` where you can create your own, or edit an existing, file.

Why would we need a bot? There are a wide variety of uses for something like this, from creating and posting messages to social media to providing a customer with a bot that asks whether they require assistance as they visit a web page, for example. These are only some of the things we can do with a chatbot.

Now let's take a look at an algorithm that creates a chatbot. The full file can be found in the repository. We've commented and described what goes on in some of the sections in snippets:

1. Let's start by importing the libraries here:

ch15_chatBot.py

```
import nltk
import json
import pickle
import numpy as np

nltk.download('punkt')
nltk.download('wordnet')
```

```
from nltk.stem import WordNetLemmatizer
lemmatizer = WordNetLemmatizer()
```

From the preceding code, note that you do not have to download the `nltk` modules each time. However, it won't hurt to have this code. The system won't install multiple copies each time; it will just identify that they are there and not install them a second time.

2. Let's continue grabbing what we need from our libraries and packages:

```
from keras.models import Sequential
from keras.optimizers import SGD
from keras.layers import Activation, Dense, Dropout
import random
```

3. Now that we have what we need, we have to look at our `.json` file. That file contains those intents, as mentioned earlier in this problem. We'll need to not only open that file, but also divide the components and sort them in ways that our algorithm can understand. Take a look at the following code snippet:

```
#Upload intents file and create our lists
words=[]
classes = []
doc = []
ignore_words = ['?', '!', ',', '.']
data_words = open(r'C:\...\intents.json').read()
intents = json.loads(data_words)
```

Remember that the program won't run unless you specify the right location for the file you are trying to access, in this case, the `.json` file. Also notice this time that we are opening it in a slightly different manner, as shown in the preceding snippet. Opening the file like this, which is not the same as when we open `.csv` files with Pandas, means we do not need to use the double `\\` in the paths.

4. Now let's tell the algorithm what to do with that file:

```
for intent in intents['intents']:
    for pattern in intent['patterns']:
        #Tokenize all the words (separate them)
        w = nltk.word_tokenize(pattern)
        words.extend(w)
```

```
            #Add all the words into doc
            doc.append((w, intent['tag']))
            #Add the classes
            if intent['tag'] not in classes:
                classes.append(intent['tag'])
print(doc)
```

Here, we are tokenizing our information, that is, we're breaking down everything into words and then adding them to lists. This is what makes it possible to process the information. After we split them up, we are going to group them by what the words mean.

5. The words are then sorted, as can be seen in the following code snippet:

```
#lemmatization

words = [lemmatizer.lemmatize(w.lower()) for w in words 
if w not in ignore_words]
words = sorted(list(set(words)))
classes = sorted(list(set(classes)))

pickle.dump(words,open('words.pkl','wb'))
pickle.dump(classes,open('classes.pkl','wb'))
```

Notice in the preceding code that we used `pickle()`. Pickle is a method in Python that we can use to serialize data (or deserialize it). The method is used to then replace current file data so that it can be used as converted.

6. Now that we have done all of that, we need to create our training model. We won't go through all the sections of that process here, but the entire code can be found in the GitHub repository file. Remember that you first train, then create, and then compile the model.

Once we have run through that process, we save the model so that we can use it. But now let's look at the chatbot functions:

```
#Define chatbot functions
def clean_up_sentence(sentence):
    sentence_words = nltk.word_tokenize(sentence)
    sentence_words = [lemmatizer.lemmatize(word.lower()) 
for word in sentence_words]
    return sentence_words
```

```
def bow(sentence, words, show_details=True):
    sentence_words = clean_up_sentence(sentence)
    bag = [0]*len(words)
    for s in sentence_words:
        for i,w in enumerate(words):
            if w == s:
                bag[i] = 1
    return(np.array(bag))
def predict_class(sentence, model):
    p = bow(sentence, words,show_details=False)
    res = model.predict(np.array([p]))[0]
    ERROR_THRESHOLD = 0.25
    results = [[i,r] for i,r in enumerate(res) if
r>ERROR_THRESHOLD]
    results.sort(key=lambda x: x[1], reverse=True)
    return_list = []
    for r in results:
        return_list.append({"intent": classes[r[0]],
"probability": str(r[1])})
    return return_list
```

The first three functions work to create responses for the chatbot and also make predictions. Those classes will matter for how we get those responses from our bot. Let's think about it this way—if I say hello, I wouldn't want the chatbot to say goodbye. That would be rude. But remember that our bot is only as good as our training is. So if we don't have enough in the `.json` file and do not train the model correctly, the bot will be fairly useless.

7. Now let's define how we get the responses:

```
def getResponse(ints, intents_json):
    tag = ints[0]['intent']
    list_of_intents = intents_json['intents']
    for i in list_of_intents:
        if(i['tag']== tag):
            result = random.choice(i['responses'])
            break
    return result

```

```
def chatbot_response(msg):
    ints = predict_class(msg, model)
    res = getResponse(ints, intents)
    return res
```

As you can see from the preceding code snippet, the bot will craft a response and return it. We'll be calling those things in the next few pieces of code in our file.

8. But we'll skip that here and go into the look of our chatbot:

```
base = Tk()
base.title("Chat with Customer Service")
base.geometry("400x500")
base.resizable(width=FALSE, height=FALSE)
```

Notice that we established some critical information in the preceding snippet. We established the size of the window and blocked it from being resized.

9. In the next code snippet, we'll establish the background, add `scrollbar`, and establish the look of the **Send** button:

```
#Create chatbot window
ChatLog = Text(base, bd=6, bg="white", height="8",
width="70", font="Calibri")
ChatLog.config(state=DISABLED)
#Scrollbar
scrollbar = Scrollbar(base, command=ChatLog.yview,
cursor="arrow")
#Create Send button
SendButton = Button(base, font=("Calibri",12,'bold'),
text="Send", width="15", height=5,
                    bd=0, bg="pink",
activebackground="light green",fg='black',
                    command= send )

EntryBox = Text(base, bd=0, bg="white",width="29",
height="5", font="Arial")

scrollbar.place(x=376,y=6, height=386)
ChatLog.place(x=6,y=6, height=386, width=370)
EntryBox.place(x=128, y=401, height=90, width=265)
```

```
SendButton.place(x=6, y=401, height=90)

base.mainloop()
```

So this is it, in a nutshell. *We created a chatbot! But what does it look like when it runs?* Take a look at the output:

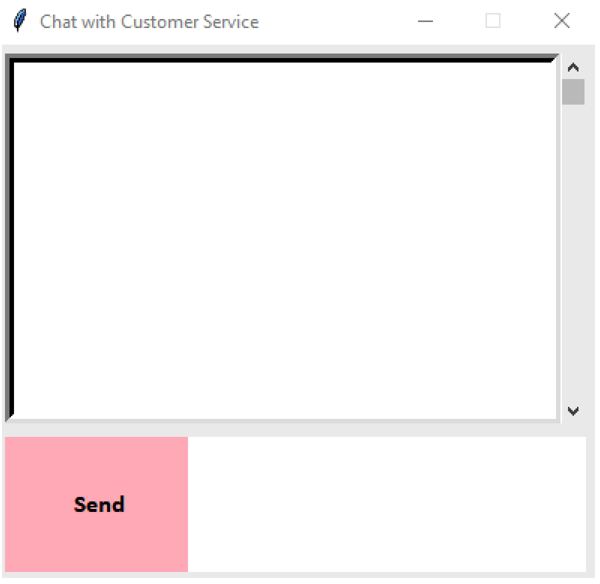


Figure 15.7 – The chatbot window

Notice our components, the scrollbar on the left, the pink **Send** button, and the title of our chatbot. And notice too that the maximize button is grayed out. That's because we said we don't want the window resized.

10. In addition, when we click the **Send** button, we want the user to know whether it was clicked. Otherwise, you could click it multiple times if you are unsure. That's why the active background color is changed in the code. The following screenshot shows what the button looks like when active:

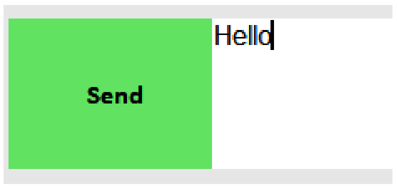
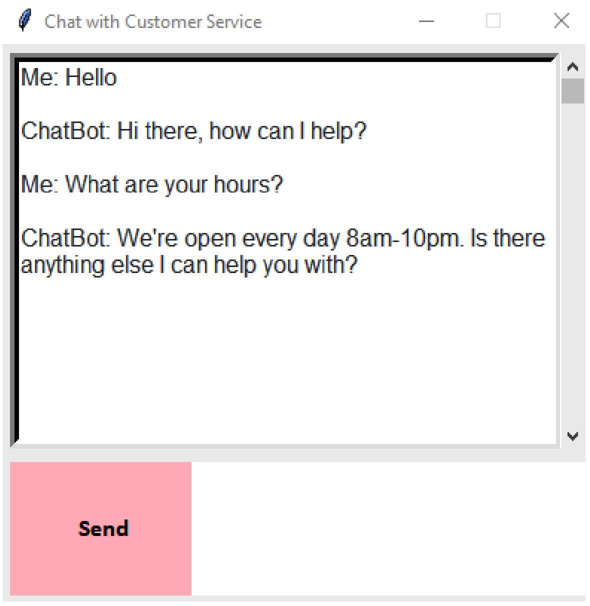


Figure 15.8 – Active Send button

Many chatbots have features like this one so that we avoid errors with the code.

Once we've said hello, the bot will respond. Before we move away from this problem, let's take a look at a quick conversation with the chatbot in the following screenshot:



```
Me: Hello

ChatBot: Hi there, how can I help?

Me: What are your hours?

ChatBot: We're open every day 8am-10pm. Is there
anything else I can help you with?
```

Figure 15.9 – The chatbot window with responses

As you can see, a few lines of code and a file can be used to create an interactive experience with a chatbot.

Feel free to play around with the code to add some flair, create a different `intents.json` file, and make it more relevant to what you need.

Summary

In this chapter, we've had a chance to explore Python in a few but very different applications while looking at actual problems.

In previous chapters, we learned about the computational thinking process, along with the elements of decomposition, pattern recognition, pattern generalization, and algorithm design, which make the algorithms make sense. When we tackle problems from clients or just while creating a script as a hobby, we have to go through the process necessary to define what we're creating with our algorithm. This critical process will ensure that we are designing the best possible algorithms we can.

In this chapter, we learned how to read from files, upload files, create ciphers and decoders, use algorithms to write stories with user input, and develop the most effective travel plan when given the cities we'll visit. In addition, we also created a basic chatbot, which can interact and adapt based on the user input.

In the next chapter, we will continue our exploration of Python and computational thinking with additional application problems in data analysis in scientific applications, housing, stock market analyses, and more.

16
Advanced Applied Computational Thinking Problems

In this chapter, we will continue providing examples in multiple areas for applications of the Python programming language and computational thinking. We will be exploring multiple areas, such as geometric tessellations, creating models of housing data, creating electric fields, analyzing genetic data, analyzing stocks, creating a **convolutional neural network** (**CNN**), and more. We will use what we've learned so far in relation to **computational thinking** and the **Python** programming language to do the following:

- Create tessellations
- Analyze biological data
- Analyze data for specific populations
- Create models of housing data
- Create electric field lines
- Analyze generic data

- Analyze stocks
- Create a convolutional neural network (CNN)

After reading this chapter, you'll have learned how to perform various different analyses in working with data, creating tables and graphs that help to analyze existing data, as well as create training and testing models to help predict outcomes based on existing large datasets.

Technical requirements

You will need the latest version of Python installed to run the code in this chapter.

You will need the following libraries and packages installed for Python:

- **NLTK**
- **Cairos**
- **Pandas**
- **Matplotlib**
- **Seaborn**

You can find the full source code used in this chapter here: `https://github.com/PacktPublishing/Applied-Computational-Thinking-with-Python/tree/master/Chapter16`

Problem 1 – Using Python to create tessellations

In this section, we are going to provide an example using the `cairo` library for Python. We are going to create a **tessellation**, more specifically, a sample of a **Penrose tiling**. Because this is a straightforward problem, we are going to define our parameters using the computational thinking process, but not adhere to it precisely.

First, let's talk about the `cairo` library. Once the `pip install cairo` command is successful, you'll need to perform one more step to include all the components needed. Use `pip install pycairo` to add the necessary components. The `cairo` and `pycairo` packages are graphics libraries that work with Python. For more information, you can visit their web page: `https://cairographics.org/pycairo`.

Now let's define some things. A **tessellation** is a tiling that uses shapes that do not overlap to create patterns. Tessellations are often explored in geometry courses. For our example, we will create a Penrose tiling pattern using two triangles. We will also get to define our space and the number of sub-divisions we want the shapes to undergo. The more sub-divisions, the smaller the pattern in the space defined. Let's take a look at the algorithm (the file `ch16_tessellation.py` contains the full algorithm discussed here0:

1. The first thing we'll do is import the necessary packages and libraries:

ch16_tessellation.py

```
import math
import cmath
import cairo
```

2. Next, we want to define our canvas and the number of sub-divisions. Note that we chose `4` for our example. In *Figure 16.1*, you'll see the example from this snippet as well as two additional examples changing the sub-divisions:

```
#Define the configuration of the image.
canvas_size = (500, 500)
numberSubdivisions = 4
```

3. For the tessellation, we need to define the **Golden Ratio**.

   The golden ratio is also known as the golden mean or divine proportion (among other names). The ratio is approximately 1.618. As an example, if we were talking about a line segment divided into two parts, then the length of the larger segment divided by the length of the smaller segment would be equal to the sum of the segments divided by the larger segment: $\frac{A}{B} = \frac{A+B}{A}$. For the tessellation, we'll need to define that golden ratio.

   Take a look at the following code snippet:

```
#Define the Golden Ratio - gr
gr = (1 + math.sqrt(5)) / 2
```

4. Now we use functions to define what happens when our triangles sub-divide:

```
def subdivide(triangles):
    result = []
    for color, A, B, C in triangles:
        if color == 0:
            P = A + (B - A) / gr
            result += [(0, C, P, B), (1, P, C, A)]
        else:
            Q = B + (A - B) / gr
            R = B + (C - B) / gr
            result += [(1, R, C, A), (1, Q, R, B), (0, R,
Q, A)]
    return result
```

In the preceding code, we defined the function to sub-divide the triangles. The function contains a conditional statement to identify the color of the triangles prior to finding the ratio.

5. To create the wheel of the triangles, we need to append the triangles to a group. Because Python is an object-oriented programming language, we can easily do that by creating an empty list and then appending the shapes using loops:

```
#Wheel of teal triangles
triangles = []
for i in range(10):
    B = cmath.rect(1, (2*i - 1) * math.pi / 10)
    C = cmath.rect(1, (2*i + 1) * math.pi / 10)
    if i % 2 == 0:
        B, C = C, B  # Make sure to mirror every second
triangle
    triangles.append((0, 0j, B, C))

for i in range(numberSubdivisions):
    triangles = subdivide(triangles)
```

6. Now we have to prepare the canvas we'll use for our tessellation. Notice that we use the `cairo` functions to define the parameters using the variables we defined in the beginning of the algorithm. Here, we use `canvas_size`:

```
surface = cairo.ImageSurface(cairo.FORMAT_ARGB32, canvas_
size[0], canvas_size[1])
cr = cairo.Context(surface)
cr.translate(canvas_size[0] / 2.0, canvas_size[1] / 2.0)
```

```
wheelRadius = 1.2 * math.sqrt((canvas_size[0] / 2.0) ** 2
+ (canvas_size[1] / 2.0) ** 2)
cr.scale(wheelRadius, wheelRadius)
```

7. Now we have to define the two triangles we'll be using. In this case, our triangles are teal and purple, but you can change the RGB values in them, that is, if you'd like to test different colors:

```
#Define the teal triangles
for color, A, B, C in triangles:
    if color == 0:
        cr.move_to(A.real, A.imag)
        cr.line_to(B.real, B.imag)
        cr.line_to(C.real, C.imag)
        cr.close_path()
cr.set_source_rgb(.2, .8, .8)
cr.fill()

#Define the purple triangles
for color, A, B, C in triangles:
    if color == 1:
        cr.move_to(A.real, A.imag)
        cr.line_to(B.real, B.imag)
        cr.line_to(C.real, C.imag)
        cr.close_path()
cr.set_source_rgb(0.7, 0, 0.7)
cr.fill()

color, A, B, C = triangles[0]
cr.set_line_width(abs(B - A) / 10.0)
cr.set_line_join(cairo.LINE_JOIN_ROUND)
```

The preceding code creates the teal triangles and the purple triangles. Each is defined with the RGB values and created with paths and lines.

8. The rotated triangles make a tile pattern, that is, our tessellation. The tiles are also separated by a border. The color of the border is also defined in the loop that follows:

```
#Triangle borders
for color, A, B, C in triangles:
    cr.move_to(C.real, C.imag)
    cr.line_to(A.real, A.imag)
    cr.line_to(B.real, B.imag)
cr.set_source_rgb(0.3, 0.5, 0.3)
cr.stroke()
```

9. Finally, we want the algorithm to create an image file with our tessellation:

```
surface.write_to_png('tessellation.png')
```

The following screenshot shows three variations using different numbers of sub-divisions for our algorithm:

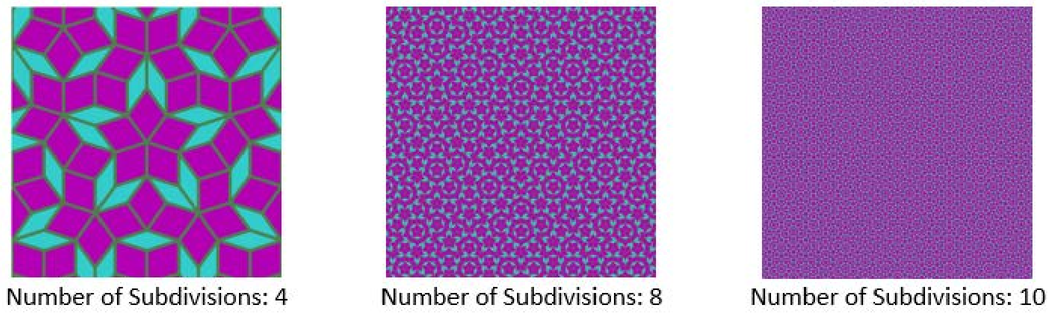

Number of Subdivisions: 4
Number of Subdivisions: 8
Number of Subdivisions: 10

Figure 16.1 – Sample tessellations

As you can see from the preceding images, the larger the number of sub-divisions, the smaller the tile pattern becomes to fit into our defined canvas size.

As you play with the algorithm, consider changing the canvas size, the sub-divisions, and the colors. If you want an additional challenge, try changing the triangle pattern to another polygon.

Problem 2 – Using Python in biological data analysis

For this particular problem, we'll be using the `Breast_cancer_data.csv` file, which can be found on **Kaggle** (`https://www.kaggle.com/nsaravana/breast-cancer?select=breast-cancer.csv`). The file has also been uploaded to the book's GitHub repository.

When looking at data, sometimes we want to make comparisons with the data we currently have, or we want to use it for predictions in machine learning. In this case, we're going to look at how we can present another type of plot, the **scatterplot**, using two specific values of columns in our dataset.

Let's imagine you received this data and already determined that your mean perimeter and mean textures are better predictors than the other values in the columns. Your goal now is to create an algorithm that will analyze the values for those two columns by comparing them using a scatterplot. Our goal is only to get that scatterplot. For additional analysis and machine learning applications, feel free to explore *Chapter 13*, *Using Classification and Clusters*, and *Chapter 14*, *Using Computational Thinking and Python in Statistical Analyses*, for additional help.

The full code for this problem can be found in the file `ch16_BreastCancerSample.py`. We can now start to design our algorithm:

1. We begin as we normally do with data, importing the libraries we'll use. Note that we are using two display libraries here, the `matplotlib` and `seaborn` libraries. This is the first time we'll use `seaborn`. We are using `seaborn` because the additional work, such as finding regression lines, is handled easily with the help of this library:

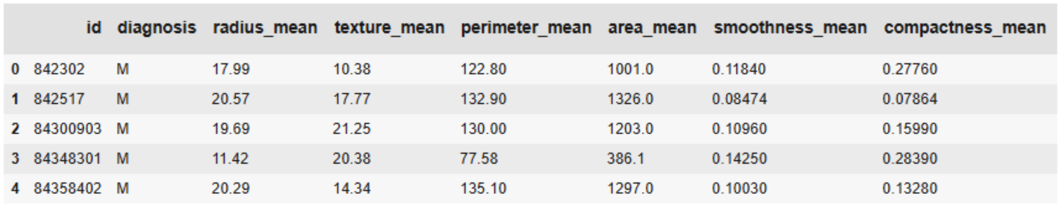
```
#Import libraries needed
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
%matplotlib inline
```

2. Now we're going to find the `.csv` file. Remember that you can always establish the directory first. Otherwise, make sure you include the full location of the file. Since our directories are different, make sure you change that before running the file:

```
#Get data set. Remember to check your directory and/or
add the full location of the file.
```

```
dataset = pd.read_csv('C:\\... \\breast-cancer.csv')
dataset.head()
dataset.isnull().sum()
```

Notice the `dataset.head()` command in the algorithm. If we run the code up until that point only, then we get the following output:

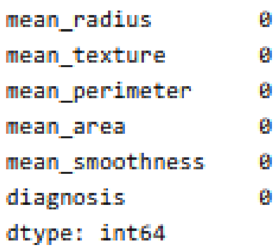


	id	diagnosis	radius_mean	texture_mean	perimeter_mean	area_mean	smoothness_mean	compactness_mean
0	842302	M	17.99	10.38	122.80	1001.0	0.11840	0.27760
1	842517	M	20.57	17.77	132.90	1326.0	0.08474	0.07864
2	84300903	M	19.69	21.25	130.00	1203.0	0.10960	0.15990
3	84348301	M	11.42	20.38	77.58	386.1	0.14250	0.28390
4	84358402	M	20.29	14.34	135.10	1297.0	0.10030	0.13280

Figure 16.2 – Table showing the heading values

The `dataset.isnull().sum()` command helps us see whether we have empty data points or values.

3. If we have many null points, we can clean the dataset before we start the analysis. This data is clean, as can be seen from the following output if we run the program up until `dataset.isnull().sum()`, as can be seen in the following screenshot:



```
mean_radius        0
mean_texture       0
mean_perimeter     0
mean_area          0
mean_smoothness    0
diagnosis          0
dtype: int64
```

Figure 16.3 – Output of the null check

4. Since there are no missing values, as seen in the preceding screenshot, we're going to continue to our next snippet, where we'll create the `count` variable for diagnosis:

```
#Create count variable for diagnosis
count = dataset.diagnosis.value_counts()
count
```

The `count` variable is created in the preceding snippet, meaning we can create a bar graph using the values of the diagnosis, whether it was malignant or benign.

5. The following code snippet creates that bar graph and shows us the resulting output:

```
#Create bargraph of the diagnosis values
count.plot(kind = 'bar')
plt.title('Tumor distribution (malignant: 1, benign: 0)')
plt.xlabel('Diagnosis')
plt.ylabel('count')
plt.show()
```

Take a look at the following screenshot, which shows the bar graph using the diagnosis values. As you can see, the bar graph shows the count of **malignant** tumors versus **benign** tumors:

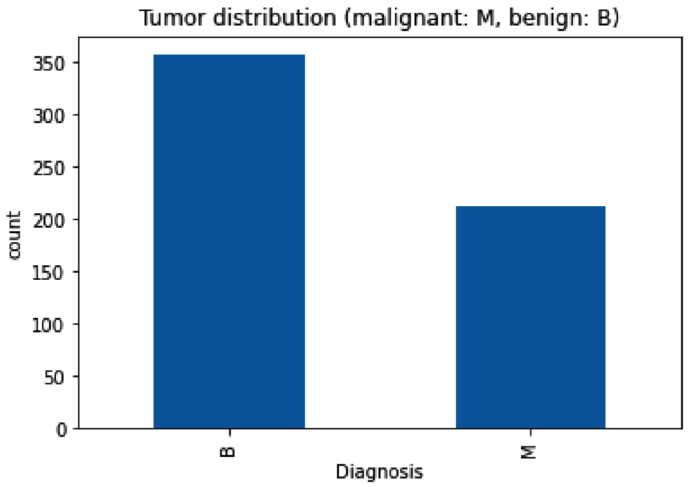


Figure 16.4 – Malignant versus benign diagnosis bar graph

Now that we have that information and the bar graph, we can start looking at other combinations and comparisons using the values from our dataset.

6. You can run a different analysis to see which are more relevant, but for now, we are just going to use the perimeter mean and texture mean to create our scatterplot. The following code snippet shows how to create those using the `seaborn` library:

```
y_target = dataset['diagnosis']
dataset.columns.values
dataset['target'] = dataset['diagnosis'].
map({0:'B',1:'M'})

#Create scatterplot of mean perimeter and mean texture
```

```
sns.scatterplot(x = 'perimeter_mean', y = 'texture_mean',
data = dataset, hue = 'diagnosis', palette = 'bright');
```

Once we have created our scatterplot, the algorithm will return the following output, which shows the mean perimeter scatterplot compared with the mean texture scatterplot:

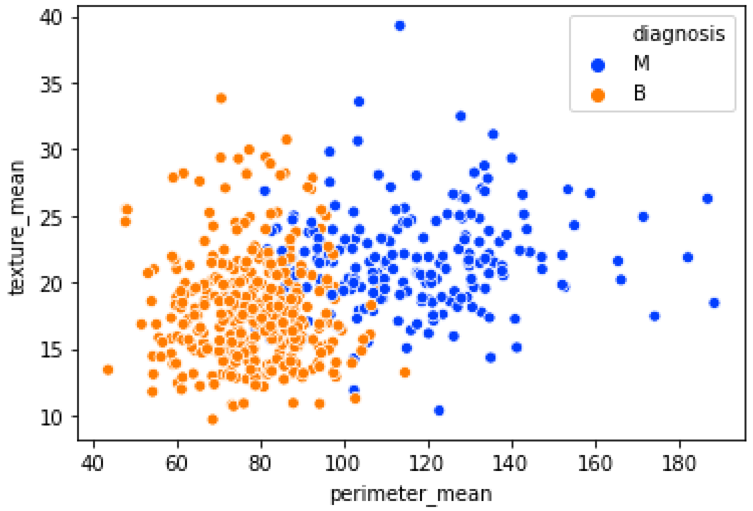


Figure 16.5 – Mean perimeter versus mean texture scatterplot

We are going to pause here for this data analysis. However, please note that you can take this example much further. In fact, you can find multiple applications and analyses done in Kaggle with this particular dataset and how some developers and coders incorporated machine learning in order to make predictions. The world of bioinformatics is wide and data science applications are continuing to grow. The use of Python in these problems is helpful due to its ease of use and applicable libraries.

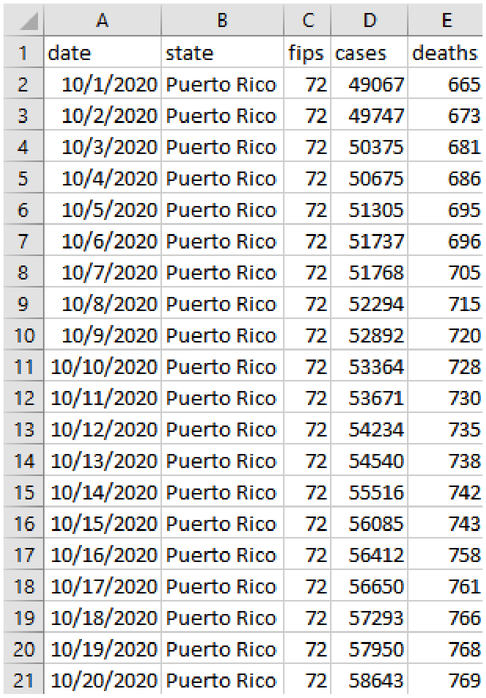
Problem 3 – Using Python to analyze data for specific populations

For this section, we'll state our problem this way—the year is 2020 and the world is overwhelmed by a pandemic due to the **SARS-COV-19** virus, also known as **coronavirus** or **COVID-19**. The data is available widely and we are trying to look at what's happening in a specific location, in particular, how the number of deaths are growing for that location. We find the New York Times GitHub repository, which contains the COVID-19 data, and download the master data, which is updated daily. Let's look at what we need to do and how we find it.

Defining the specific problem to analyze and identify the population

This problem is broad. *Too broad!* So let's first look at 1 location and for only 1 month. For example, let's choose Puerto Rico and the month of October. From the master `.csv` file, we've pulled only the data specific for Puerto Rico and added it to our repository. Again, the master can be found in the New York Times, covid-19-data repository, and you can perform multiple analyses using the full data, a specific state, or even a specific county.

For now, we're going to concentrate on creating a visual for the data for deaths specific to Puerto Rico in October 2020. From just looking at the data, we see that the deaths are rising. Take a look at the following screenshot:



	A	B	C	D	E
1	date	state	fips	cases	deaths
2	10/1/2020	Puerto Rico	72	49067	665
3	10/2/2020	Puerto Rico	72	49747	673
4	10/3/2020	Puerto Rico	72	50375	681
5	10/4/2020	Puerto Rico	72	50675	686
6	10/5/2020	Puerto Rico	72	51305	695
7	10/6/2020	Puerto Rico	72	51737	696
8	10/7/2020	Puerto Rico	72	51768	705
9	10/8/2020	Puerto Rico	72	52294	715
10	10/9/2020	Puerto Rico	72	52892	720
11	10/10/2020	Puerto Rico	72	53364	728
12	10/11/2020	Puerto Rico	72	53671	730
13	10/12/2020	Puerto Rico	72	54234	735
14	10/13/2020	Puerto Rico	72	54540	738
15	10/14/2020	Puerto Rico	72	55516	742
16	10/15/2020	Puerto Rico	72	56085	743
17	10/16/2020	Puerto Rico	72	56412	758
18	10/17/2020	Puerto Rico	72	56650	761
19	10/18/2020	Puerto Rico	72	57293	766
20	10/19/2020	Puerto Rico	72	57950	768
21	10/20/2020	Puerto Rico	72	58643	769

Figure 16.6 – Data for the first 20 days of October 2020 in Puerto Rico

As you can see from the preceding screenshot, the **deaths** column continues to rise, as does the number of cases, which we will take a look at a little later in this problem.

While data read in a table format can be helpful, a visual representation is critical if we were to present this information, especially if we want to identify trends and influence policy changes. So let's take a look at how we'd create a scatterplot for this particular data. The full file can be found in `ch16_CovidPR.py`:

1. As always when working with data, we need to make sure we import the libraries we'll be using:

```
import pandas as pd
import matplotlib.pyplot as plt
```

2. Next, we'll need to get our file. Remember, there are multiple ways to do this. You can give Python the full location of your file or you can first identify the directory and only provide the filename. Please make sure you change the location of the `.csv` file you'll be using before running the program:

```
df = pd.read_csv('C:\\...\\us-PuertoRico.csv')
```

3. After identifying the file, we'll just create a simple scatterplot using the dates as our *x*-axis, and the deaths as our *y*-axis. The next few commands in this code snippet are done to make it easier to read the chart, such as the *y*-axis label, the rotation of the *x-tick* marks, and the title of the chart. The *x-tick* marks are the division marks for the horizontal axis, or *x*-axis. You can see the *x-tick* marks and the labels in *Figure 16.6*:

```
plt.scatter(df['date'], df['deaths'])
plt.xticks(rotation = 90)
plt.title('Deaths per day during October 2020 due to
COVID19 in Puerto Rico')
plt.ylabel('Number of Deaths')
fig.tight_layout()
plt.show()
plt.savefig('COVID_PR.png')
```

As you can see from the preceding code snippet, we also created an image file for us to use later, if needed. The chart will show on our screen, as can be seen in the following screenshot:

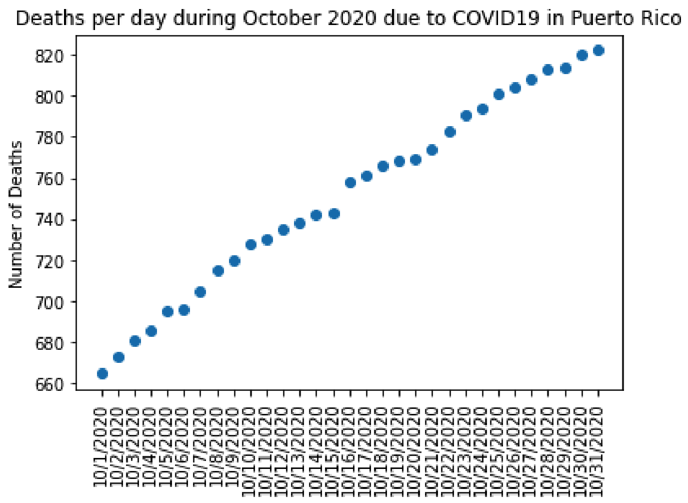


Figure 16.7 – Deaths per day during October 2020 due to COVID-19 in Puerto Rico

This is a helpful chart to see that the number of deaths is increasing at a steady pace. There are more things we can do, such as try to determine the regression, which is another functionality we can do with Python using the `numpy` library, which you are welcome to work out!

For now, we're going to take a look at the cases by date. The code is the same as previously, except that our *y*-axis and title will be different. The full code can be found in the `ch16_CovidPR_2` file. As the code is very similar, we are not sharing it here. However, our resulting graph can be seen in the following screenshot:

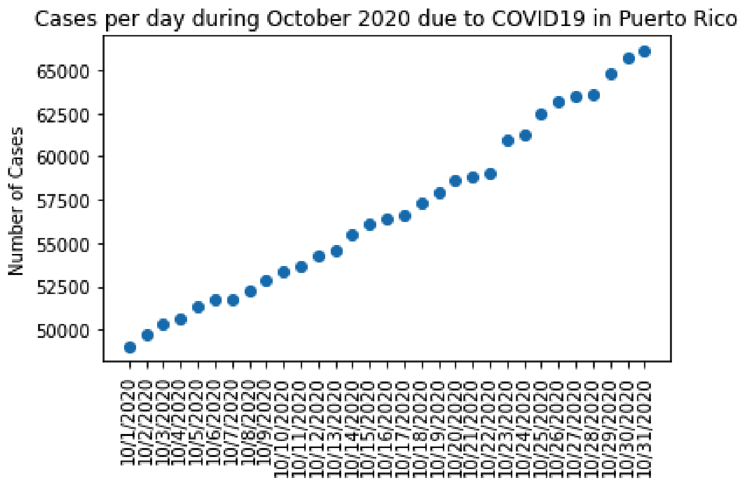


Figure 16.8 – Cases per day during October 2020 due to COVID-19 in Puerto Rico

As you can see from the preceding screenshot, the number of cases has continued to rise each day in Puerto Rico. There are multiple things we could do with these two graphs; analyzing their regressions, verifying additional trends by looking at other monthly data, and so on. You've now seen how to create a simple plot to display the data based on your `.csv` file; the rest is up to you. We will look at a new problem in the next section.

Problem 4 – Using Python to create models of housing data

Let's take a look at a problem where we want to display trends and information about the housing market in Brooklyn, New York. The dataset includes information from the NYC Housing Sales Data for 2003-2017. The dataset used has the information merged in a usable format and can be found on Kaggle here (`https://www.kaggle.com/tianhwu/brooklynhomes2003to2017`). In addition, a copy of the `.csv` file can be found in the GitHub repository under the name `brooklyn_sales_map.csv`.

Defining the problem

We have a large data file for this particular problem. We can look at information by neighborhood, sale prices by year, compare the year built to the neighborhood to find trends, history, and so on. We could spend hours, days, and weeks just on this one dataset. So let's try to focus our energy into what we are going to accomplish with this example. For this, we're going to create two visual models. The first is a horizontal bar graph of housing percentages in a sale range according to the year of sale. The second is a bar graph that shows the price range by the neighborhoods where houses were sold.

The horizontal bar graph can help display the data in a much clearer way so that we can see house price ranges and whether there are significant changes. The vertical bar graph can show us those same price ranges by neighborhood, so we can see whether there are significant changes depending on the neighborhoods where the houses were sold.

Algorithm and visual representations of data

Let's take a look at the code snippet. The full file can be found in the GitHub repository under `ch16_housingBrooklyn.py`. As usual, don't forget to update the file location in the file before trying to run the program:

1. For this particular program, we'll need the `pandas` and `matplotlib` libraries, so we need to import them:

   ```
   import pandas as pd
   import matplotlib.pyplot as plt
   ```

2. Next, we need to read our file. This is where you'll need to update this code in order to run it from your machine:

   ```
   df = pd.read_csv('C:\\...\\brooklyn_sales_map.csv')
   ```

3. Now we're going to create our `bins`. These are our range of values and we'll call them when we are creating the charts, as you can see under `df['price_range']` within the following few lines of code:

   ```
   bins = [-100000000,20000,40000,60000,80000,100000,1000000
   ,10000000,500000000]
   ranges_prices = ['$0-$200k','$200k-$400k','$400k-
   $600k','$600k-$800k','$800k-$1mlln','$1mlln-
   $10mlln','$10mlln-$100mlln','$100mlln-$500mlln']
   df['price_range'] = pd.cut(df['sale_price'], bins = bins,
   labels = ranges_prices)
   ```

4. Now we're going to define a function, where we are going to convert some of the data. Notice that we run that function on each of the years from the dataset to find our percent total, which we'll use later on for `housing_df`:

   ```
   def convert(year):
       return df[df['year_of_sale'] == year].groupby('price_
   range').size()
   percent_total = [x/sum(x)*100 for x in [convert(2003),
   convert(2004),convert(2005),convert(2006),
   convert(2007),convert(2008),convert(2009),convert(2010),
   convert(2011),convert(2012),convert(2013),convert(2014),
   convert(2015),convert(2016),convert(2017)]]
   year_names = list(range(2003,2018))

   ```

```
housing_df = pd.DataFrame(percent_total, index = year_
names)
ax_two = housing_df.plot(kind = 'barh', stacked = True,
width = 0.9, cmap = 'Spectral')
plt.legend(bbox_to_anchor = (1.45, 1), loc='upper right')

ax_two.set_xlabel('Percentages', fontname='Arial',
fontsize = 12)
ax_two.set_ylabel('Years', fontname='Arial', fontsize =
12)
ax_two.set_title('Housing Sale ')
```

The preceding snippet helps us to create the first of the two models. This one is a horizontal bar graph. We labeled all the axes and the graph, and then, in the next line shown in the preceding code snippet, we also defined the color map we'll use for this graph, in this case, `'Spectral'`. You can play with the color mappings available for easier reading. Take a look at our first graph, shown as follows:

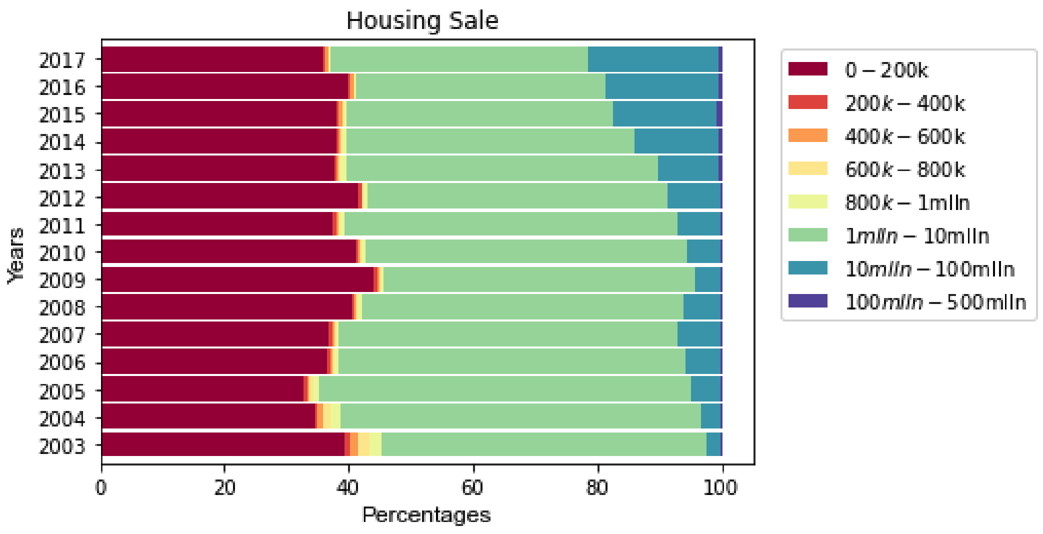


Figure 16.9 – Housing sales in Brooklyn by year

Notice that we use percentages in the preceding screenshot. This allows us to show how much of the sales were in each price range, but it does not show us the actual number of sales in each price range. Those two things are quite different. Here, we are looking for trends. The percentage of sales that were higher than $1,000,000 has consistently increased after a slight dip from **2008** to **2009**. In **2017**, a much larger percentage of sales was above that price point than in **2003**.

But that's total sales. If we were only looking at this graph without looking at the numbers as well, we wouldn't know whether fewer houses were sold in total in 2017, for example. Again, the important thing to note here is that this graph is extremely helpful for understanding the share of housing selling under each price range, but that's all this graph really gives us. Now let's look at the remaining code from our file.

5. In this next snippet, we create our second graph, which uses our information to produce a vertical bar graph with the percentages within each price range for each neighborhood:

```
df.groupby(['neighborhood','price_range']).size().
unstack().plot.bar(stacked = True, cmap = 'rainbow')
```
```
plt.legend(bbox_to_anchor = (1.45, 1), loc = 'upper
right')
```
```
plt.title('Pricing by Neighborhoods in Brooklyn from 2003
to 2017')
```
```
plt.ylabel('Price Range')
```
```
plt.xticks(fontsize = 6)
```

This graph shows us the price range in each neighborhood using a bar graph. Let's take a look at our second graph in the following screenshot:

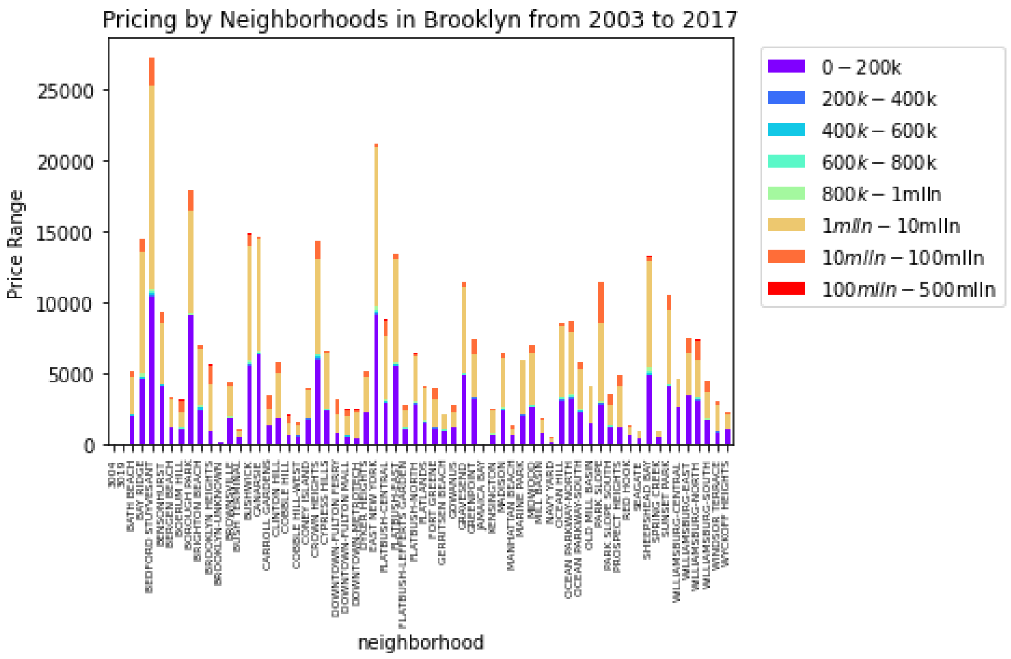


Figure 16.10 – Pricing by neighborhoods in Brooklyn from 2003-2017

As you can see from the preceding screenshot, we get some important information that is additional or more detailed than that in *Figure 16.1*. In this case, the data is provided by neighborhood and the breakdown is provided by price range in those neighborhoods.

When we look at a large dataset, we can create multiple different models and even use them to predict values moving forward. Take a look at the data available in the `.csv` file and try to create some different representations using other data, such as commercial versus residential sales, tax class breakdowns, and more.

Problem 5 – Using Python to create electric field lines

In this section, let's look at some applications of Python for engineering and, in particular, physics. We are going to be creating an **electric field lines plot**. *Why would we want to create that kind of plot and what exactly is it?* An electric field happens when there is an electric charge. We use vectors to show these electric fields for every point in space. In physics, an electric field is an electric force per unit charge. Take a look at what that field looks like for a positive point charge and a negative point charge, as shown in the following diagram:

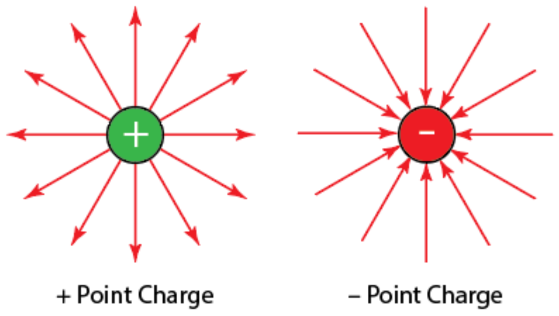


Figure 16.11 – Electric field sample

As you can see, an electric field line will begin at the charge or end at the charge. If it begins at the charge, it is positive, while it if ends at the charge, it is negative, as can be seen in the preceding screenshot. There are a fewer number of lines for a lesser charge and more lines for a larger charge. In addition, the lines will be closer for a larger charge than for a smaller charge.

For our problem, we want to create an electric field lines plot for any number of charges. So let's take a look at what that would look like in the following code file. Note that we've broken the code down to explain various sections, but the full file can be found in the GitHub repository:

1. As usual, first, we'll begin by importing the necessary libraries:

ch16_electricFieldLines.py

```
import numpy as np
import matplotlib.pyplot as plt
import random
```

2. Then, we'll set up our *x* and *y* axes by defining some variables:

```
np.seterr(divide='ignore', invalid='ignore')

#Define the size of the electric field lines grid
N = 20
M = 25

#Set the x and y coordinates
x_coor = np.arange(0, M, 1)
y_coor = np.arange(0, N, 1)
x_coor, y_coor = np.meshgrid(x_coor, y_coor)

E_x = np.zeros((N, M))
E_y = np.zeros((N, M))
```

   In the preceding code snippet, we work on defining our grid and setting up the coordinates. Then we create the mesh grid. The `meshgrid()` function returns matrices of coordinates from vector coordinates.

3. After we have set our coordinates and set our mesh grid, we can start defining what happens with our charges. First, we'll need to identify how many charges will be plotted:

```
#Set the number of total charges to plot
nq = 8
#Create empty lists to store coordinates of charges
qq = [[], []]
```

```
for dummy in range(nq):
    q = random.choice([-1, 1])
    q_x, q_y = random.randrange(1, N), random.
randrange(1, M)
    qq[0].append(q_y)
    qq[1].append(q_x)
    for i in range(N):
        for j in range(M):
            denom = ((i - q_x) ** 2 + (j - q_y) ** 2) **
1.5
            if denom != 0:
                E_x[i, j] += q * (j - q_y) / denom
                E_y[i, j] += q * (i - q_x) / denom
```

As you can see from the preceding snippet of code, after we identified the number of charges, we created two empty lists. Then we added coordinates to those lists based on the number of charges using nested `for` loops.

4. After we do the necessary math to get our coordinates and vectors, we can then go ahead and plot our electric field lines graph. We'll use a `quiver` plot, which is a `matplotlib` graphic we can use for creating vector fields:

```
C = np.hypot(E_x, E_y)
E = (E_x ** 2 + E_y ** 2) ** .5
E_x = E_x / E
E_y = E_y / E

plt.figure(figsize=(12, 8))

#Plot charges
plt.plot(*qq, 'ms')
#Create 2D array
rr = plt.quiver(x_coor, y_coor, E_x, E_y, C,
pivot='middle')

cbar = plt.colorbar()
cbar.ax.set_ylabel('Magnitude')

#Label graph
```

```
plt.title('Electric Field Lines in Python')
plt.axis('equal')
plt.axis('off')
plt.show()
```

It's important to always add labels to our graphs and plots, as this will make the information more readable, especially for those who do not know what the code behind this means or what the graphs and plots represent. When we run our code snippet, we get the following output:

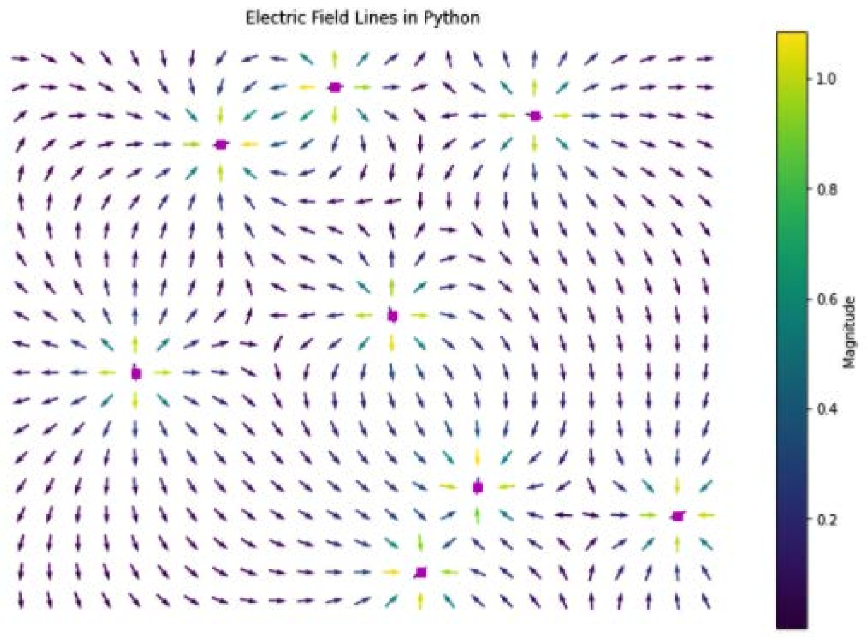


Figure 16.12 – Electric field lines for eight charges

As you can see in our plot, there are positive and negative charges. Take a look at the bottom right-hand corner charge in the graph. That charge is negative, as the arrows are pointing toward the charge. The *left-most* charge, which has been zoomed-in, shown as follows, shows a positive charge, as the arrows are pointing away from the charge:

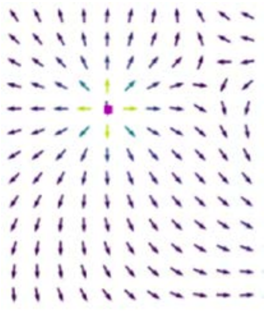


Figure 16.13 – Zoomed positive charge

5. Let's take a look at one final graphic, shown in the following screenshot, with an electric field lines plot that has three charges:

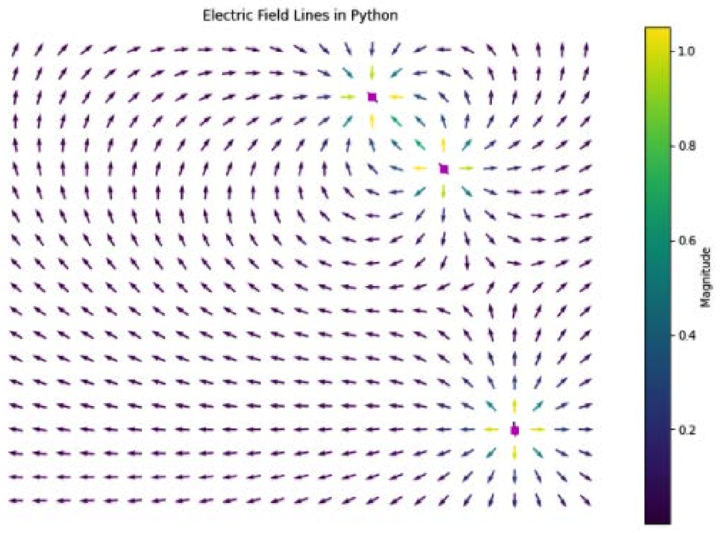


Figure 16.14 – Electric field lines with three charges

As you can see, this particular plot contains two positive charges and one negative charge. If we ran this again, we may get three positive charges, for example, as each time we run the algorithm, we get a new representation with random values for positive and negative charges.

These kinds of fields and learning how to use vectors and quiver plots can help us represent a lot of information. In the case of electric field lines, we can learn a lot about the charges, direction, and the strength of those charges with a simple visual plot.

Take a look at the code snippet in the GitHub repository and try changing some of the parameters, such as the size of the plot and the number of charges. The more practice you get with these plots and altering some of the parameters, the easier they become to create.

Problem 6 – Using Python to analyze genetic data

Let's shift focus to looking at a larger dataset. You're working with laboratory mice and get data for **trisomy mice** and protein expressions in these mice. We've truncated some of the data from the public domain file in Kaggle for this due to its huge size. We're only focusing on six protein expressions for the mice and again, only the trisomy (**Down syndrome**) mice in the study. The full file can be found in Kaggle at `https://www.kaggle.com/ruslankl/mice-protein-expression`. The truncated file can be found in our GitHub repository.

Let's say you don't know where to start with this data. *What should you even be looking at?* Well, that's often the first thing we encounter in data science. We don't always get to be part of the study design or data collection. Many times, we receive large data files and need to figure out what to look for, how to tackle the problem, whatever we decide the problem is, and how to display the information in the best possible way.

Also, this is your reminder to change the location of the file before you attempt to run this program. This very simple program can be found in the `ch16_pairplots.py` file. Let's begin with the algorithm:

1. The `seaborn` library can help us a bit just to get us started. We can create `pairplot()`, which will correlate the numerical data in the `.csv` file using histograms and scatterplots. It's kind of like a fantastic magic trick. We can use two lines of code to see what we see. Take a look at the two lines that are needed to generate *Figure 16.7* (note that there's actually four lines of code, but I'm not counting the two lines I'm using to import my libraries):

```
import seaborn as sns
import pandas as pd
df = pd.read_csv('C:\\...\\Data_Cortex_Nuclear.csv')
sns.pairplot(df, hue = 'Treatment')
```

Be patient when you run this program. The algorithm may be simple, but what it's doing in the background isn't so simple. Take a look at the following screenshot to see our pairplot:

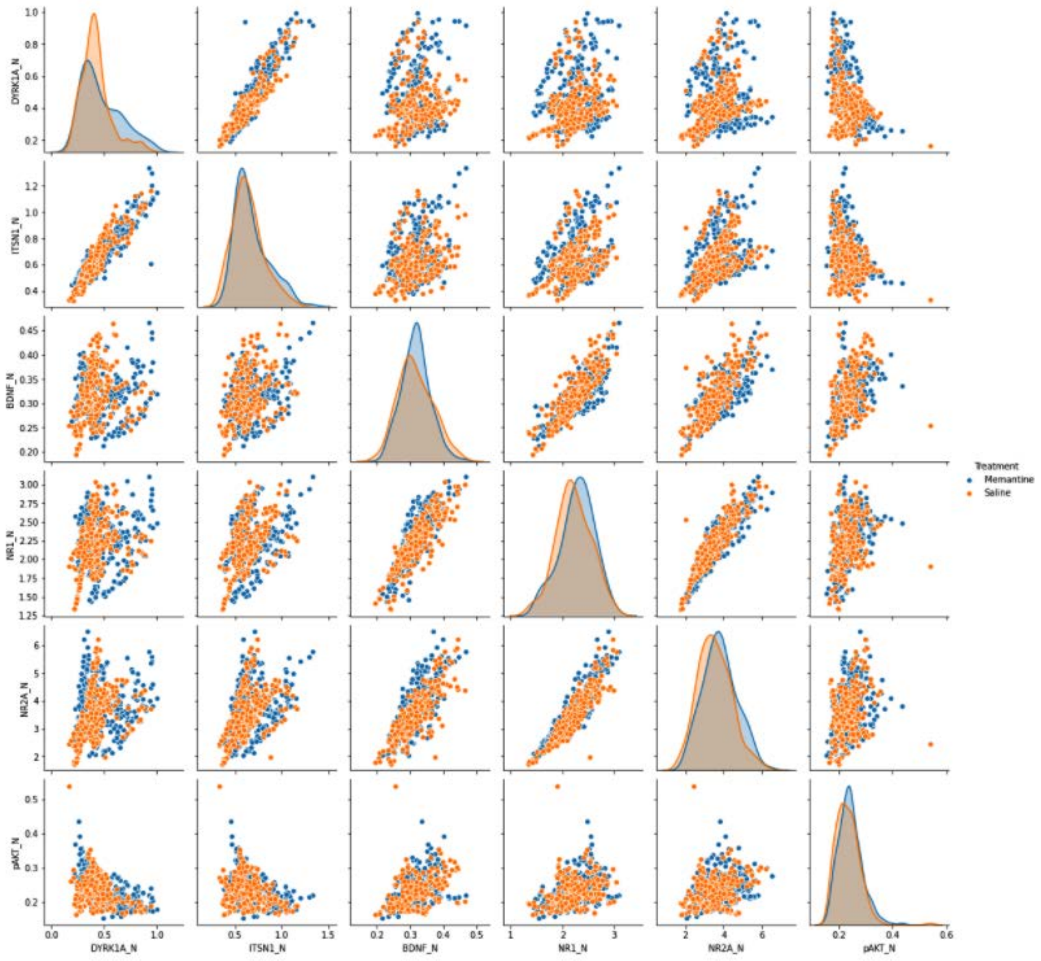


Figure 16.15 – Pairplot of protein expressions in trisomy mice with the treatment variable

Notice that our data has two colors based on the treatment, which is an injection of memantine or saline, respectively. From the plots, we can see that some of the proteins seem to have a higher correlation than others. Let's pause on that for a second.

2. Let's now say that our goal wasn't to check on the expression based on the treatment, but rather the class. Then we can run the code, but first we change the hue to `class` in our algorithm. The result is shown as follows:

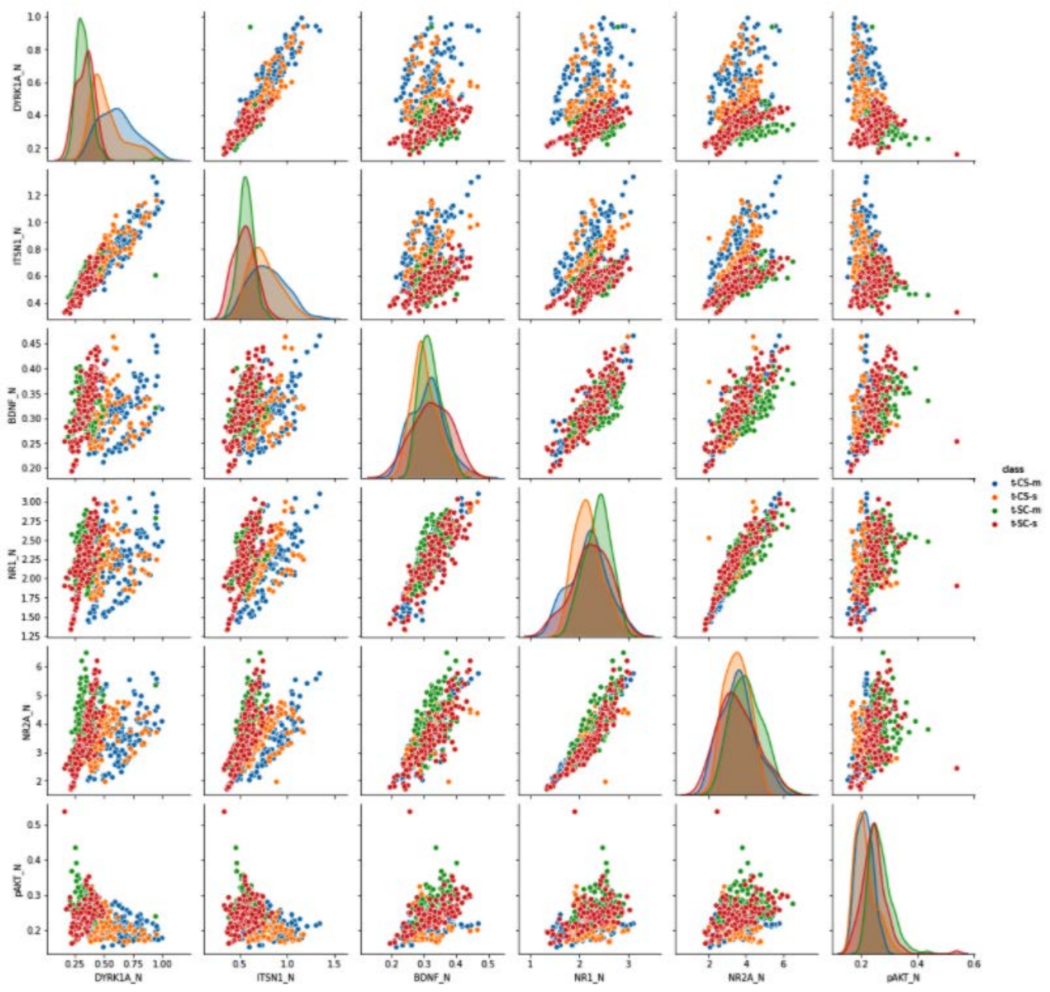


Figure 16.16 - Pairplot of protein expressions in trisomy mice with the class variable

Notice that the plots are extremely similar. Where the plots do differ, however, is in the identification of where each of the points lies based on another characteristic. For example, in the `class` variable chart, we have four colors because there are four classes of mice in our particular dataset.

They are, namely, **t-CS-s**, which refers to the mice stimulated to learn (shock) and injected with saline; **t-CS-m**, which refers to the mice stimulated to learn (shock) and injected with memantine; **t-SC-s**, which refers to mice not stimulated to learn and injected with saline; and **t-SC-m**, which refers to mice not stimulated to learn and injected with memantine.

Looking at our correlations, we can see that there are strong positive correlations between many of the proteins, such as **NR2A_N** and **BDNF_N**. Whether or not that's relevant, whether it matters in our study, and whether it's not significant is something that we'd have to take into consideration if this were our study. Once we see the plots, we can then choose to explore the information further.

Another type of plot that can be helpful when looking at this dataset is the boxplot. We can use the boxplot to see the protein expression level by class for a protein we want to look at more closely. Let's take the `NR2A_N` protein. Using the `seaborn` boxplot, we can create a plot for this particular protein using the code in the `ch16_boxplot.py` file. As always, check the file location first:

```
import pandas as pd
import scipy.stats as stats
import matplotlib.pyplot as plt
import seaborn as sns

df = pd.read_csv('C:\\...\\Data_Cortex_Nuclear.csv')

protein = df[['NR2A_N', 'class']].dropna()
sns.boxplot(x='class', y='NR2A_N', data = protein)
plt.show
```

In the preceding code, we identify the things we want to compare, which, in this case, are the protein and the class. We'll then create the boxplot using our `seaborn` library, which can be seen as follows:

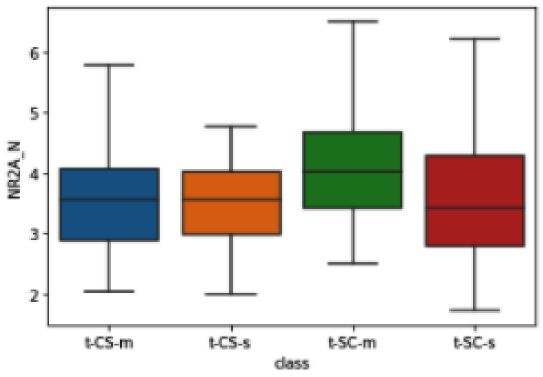


Figure 16.17 – Boxplot of the NR2A_N protein expression by class

As you can see from the graph, the distributions for our trisomy mice vary by class, with the mice that were not stimulated to learn and injected with saline showing a wider range in the expression of this protein. Let's try changing that protein to one of our others in the dataset, that is, the **ITSN1_N** protein. The following screenshot shows the resulting boxplot:

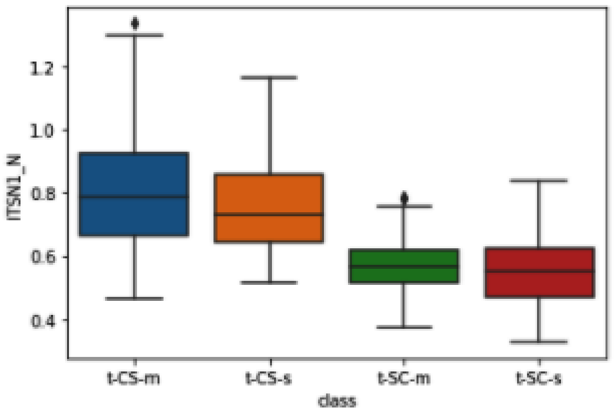


Figure 16.18 – Boxplot of the ITSN1_N protein expression by class

In this particular boxplot, we can identify outliers in the **t-CS-m** and **t-SC-m** classes, that is, both mice classes that were injected with memantine. That may tell us to seek more information about any relationship between the memantine injections and that particular protein. *If we were to look at non-trisomy mice, would these spreads of data (range) hold for that protein if the other elements were the same?* Those are some of the things we would ask ourselves in looking at datasets such as this one.

If you remember, the computational thinking process is rarely a straight line. If we identify things we want to consider in our algorithm, we don't just leave our algorithm alone and decide it was already done, so we won't change it. We go back to identify what we need again, make the necessary changes to our design, and create our algorithm again. That's more closely what happens when we are dealing with larger datasets. We look at some initial visualizations, maybe create a few different types of plots, run some statistical analysis, and then decide where to go next with the data. This is just a glimpse into what is possible with Python.

Problem 7 – Using Python to analyze stocks

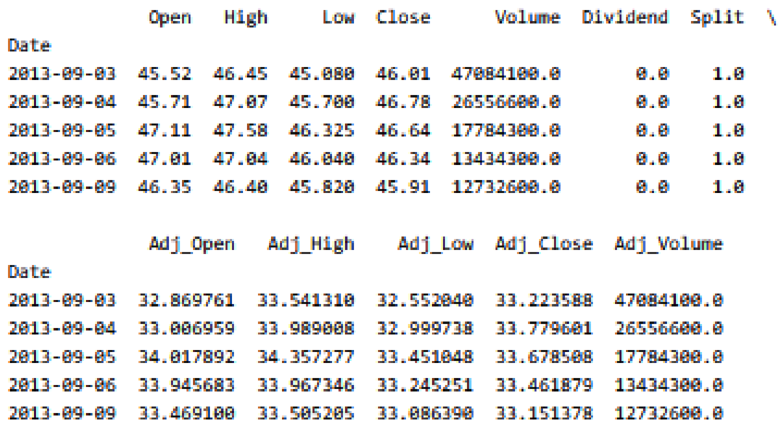
Time to play with some stocks. You can access a lot of data through **Quandl**, which allows for the use of a free API for educational uses. There are also premium datasets available. We're sticking to educational purposes, so that should be enough for our requirements.

In this problem, we're going to learn how to pull data from Quandl and look at the VZ stock prices. **VZ** is the code for **Verizon** stock prices. We're going to use them to predict the prices using `quandl`, which is a package for Python in addition to being a website full of useful information. Let's take a look at how we grab the information we want. The full code, minus the API key, can be found in our repository under the `ch16_stockAnalysis.py` file:

1. Let's take a look at how we can import the data. You'll need your own API for this. If you want to check another stock, say for `AMZN`, you'd substitute `'EOD/VZ'` with `'WIKI/AMZ'`, for example. **AMZN** is the code for **Amazon** stock. But let's take a look at our `VZ` set:

```
import quandl
import numpy as np
from sklearn.linear_model import LinearRegression
from sklearn.svm import SVR
from sklearn.model_selection import train_test_split

#Get data from Quandl. Note that you'll need your own API to substitute in the api.key below.
quandl.ApiConfig.api_key = '…'
VZ = quandl.get('EOD/VZ')
print(VZ.head())
```

   When we run the preceding code, we get the table for the first five values in our dataset. The following screenshot shows our table of values:



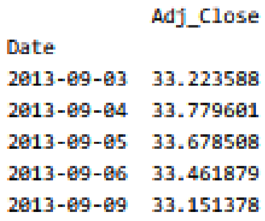
```
             Open   High     Low  Close      Volume  Dividend  Split  \
Date
2013-09-03  45.52  46.45  45.080  46.01  47084100.0       0.0    1.0
2013-09-04  45.71  47.07  45.700  46.78  26556600.0       0.0    1.0
2013-09-05  47.11  47.58  46.325  46.64  17784300.0       0.0    1.0
2013-09-06  47.01  47.04  46.040  46.34  13434300.0       0.0    1.0
2013-09-09  46.35  46.40  45.820  45.91  12732600.0       0.0    1.0

             Adj_Open   Adj_High    Adj_Low  Adj_Close  Adj_Volume
Date
2013-09-03  32.869761  33.541310  32.552040  33.223588  47084100.0
2013-09-04  33.006959  33.989008  32.999738  33.779601  26556600.0
2013-09-05  34.017892  34.357277  33.451048  33.678508  17784300.0
2013-09-06  33.945683  33.967346  33.245251  33.461879  13434300.0
2013-09-09  33.469100  33.505205  33.086390  33.151378  12732600.0
```

Figure 16.19 – EOD/VZ stock table

2. Now say we only wanted to focus on the adjusted close value in order to later make predictions. Then we can use the following code snippet to do so:

```
#Grab the Adj_Close column
VZ = VZ[['Adj_Close']]
print(VZ.head())
```

After running the preceding code, our adjusted table can be seen as follows:



```
            Adj_Close
Date
2013-09-03  33.223588
2013-09-04  33.779601
2013-09-05  33.678508
2013-09-06  33.461879
2013-09-09  33.151378
```

Figure 16.20 – EOD/VZ adjusted close stock value table

Now that we've learned to pull current data, we're going to now work with a dataset that we've placed in the GitHub repository so that we can ensure that you are able to replicate the results. You can then try to do this for current data using the Quandl API.

Let's take a look at the dataset with the filename `VZ.csv`. This contains the same data for VZ from 1983 to April 2020. *What do we want from this dataset?* We want to make some predictions. So let's build that model.

Note that the code is fairly large, so the file with everything you need (minus the file location on *line 15*, which you'll need to add), is in `ch16_predictionsModel.py`. But let's take a look at some of the snippets of code from that file:

1. The following code snippet will create a plot for our existing data in the dataset. It selects the `Date` column from the file and sets that as the index value. Then it creates a figure and adds the labels for the graph and axes:

```
VZ["Date"]=pd.to_datetime(VZ.Date,format="%Y-%m-%d")
VZ.index=VZ['Date']
plt.figure(figsize=(16,8))
plt.plot(VZ["Close"],label='Close price history')
plt.title('Closing price over time', fontsize = 20)
plt.xlabel('Time', fontsize = 15)
plt.ylabel('Closing price', fontsize = 15)
plt.show()
```

We're not looking at a model yet. We haven't defined our training data. We're just looking at what happened to our stock prices from **1983** to **2020**, shown as follows. Note that the first tick mark label states **1984**. Our data can be seen to start just slightly before **1984**. The tick marks are every 4 years since 1980, as shown:

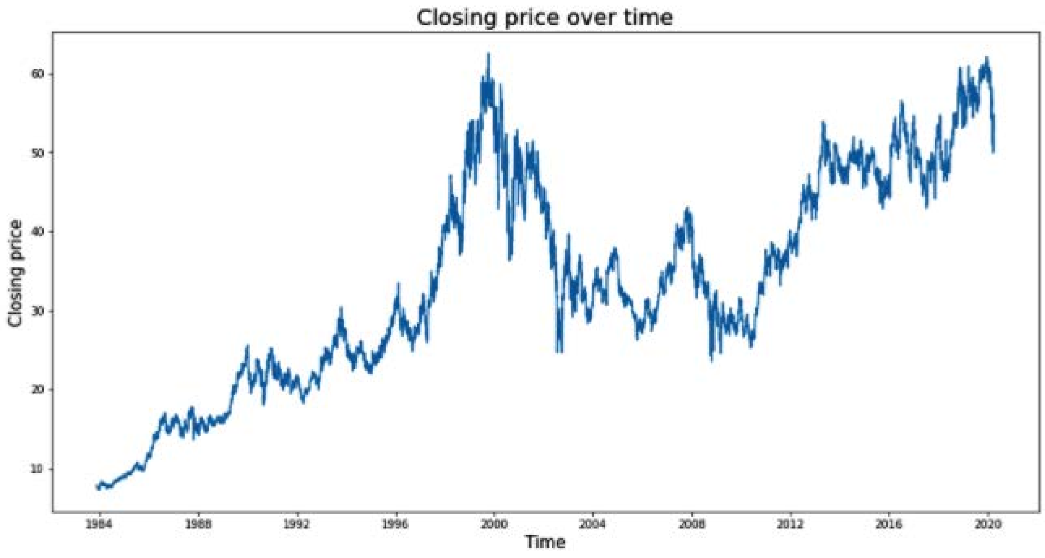


Figure 16.21 – Closing price over time for VZ stock

Notice from the preceding chart that stock prices are not linear in any way. They rise, fall, and rise again. A predictive model will require a lot of data in order for us to prepare the best predictions possible. Our dataset has 9,166 rows of data. That's going to come in handy in a second.

2. Let's take a look at another snippet of code that we'll be using:

```
VZ3=VZ2.values
train_data=VZ3[0:750,:]
valid_data=VZ3[750:,:]
VZ2.index=VZ2.Date
VZ2.drop("Date",axis=1,inplace=True)
scaler=MinMaxScaler(feature_range=(0,1))
scaled_data=scaler.fit_transform(VZ2)
x_train_data,y_train_data=[],[]
for i in range(60,len(train_data)):
    x_train_data.append(scaled_data[i-60:i,0])
    y_train_data.append(scaled_data[i,0])

x_train_data,y_train_data=np.array(x_train_data),np.
array(y_train_data)
x_train_data=np.reshape(x_train_data,(x_train_data.
shape[0],x_train_data.shape[1],1))
```

Notice the `750` value in the `train_data=VZ3[0:750,:]` line of code. That means I'm using just the first 750 rows of 9,166 possible rows of data to train my model. This isn't so great.

Let's take a look at the following screenshot, which shows the results when we run this prediction model. Note that we did choose to copy the original information into our graphic. Python will point that out as a possible thing we want to fix. That's up to you to do so. For now, having the original data available as overlaid for our graph provides us with a great visual for comparison purposes as regards our model:

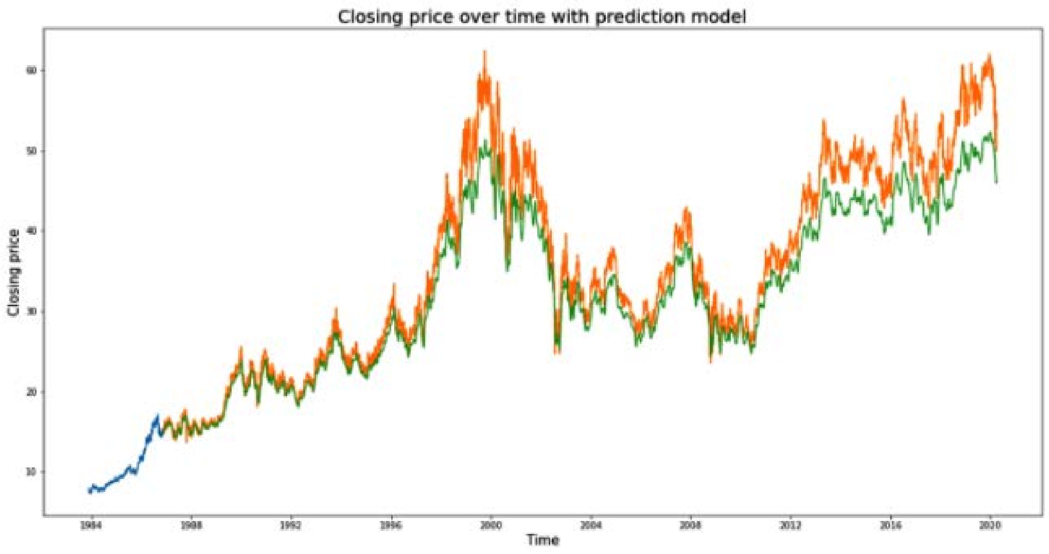


Figure 16.22 – Closing price predictions

As you can see in the preceding graphic, shown in orange here, we have the original, copied values. The green shows the predictions made by our model. They're not terrible, but they're not as tight as they could be.

3. Let's see what happens when we use 7,500 rows of data instead, which is roughly 82% of the data available. As a note, the file in the GitHub repository uses the value of 7,500, but feel free to change and adjust those so you can test the accuracy of your model. The following graph shows our results:

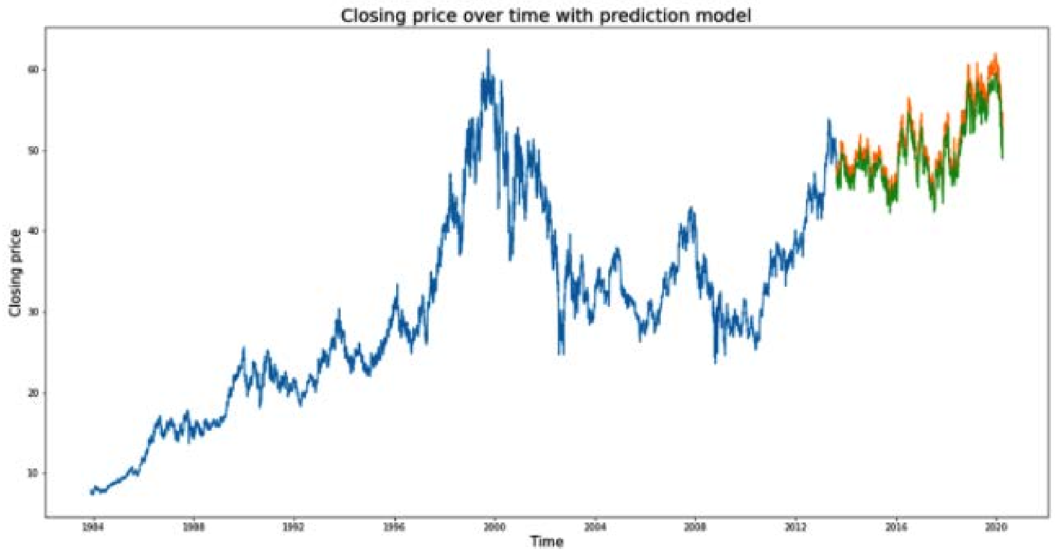


Figure 16.23 – Prediction model using 7,500 rows of data

Notice how much closer the real lines and the prediction lines are in this model. That's because the more data we have to train our model with, the better our predictions will become.

Before we move on from this example, note that we did not cover the entirety of the code file here. Some of the code has been discussed in other areas of this book, so we focused on the parts of the algorithm that were new and critical to the algorithm solution due to the complexity of the algorithm. The final piece of that code file does use a **Long Short-Term Memory (LSTM)** model. An LSTM model is a type of artificial recurrent neural network. We use this model in machine learning to create deep learning models.

Will our models actually predict the price of the stock? No. Otherwise, we'd all have an easier time with the market. But models can get really good at predicting not the price, but whether or not the price will go up or down.

Problem 8 – Using Python to create a convolutional neural network (CNN)

In this section, we're going to take a look at a problem that uses **artificial intelligence** (**AI**). More specifically, we're going to work on creating a **convolutional neural network**, or **CNN**. *So what is a CNN?* A CNN is a **Deep Learning algorithm**. CNNs take images as input. The image is then processed and given importance based on predetermined conditions that will help us differentiate and classify the images.

The following diagram illustrates the process involved in the convolutional neural network:

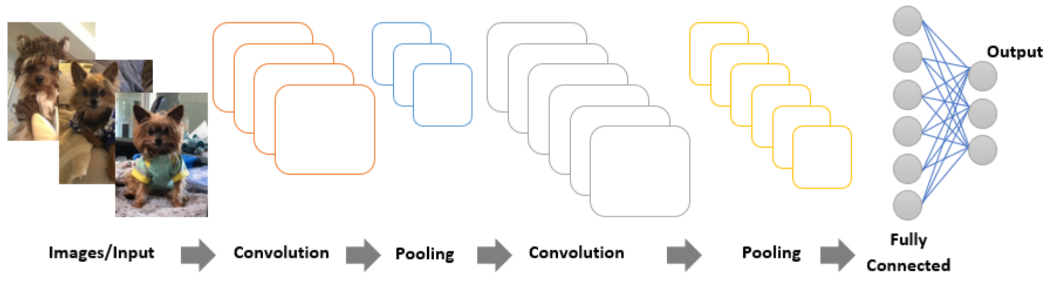


Figure 16.24 – Convolutional neural network process

The CNN is created in order to simplify how we categorize the images without sacrificing accuracy in terms of the predictions we want to be able to get from our image analyses. It's like if we were applying a filter. Once we apply the filter, we can see the characteristics. The preceding diagram shows a simplified schematic for this process.

The problem we're going to be diving into is handwriting training and analysis. Thinking about the computational thinking process, what we're really trying to do is analyze handwriting as accurately as possible. To do so, we analyze hundreds or thousands of images to create and train our model. The more images we use, the more accurate our model will be.

For our model, we're going to use a dataset that contains 70,000 images. The first 60,000 images are used for training, while we'll use the other 10,000 for testing. The full code can be found in the `ch16_CNN_mnist.py` file. We'll take a look at some of the snippets from that code and also use some adjustments to show additional components. You can run the file in the GitHub repository without making changes as long as you have all the necessary libraries and packages installed.

Let's start designing the model:

1. Let's first take a look at a snippet of code that will upload the dataset, and then show the first item in the dataset:

```
from keras.datasets import mnist
#Grab the testing and training data from existing dataset
(X_train, y_train), (X_test, y_test) = mnist.load_data()

import matplotlib.pyplot as plt
#Take a look at the first item in the dataset
plt.imshow(X_train[0], cmap='Greys')
```

   We used the index `0` in the training set to see that first image. The `cmap` property will make the colormap gray. You can adapt that and adjust it, as needed. As a side note, for those who have trouble seeing color or have particular color needs, changing the colormap can make a significant difference to how the images are perceived. Let's take a look at that first image in the following screenshot:

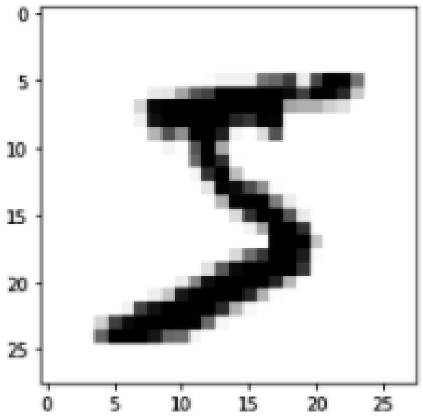


Figure 16.25 – First image in the MNIST training set

   As you can see from the preceding screenshot, this is a handwriting sample, most likely of the number **5**. We can run the program a few more times with different indexes to see some other samples in our dataset. The following screenshot shows some of those samples and their corresponding indexes:

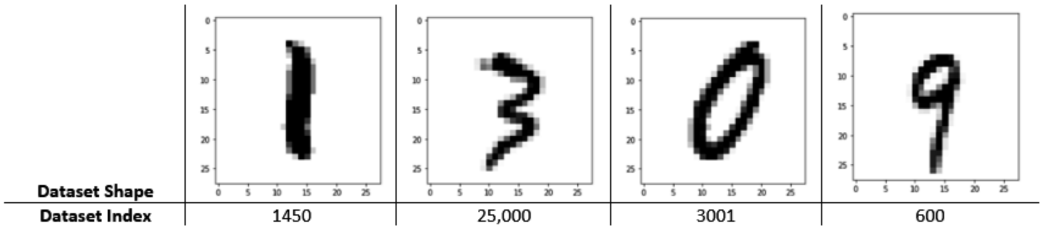


Figure 16.26 – Sample images in the dataset by index

The data we are using is not quantitative, it is qualitative. We are looking at images, so we are in need of a process that can analyze those images. To do so, we use **One-Hot encoding**, which replaces integer encoded variables with new binary variables.

2. Now that we've looked at what we're working with, let's reshape and encode our model using the following code snippet. As a reminder, the full code can be found in the repository, but some of the components will be slightly different (such as our file will not be testing the images in the dataset):

```
#Reshape the model
X_train = X_train.reshape(60000,28,28,1)
X_test = X_test.reshape(10000,28,28,1)

from keras.utils import to_categorical
#Use One-Hot encoding
y_train = to_categorical(y_train)
y_test = to_categorical(y_test)
y_train[0]
```

As you can see, we are dividing the images into training and test sets. Then we encode them.

3. Once we have performed one-hot encoding, we can create our model:

```
#Creating the model
from keras.models import Sequential
from keras.layers import Dense, Conv2D, Flatten

model = Sequential()
model.add(Conv2D(64, kernel_size=3, activation='relu',
input_shape=(28,28,1)))
model.add(Conv2D(32, kernel_size=3, activation='relu'))
model.add(Flatten())
model.add(Dense(10, activation='softmax'))
#Compile the model
model.compile(optimizer='adam', loss='categorical_
crossentropy', metrics=['accuracy'])
```

In the preceding code snippet, we used a `softmax` function. The `softmax` function is sometimes referred to as a normalized exponential function. We use it to normalize the output.

4. Now let's train the model. We're going to first fit the model and then validate the data. Take a look at this code snippet:

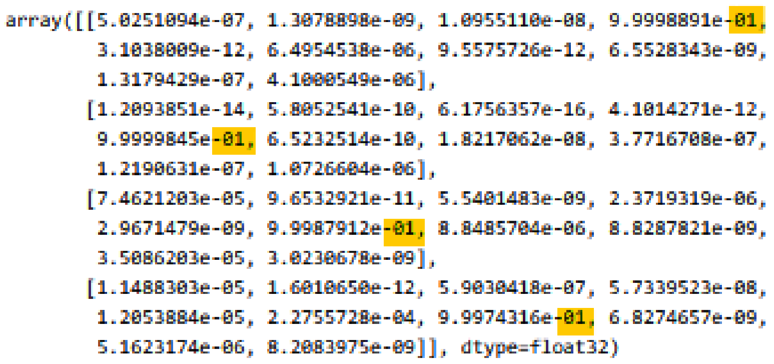
```
#train the model
model.fit(X_train, y_train, validation_data=(X_test, y_
test), epochs=3)
```

So here's one of the great things about training and testing. That is, when we understand and practice with them, we realize it takes just a few lines of code to do some pretty amazing things. The preceding two lines of code (the third is a comment) will make some great things happen and allow our algorithm to test other images.

5. Now we can predict the images in the dataset. We'll start with the last four, because everyone starts with the same four numbers, so I want to start backward this time. As a note, be patient. There are thousands of images to get through in this algorithm. While the epochs are running, you'll have a clock that will tell you how long until the information is processed. For this particular algorithm, it takes just a few minutes. Let's take a look at the snippet of code we'll need in order to run that prediction:

```
#Predict last 4 images
model.predict(X_test[9996:])
```

When we run this code, we get a pretty intense array of numbers. Take a look at the following screenshot. We have highlighted a key piece of the code for discussion:



```
array([[5.0251094e-07, 1.3078898e-09, 1.0955110e-08, 9.9998891e-01,
        3.1038009e-12, 6.4954538e-06, 9.5575726e-12, 6.5528343e-09,
        1.3179429e-07, 4.1000549e-06],
       [1.2093851e-14, 5.8052541e-10, 6.1756357e-16, 4.1014271e-12,
        9.9999845e-01, 6.5232514e-10, 1.8217062e-08, 3.7716708e-07,
        1.2190631e-07, 1.0726604e-06],
       [7.4621203e-05, 9.6532921e-11, 5.5401483e-09, 2.3719319e-06,
        2.9671479e-09, 9.9987912e-01, 8.8485704e-06, 8.8287821e-09,
        3.5086203e-05, 3.0230678e-09],
       [1.1488303e-05, 1.6010650e-12, 5.9030418e-07, 5.7339523e-08,
        1.2053884e-05, 2.2755728e-04, 9.9974316e-01, 6.8274657e-09,
        5.1623174e-06, 8.2083975e-09]], dtype=float32)
```

Figure 16.27 – Model predictions for the CNN image

So, I can tell you that the first number predicted is 3. *How would we know that the number represents 3?* Because each list represents the digits 0 to 9. So imagine replacing the first list with [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]. So if we think of this as indexes, the number with the **01** (which is highlighted in the preceding screenshot) at the end of the number is in index 3, which is the number 3. So our numbers are 3, 4, 5, and 6.

6. *But do you trust the model?* We can just go ahead and return to that snippet of code at the beginning of this discussion and print our results. Remember to change the code slightly to print your test images, and not the training images, as shown in the following code snippet:

```
plt.imshow(X_test[9996], cmap='Greys')
```

When running the code, remember you will need to run it for each of the indexes to see the images. The following screenshot shows the images for each of the relevant indexes of the test images:

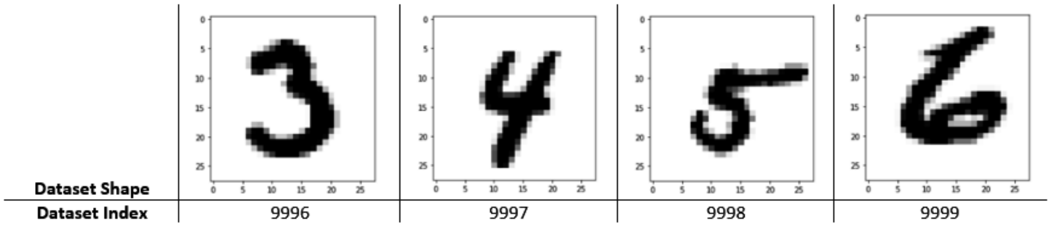


Figure 16.28 – Testing data verification images

As you can see, our model predicted the correct values for each of the handwritten number images in those indexes.

Before we close this discussion, it would be important to note that these models are used fairly extensively right now in websites to verify whether a visitor to the site is human or a bot. Some websites will have a **CAPTCHA**, which sometimes provides handwritten letters that the user must identify to be allowed to proceed. Those CAPTCHAs often use deep learning as well. The applications of CNNs and these kinds of models are endless.

Summary

In this chapter, we were able to explore more topics in computational thinking, especially in dealing with data and deep learning, using the Python programming language. We learned how to create pairplots in order to determine the relationship between variables in a dataset. We also learned how to produce various types of plots to visually represent our datasets. We also learned how to create electric field lines using Python. In short, we applied what we'd learned throughout the previous chapters and extended our knowledge while working in applied problems.

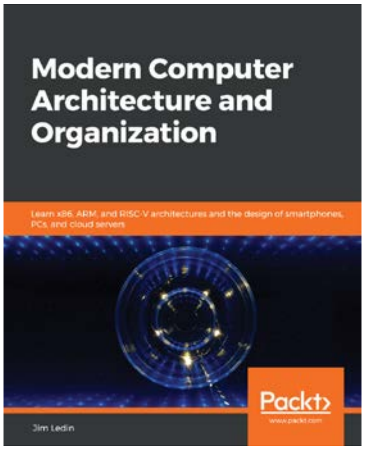
And that's really what this book sought to do: Show a wide variety of Python applications while looking at real problems in context. *Did we cover everything Python can do?* That's fairly impossible, as Python capabilities continue to grow because of its ease of use, how easy it is to learn, and how many applications continue to be added because of its open source nature. Hopefully, you got to work with some new scripts, learned about some of the functions and capabilities you still hadn't explored, and enjoyed exploring these scenarios.

Will we ever be able to say that we've created the perfect algorithm? We, the authors of this book, don't think so. And the reason is that we are always thinking about ways to improve. We always question additional applications. We always want to make them more efficient. And that's really what computational thinking helps us do. We can analyze, design, test, go back, and see whether we did what we wanted, and then refine, redesign, perform tests, and repeat.

Hopefully, after this chapter, you've had a chance to practice and learn about more of Python's capabilities. And hopefully, having read this book, you've had the opportunity to learn about the importance of computational thinking in programming. Thank you for joining us on this journey!

Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:


Modern Computer Architecture and Organization
Learn x86, ARM, and RISC-V architectures and the design of smartphones, PCs, and cloud servers
Packt
www.packt.com
Jim Ledin

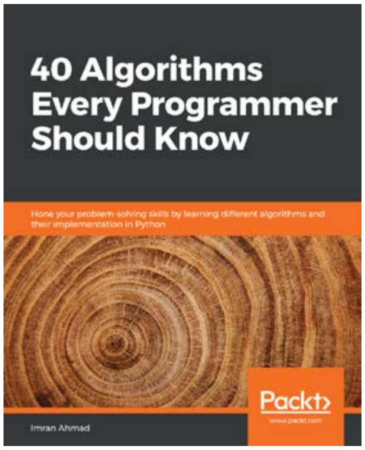
Modern Computer Architecture and Organization

Jim Ledin

ISBN: 978-1-83898-439-7

- Get to grips with transistor technology and digital circuit principles
- Discover the functional elements of computer processors
- Understand pipelining and superscalar execution
- Work with floating-point data formats
- Understand the purpose and operation of the supervisor mode
- Implement a complete RISC-V processor in a low-cost FPGA

- Explore the techniques used in virtual machine implementation
- Write a quantum computing program and run it on a quantum computer


40 Algorithms Every Programmer Should Know
Hone your problem-solving skills by learning different algorithms and their implementation in Python
Packt
www.packt.com
Imran Ahmad

40 Algorithms Every Programmer Should Know

Imran Ahmad

ISBN: 978-1-78980-121-7

- Explore existing data structures and algorithms found in Python libraries
- Implement graph algorithms for fraud detection using network analysis
- Work with machine learning algorithms to cluster similar tweets and process Twitter data in real time
- Predict the weather using supervised learning algorithms
- Use neural networks for object detection
- Create a recommendation engine that suggests relevant movies to subscribers
- Implement foolproof security using symmetric and asymmetric encryption on Google Cloud Platform (GCP)

Leave a review - let other readers know what you think

Please share your thoughts on this book with others by leaving a review on the site that you bought it from. If you purchased the book from Amazon, please leave us an honest review on this book's Amazon page. This is vital so that other potential readers can see and use your unbiased opinion to make purchasing decisions, we can understand what our customers think about our products, and our authors can see your feedback on the title that they have worked with Packt to create. It will only take a few minutes of your time, but is valuable to other potential customers, our authors, and Packt. Thank you!

Index