INTRODUCTION TO COMPUTER PROGRAMMING WITH PYTHON

Harris Wang

Introduction *to* Computer Programming *with* Python

This page intentionally left blank

Introduction *to* Computer Programming *with* Python

Harris Wang

Remix
Athabasca University

Copyright © 2023 Athabasca University
1 University Drive, Athabasca, AB Canada
DOI: https://doi.org/10.15215/remix/9781998944088.01

Published by Remix, an imprint of Athabasca University Press. For more information, please visit aupress.ca or email OERpublishing@athabascau.ca.

Cover design by Lisa Mentz
Images that appear in this text are created by the author.

Library and Archives Canada Cataloguing in Publication

Title: Introduction to computer programming with Python / Harris Wang.
Names: Wang, Harris, author.
Identifiers: Canadiana (print) 20230509916 | Canadiana (ebook) 20230509924 | ISBN 9781998944071 (softcover) | ISBN 9781998944088 (PDF) | ISBN 9781998944095 (EPUB)
Subjects: LCSH: Python (Computer program language)—Textbooks. | LCGFT: Textbooks.
Classification: LCC QA76.73.P98 W36 2023 | DDC 005.13/3—dc23

Introduction to Computer Programming with Python by Harris Wang is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License, except where otherwise noted. This license allows users to copy and redistribute the material in any medium or format and to remix, transform, and build upon the material as long as the original source is properly credited, the work is not used for commercial purposes, and the new creation is licensed under the same terms.

Remix name, Remix logo, and Remix book covers are not subject to the Creative Commons license and may not be reproduced without the prior and express written consent of Athabasca University.

Contents

Chapter 1 Introduction 1

Learning Objectives 1

1.1 A Brief History of Computers 1

1.2 Fundamentals of Computing and Modern Computers 5

Number Systems and the Foundation of Computing 6

Computability and Computational Complexity 9

The Construction of Modern Computers 11

Analog Computers 11

Digital Computers 12

Mechanic-Based Components 13

Vacuum Tube–Based Components 14

Transistors 14

Integrated Circuits and Very Large-Scale Integrated Circuits 14

1.3 Programming and Programming Languages 15

1.4 Python Programming Language 17

The Development and Implementation of Python 17

Advantages of Python 18

Resources for Python and Python Education 19

1.5 Getting Ready to Learn Programming in Python 20

Installing and Setting Up the Python Programming Environment 21

Installing Python 21

Setting Up a Virtual Environment for a Python Project 24

Installing Jupyter Notebook 26

Installing Visual Studio Code 27

Additional Tools Supporting Software Development in Python 27

Buildbot 27

Trac 28

Roundup 28

1.6 Getting a Taste of Programming with Python 28
Program Interactively with Python Interactive Shell 28
Program with VS Code IDE 30
Use Jupyter Notebook Within VS Code to Program Interactively 32
Write Documentation in Markdown 33
Headings 33
Paragraphs 34
New Lines 34
Italic, Bold, and Strikethrough Texts 34
Horizontal Rules 35
Keyboard Keys 35
Unordered Lists 35
Ordered Lists 36
Definition Lists 36
Links 37
Links to Internal Sections 37
Images 38
Blockquotes 38
Tables 39
Inline Program / Script Code 39
Code Block 40
Mathematical Formulas and Expressions 40
To-Do List 42
Escape Sequence for Special Characters 42
Programming Interactively with Jupyter Notebook Within VS Code 42
Run Python Programs Outside IDE 48
Make the Python Program File Executable 50
Errors in Programs 52
1.7 Essentials of Problem Solving and Software Development 54
Design Algorithms to Solve Problems 54
Phases of Software System Development 56
Phase 1. Understand the Project 57
Phase 2. Analyze the Requirements to Identify Computer-Solvable Problems and Tasks 57
Phase 3. Design the System 58
Phase 4. Implement the System 58
Phase 5. Test the System 58
Phase 6. Maintain the System 58

1.8 Manage Your Working Files for Software Development Projects 58
Set Up Git on Your Computer and Version-Control Locally 59
Set Up an Account on GitHub and Version-Control with Remote Repositories 62
Chapter Summary 67
Exercises 69
Projects 71

Chapter 2 Essential Building Blocks of Computer Programs 73

Learning Objectives 73
2.1 Primary Constructs of Computer Programs in Python 74
Vocabulary of the Programming Language 74
Rules of Naming Identifiers 76
Python Naming Conventions 79
Names with Leading and/or Trailing Underscores 79
Rules of Scope Resolution for Identifiers 81
Simple Data Types 82
Signed Integers (int) 83
Float (float) 87
Boolean (bool) 88
Complex (complex) 89
Compound Data Types 89
String (str) 90
List 95
Tuple 97
Set 98
Dictionary 99
Object 99
Variables and Constants 101
Variables 101
Built-In Constants 104
Operators 106
Arithmetic Operators 106
Comparison Operators 108
Logical Operators 109
Bitwise Operators 111
Assignment Operators 112
Identity Operators 115

Sequence Operators 115
Membership Operator 116
Built-In Functions 117
Expressions 135
2.2 Higher-Level Constructs of Python Programs 137
Structure of Python Programs 137
Documentation and Comments 139
Simple Statements 141
Expression Statement 141
Assignment Statement 143
print Statement 146
input Statement 149
assert Statement 150
pass Statement 151
del Statement 151
return Statement 152
open Statement 152
yield Statement 153
raise Statement 154
break Statement 155
continue Statement 155
import Statement 156
global Statement 156
nonlocal Statement 157
help Statement 158
Compound Statements 159
Code Blocks 159
Rules of Indentation 160
Rules of Spacing 160
if Statement 161
if-else Statement 162
if-elif Statement 163
if-elif-else Statement 165
while Statement 165
for Statement 166
def Statement 167
class Statement 168
try-except Statement 168
with Statement 169
Chapter Summary 170

Exercises 171
Projects 172

Chapter 3 Flow Control of Statements 175

Learning Objectives 175
3.1 Selective with the *if* Statement 175
3.2 Single-branch selective with *if* Statement 176
3.3 Multiple-Branch Selective with *if-elif-…* and *if-elif-…-else* Statements 177
3.4 Iterate with *for* Statement 180
Using *break* and *continue* Statements and an *else* Clause Within Loops 184
Common Coding Mistakes with the *for* Loop 185
3.5 Iterate with the *while* Statement 185
Common Coding Mistakes with a *while* Loop 190
3.6 Iterate with *for* Versus *while* 190
Chapter Summary 195
Exercises 195
Projects 196

Chapter 4 Handle Errors and Exceptions in Programs 201

Learning Objectives 201
4.1 Errors in Your Programs 202
Exception 203
ArithmeticError 204
OverflowError 204
ZeroDivisionError 204
FloatingPointError 205
AssertionError 205
AttributeError 205
BufferError 206
EOFError 206
GeneratorExit 206
ImportError 206
IndexError 207
KeyError 207
KeyboardInterrupt 207
MemoryError 207

ModuleNotFoundError 207
NameError 208
NotImplementedError 208
OSError 208
BlockingIOError 208
ChildProcessError 208
ConnectionError 208
BrokenPipeError 209
ConnectionAbortedError 209
ConnectionRefusedError 209
ConnectionResetError 209
FileExistsError 209
FileNotFoundError 209
IsADirectoryError 209
NotADirectoryError 209
PermissionError 209
ProcessLookupError 210
TimeoutError 210
RecursionError 210
ReferenceError 210
RuntimeError 210
StopIteration 210
StopAsyncIteration 210
SyntaxError 210
IndentationError 211
TabError 211
SystemError 211
SystemExit 211
TypeError 211
UnboundLocalError 212
UnicodeError 212
UnicodeEncodeError 212
UnicodeDecodeError 212
UnicodeTranslateError 212
ValueError 212
4.2 Handling Runtime Errors and Exceptions 216
Chapter Summary 219
Exercises 220

Chapter 5 Use Sequences, Sets, Dictionaries, and Text Files 221

Learning Objectives 221
5.1 Strings 222
Methods of Built-In Class str 222
Built-In Functions and Operators for Strings 232
Constructing and Formatting Strings 234
Regular Expressions 242
5.2 Lists 251
5.3 Tuples 258
5.4 Sets 261
5.5 Dictionaries 266
5.6 List, Set, and Dictionary Comprehension 271
List Comprehension 272
Set Comprehension 273
Dictionary Comprehension 274
5.7 Text Files 274
Opening and Closing a File 275
Write or Append to a File 278
Reading from a File 280
Update Existing Content of a Text File 284
Deleting Portion of a Text File 286
Chapter Summary 289
Exercises 291
Projects 292

Chapter 6 Define and Use Functions 295

Learning Objectives 295
6.1 Defining and Using Functions in Python 296
6.2 Parameters and Arguments in Functions 299
6.3 Recursive Functions 304
6.4 Anonymous Functions: *lambda* Expressions 308
6.5 Special Functions: Mapping, Filtering, and Reducing 310
Mapping 310
Filtering 311
Reducing 312
6.6 Generators: Turning a Function into a Generator of Iterables 312
6.7 Closures: Turning a Function into a Closure 316
6.8 Decorators: Using Function as a Decorator in Python 317

6.9 Properties of Functions 320
Chapter Summary 322
Exercises 322
Projects 323

Chapter 7 Object-Oriented Programming with Python 325

Learning Objectives 325
7.1 Introduction to Object-Oriented Programming (OOP) 326
Abstraction 326
Information Hiding or Data Encapsulation 326
Inheritance 327
7.2 Defining and Using Classes in Python 327
Inheritance: Subclass and Superclass 333
Public, Private, and Protected Members of a Class 334
Class Methods 336
Static Methods 337
Class Attributes 338
7.3 Advanced Topics in OOP with Python 340
Dunder Methods in Class Definition 340
Using Class as Decorator 346
Built-In Property() Function and Property Decorator 348
Creating a New Class Dynamically and Modify a Defined Class or Instance 352
Keeping Objects in Permanent Storage 356
Chapter Summary 358
Exercises 359
Project 361

Chapter 8 Modules and Packages 363

Learning Objectives 363
8.1 Creating Modules and Packages 364
8.2 Using Modules and Packages 367
8.3 Install and Learn About Modules Developed by Others 369
8.4 Module for Generating Random Numbers 377
Functions for Bookkeeping 379
Functions for Generating Random Integers 379
Functions for Randomly Generating Float Numbers 380
Functions for Randomly Selected Item(s) from Sequences 382
8.5 Module for Mathematical Operations 385

8.6 Modules for Time, Date, and Calendar 395
The Datetime Module 395
The Time Module 405
The Calendar Module 410
8.7 Modules for Data Representation and Exchange 415
8.8 Modules for Interfacing Operating Systems and Python Interpreter 418
OS Module for Interacting with the Operating System 418
The path Submodule from os for Manipulating File Paths 423
The sys Module for Interaction Between the Python and Python Interpreter or Python Virtual Machine (PVM) 426
8.9 Module for Logging Events During Program Runtime 434
8.10 Modules for Playing and Manipulating Audio and Video Files 436
winsound 436
PyGame 439
8.11 Modules for Creating and Manipulating Graphics and Images 442
Create Graphics with Tkinter 443
Manipulate Images with Pillow 448
8.12 Modules for Data Analytics 451
Chapter Summary 455
Exercises 456
Projects 457

Chapter 9 Develop GUI-Based Applications 459

Learning Objectives 459
9.1 Terminal-Based Applications Versus GUI-Based Applications 460
9.2 Designing and Developing GUI-Based Applications in Python 461
Tkinter Module 463
tkinter.ttk—Tk-Themed Widgets 475
Chapter Summary 482
Exercises 483
Projects 484

This page intentionally left blank

Chapter 1

Introduction

This chapter prepares you to learn how to program with Python. Preparation includes a brief introduction to computers and computing, programming, and programming languages, as well as the installation of Python and Python interactive programming and integrated development environments (IDEs), including Jupyter Notebook for interactive programming and VS Code as a required IDE.

Learning Objectives

After completing this chapter, you should be able to

- talk about the history of computers.
- describe the basic components of modern computers and their roles.
- explain the basic principles of modern computers.
- discuss the basics of computability and computational complexity.
- explain how the construction of modern computers has evolved.
- explain what computer systems are made of.
- discuss computer programming languages.
- describe Python and discuss its development, features, and advantages.
- install Python and required Python IDEs on the computer.
- get Python and Python IDEs running for the learning activities included in this textbook.

1.1 A Brief History of Computers

The history of human tools, devices, or instruments to help us count, compute, and think can be traced back to the Stone Age, when our ancestors used knots on ropes, marks on bark, stones, and balls of clay. One of the most well-known

https://doi.org/10.15215/remix/9781998944088.01

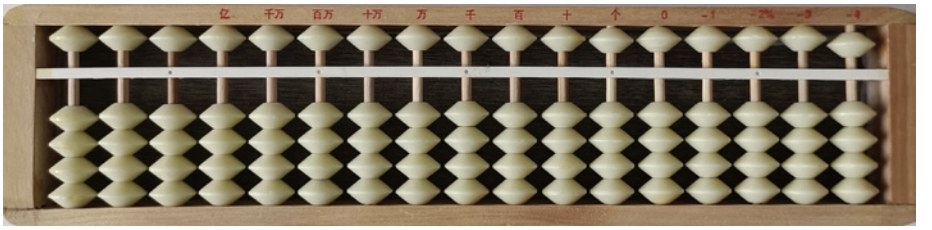


Figure 1-1: A traditional abacus

and widely used devices for computing in human history is the abacus, shown in Figure 1-1.

The exact origin of the abacus is unknown, but before the adoption of the written Hindu-Arabic numeral system, the abacus had already been widely used in many countries, including China, Russia, and some European countries. Abacuses continued to be widely used by almost all accountants in China before the adoption of modern computers and calculators.

Today, although abacuses are rarely used in real applications such as accounting, their working principles are still used to train people to do mental math, such as in programs like the Universal Concept of Mental Arithmetic System (UCMAS; see ucmas.ca). UCMAS holds annual competitions worldwide, including in Canada.

The invention and development of today's modern computers can be attributed to the invention and development of all previous relevant concepts, principles, and technologies. The concept of a digital, programmable computer originated with Charles Babbage, an English mathematician, philosopher, inventor, and machine engineer. In 1822, he designed a steam-driven calculating machine for automatically computing tables of numbers. Although his government-funded project failed, the many ideas he developed and used in his analytical engine were adopted and used in the world's first modern programmable computer, built a century later. That is why Charles Babbage is considered to be one of the "fathers of computers."

In 1847, George Boole, an English mathematician introduced Boolean logic, propositional calculus, and Boolean algebra as the laws of thinking, which later became the foundation of modern electronic computers and digital devices in general. For that reason, George Boole is regarded as a founder of computer science.

In 1890, Herman Hollerith successfully designed and built a punch-card-based system for the US government to calculate its 1890 census. It saved the US government US$5 million by finishing the calculation in one year instead of 10 years it would have taken using traditional methods, as in the previous census. In 1896, Herman Hollerith established a company called the Tabulating Machine Company to make the machines. This company ultimately became IBM.

https://doi.org/10.15215/remix/9781998944088.01

The central concept and theory of modern computers were conceived in 1936 by English mathematician Alan Turing in his "universal machine," which became known as the Turing machine in his honour. He successfully proved that his universal machine could calculate anything that is computable. Alan Turing is also considered by some to be "a father of computers"—or more precisely, "a father of computing"—for his seminal paper on the theory of computation. In the eyes of the general public, Alan Turing is more famous for his role in cracking the so-called unbreakable codes used by the German army during World War II, as presented in *The Imitation Game*, a well-known Hollywood movie.

Until 1937, all computing machines or computers were mechanically based, using gears, cams, belts, or shafts. John Vincent Atanasoff, an American mathematician and physician at Iowa State University, attempted in 1937 to build the first electronic computer. He and Clifford Berry, one of his graduate students at the time, designed and built a special-purpose digital computer, ABC. For that reason, he is considered to be "the father of the modern computer." It was in the ABC machine that binary math and Boolean logic were successfully used for the first time.

In 1943, across the Atlantic Ocean in England, Colossus Mark I, a prototype of a special-purpose computer, was built. A year later, in 1944, Colossus Mark II was built and used to break the encrypted radiotelegraphy messages transmitted by the German army at the end of World War II.

In 1946, the first general-purpose digital computer, the Electronic Numerical Integrator and Computer (ENIAC), was built by two professors at the University of Pennsylvania: John Mauchly and J. Presper Eckert. The computer had over 18,000 vacuum tubes and weighed 30 tons. A rumour at the time said that whenever the computer was turned on, the lights in some districts of Philadelphia would dim.

During the same time, the same group of people at the University of Pennsylvania also designed and built EDVAC (the Electronic Discrete Variable Automatic Computer, completed in 1949), BINAC (the Binary Automatic Computer, also completed in 1949), and UNIVAC I (the Universal Automatic Computer I, completed in 1950), which were the first commercial computers made in the US. Although the computing power of these computers was not even comparable to that of today's smartphone, their contributions to the development of today's modern computers are enormous. These contributions include the concepts, principles, and technology of stored programs, subroutines, and programming languages.

Around the same time that ENIAC was built, the Harvard Mark I automatic sequence-controlled calculator was also built at Harvard University. It is

https://doi.org/10.15215/remix/9781998944088.01

believed that John von Neumann, a mathematician and physicist working at the Los Alamos National Laboratory, was the first user of the Harvard Mark I, which he used to run programs on the machine to calculate for the Manhattan Project.

In 1944, Von Neumann had an opportunity to join a discussion with J. Presper Eckert and John Mauchly, who were developing ENIAC and EDVAC machines at the time. He wrote up a report on the EDVAC, *First Draft of a Report*, in which he described a computer architecture later known as Von Neumann architecture (see Figure 1-2). The key idea behind the architecture is a stored-program computer that includes the following components:

- a processing unit that contains an arithmetic logic unit and processor registers
- a control unit that contains an instruction register and program counter
- internal memory that stores data and instructions
- external mass storage
- input and output mechanisms

The unpublished report was widely circulated. It had great impact on the later development of modern computers, although to many people, Turing was the originator of this key idea and architecture, and many others, including J. Presper Eckert and John Mauchly, made significant contributions to its evolution as well. Regardless, it is Von Neumann architecture that has guided the design of modern computers. All computers we use today are essentially still Von Neumann computers.

Modern computers can be categorized into three generations based on the core hardware technology used. The first generation of computers (1937–46) used vacuum tubes, the second generation (1947–62) used transistors, and the third generation (1963–present) used integrated circuits (IC).

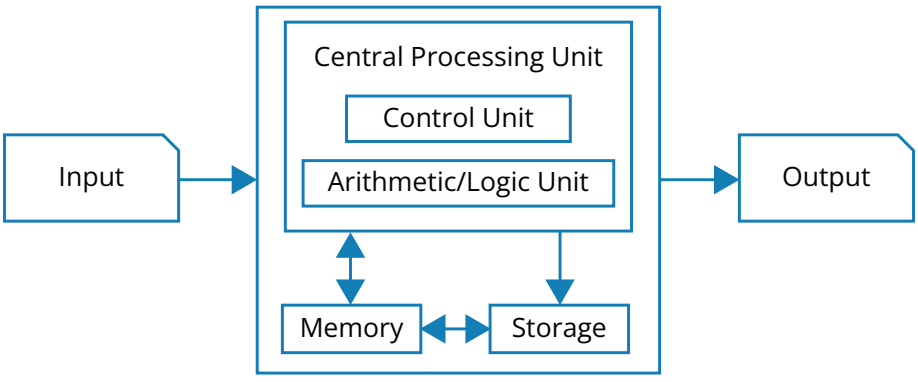


Figure 1-2: Von Neumann architecture

https://doi.org/10.15215/remix/9781998944088.01

Both hardware and software play important roles in computers. On the hardware front, in the first generation of modern computers, magnetic tapes and disks were developed and used for storage, and printers appeared and were used for output. On the software side, simple operating systems (OSs) were developed and used, and over 100 high-level programming languages were developed.

During the second generation of computers, the Universal Automatic Computer (UNIVAC I), the first computer for commercial use, was introduced to the public (1951). The International Business Machines Corporation (IBM) also brought their IBM650 and IBM700 series computers to the computer world (1953).

Since 1963, the beginning of the third generation of modern computers, advances in hardware have made computers smaller and smaller but much more powerful, with a higher speed of CPU and a bigger memory with faster throughput.

In 1980, Microsoft developed MS-DOS, short for Microsoft Disk Operating System, and in 1981, IBM introduced the first personal computer (PC) for home and office use. In 1984, Apple brought its Macintosh computer with its icon-driven interface to the world, and in the 1990s, Microsoft brought the world the Windows operating system. Billions of computer users around the world have enjoyed both Microsoft Windows and Apple's macOS.

More information about the history of computers can be found on the internet by searching for "history of computers," including the following articles:

https://en.wikipedia.org/wiki/Computer
https://www.computerhistory.org/timeline/computers/
https://www.explainthatstuff.com/historyofcomputers.html
https://www.britannica.com/technology/computer/History-of-computing

1.2 Fundamentals of Computing and Modern Computers

The development of modern computers is the result of the collective efforts of many great mathematicians, scientists, and engineers and advances in both theories and technologies. This section looks at the theoretical and technical fundamentals of computing and modern computers in order to show how modern computers work internally.

https://doi.org/10.15215/remix/9781998944088.01

Number Systems and the Foundation of Computing

Initially, computers and computing devices were developed to deal with numbers. They later made their way into the realm of text handling and information processing. This is done through encoding. When properly coded, all text can be represented as numbers—codes—and all ways of manipulating text can be accomplished through operations on those codes. Number systems and operations on numbers are really a basic foundation of computing and modern computers.

We all know numbers, and the number system we have known since childhood is base-10, which is represented using 10 digits from 0 to 9. A base-10 number such as 2539867 can be written as

$$2 * 10^6 + 5 * 10^5 + 3 * 10^4 + 9 * 10^3 + 8 * 10^2 + 6 * 10^1 + 7 * 10^0$$

In general, an n-digit number $d_{n-1}d_{n-2}...d_1d_0$ (where n is any positive integer and each $d \in [0, 1, 2, ..., 9]$, where $0 < i < n$) in a base-10 number system can be rewritten as

$$d_{n-1} * 10^{n-1} + d_{n-2} * 10^{n-2} + ... + d_1 * 10^1 + d_0 * 10^0$$

In fact, a number system can be based on any whole number other than 0 and 1. There is a base-7 system for weeks, a base-12 system for the Chinese zodiac and imperial foot and inch, and a base-24 number system for the 24-hour clock. A long list of different numbering systems that have appeared since prehistory can be found at https://en.wikipedia.org/wiki/List_of_numeral_systems.

In general, a base-Q number system will require Q unique symbols representing 0, 1, ... Q–1, respectively. The base-10 equivalence of an n-digit base-Q number $d_{n-1}d_{n-2}...d_1d_0$ can be represented as

$$d_{n-1} * Q^{n-1} + d_{n-2} * Q^{-2} + ... + d_1 * Q^1 + d_0 * Q^0$$

The evaluation of the expression above will result in its base-10 equivalence. This is how we convert a number from base-Q to base-10.

For Q = 2, the numbers 0 and 1 are used to represent digits; for Q = 3, the numbers 0, 1, and 2 are used; for Q = 8, the numbers 0, 1, 2, 3, 4, 5, 6, and 7 are used; for Q = 16, the numbers 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a/A, b/B, c/C, d/D, e/E, and f/F are used.

The expression above also shows how to convert a number from base-10 to base-Q: divide the number by Q repeatedly, and the remainders will be the digits. The details of these conversions can be found on the internet.

For a number $d_{n-1}d_{n-2}...d_1 d_0$ in a base-Q number system, it is often necessary to add Q as a subscription of the sequence, as shown below:

$$d_{n-1}d_{n-2}...d_1d_{0Q} \text{ or } (d_{n-1}d_{n-2}...d_1d_0)_Q$$

https://doi.org/10.15215/remix/9781998944088.01

This explicitly indicates that the number is in a base-Q number system. For example, number 657 in a base-8 number system will be written as 657_8 or $(657)_8$, especially if the context does not indicate which number system the number is in.

All number systems have the same operations that we are already familiar with in a base-10 system, such as addition, subtraction, multiplication, and division. The only difference is that in a base-Q number system, a 1 carried to or borrowed from a higher digit is equal to Q instead of 10.

For example, in a base-2 number system, 101 can be written as 1 * 22 + 0 * 21 + 1 * 20, and its 10-base equivalence is 1 * 4 + 0 + 1 = 5, and for 101 + 110, the operation is

```
  101          110
+ 110        - 101
 1011          001
```

Given the four basic mathematical operations—addition, subtraction, multiplication, and division—it is very simple to prove that multiplication can be done by repeated addition, whereas division can be done by repeated subtraction. Therefore, if you know how to do addition and subtraction and are able to count, you will be able to do multiplication and division as well.

More importantly, because a − b can be rewritten as a + (−b), you may be able to eliminate subtraction if you can represent −b without the minus sign. In a base-Q number system, (a − b) can be conveniently handled through a + b′, in which b′ is the Q-complement of b representing (−b).

Given a negative number, −N, in which N is called the magnitude of −N, how do you get N′, the Q's complement to N? It turns out to be very easy.

For a number N written as $d_n d_{n-1} \ldots d_0$ in a base-Q number system, its Q-complement N′ is a number, also in a base-Q number system, such that $N + N' = Q^n$, and N′ can be easily calculated in the following two steps:

1. For each d_i in $d_{n-1} d_{n-2} \ldots d_0$, find t_i such that $d_i + t_i = Q-1$, to get $t_{n-1} t_{n-2} \ldots t_0$
2. Add 1 to $t_{n-1} t_{n-2} \ldots t_0$ to get N's complement $c_{n-1} c_{n-2} \ldots c_0$

For b in a − b, if b has fewer digits than a, add 0s to the left of b to make up the missing digits before working on the above steps. For example, for 768 − 31, do 768 + (−31) or 768 + the 10's complement of 31.

https://doi.org/10.15215/remix/9781998944088.01

To get 10's complement of 31, first add one 0 to the front of 3 to get 031, then find out the 10's complement of 031, which is 968 + 1 = 969. Then do 768 + 969 to get 1737, and ignore the last carried digit, 1, to get 737, which is the result of 768 − 31.

It is easy to see that if C is N's complement, then N is C's complement as well. Using the notation given above, this can be written as $(N')' = N$.

What about floating-point numbers—that is, numbers with fractions, such as 3.1415926?

In a base-Q number system, a floating-point number can be represented much like integers, as shown below:

$$d_{n-1} * Q^{n-1} + d_{n-2} * Q^{n-2} + \ldots + d_1 Q^1 + d_{-1} * Q^{-1} + d_{-2} * Q^{-2} + \ldots + d_{-m} Q^{-m}$$

This may give us a hint that a floating-point number can be represented by two sequences of digits: one for the integer portion and the other for the fraction portion. If floating-point numbers are represented this way in computers, addition and subtraction could be performed on the two portions separately, but with a mechanism to deal with carrying from the fraction to the integer in the case of addition, and a mechanism to deal with borrowing from the integer to the fraction in the case of subtraction. This would be rather clumsy, and it would be even more clumsy when doing multiplication and division.

In reality, floating-point numbers are represented in scientific notation, such as $1.3 * 10^2$ in a base-10 or decimal system, where 1.3 is called the fraction, 2 is called the exponent, and 10 is the radix. In a base-2 system, the radix would be 2.

In scientific notation, arithmetic operations on floating-point numbers can be easily done, especially if the fractions of all floating-point numbers are normalized—that is, if the number of digits before the radix point is fixed for all floating-point numbers. For example, to perform the addition or subtraction of two floating-point numbers, all you need to do is shift the digits of one floating-point number based on the difference of the two exponents, then perform addition or subtraction in the same way as on integers. More surprisingly, multiplying and dividing floating-point numbers in scientific notation is much easier than adding and subtracting. There is no need to shift the digits because

$$a * r^m * b * r^n = a * b * r^{m+n}$$

and

$$a * r^m / b * r^n = a / b * r^{m-n}$$

All you need to build a computer capable of adding, subtracting, multiplying, and dividing in a base-Q number system is the ability to add and count,

https://doi.org/10.15215/remix/9781998944088.01

which is needed for multiplication and division. Moreover, counting itself can be done by addition when counting up or subtraction when counting down. This is very encouraging.

The following discoveries are even more encouraging:

- Many mathematical and scientific problems can be represented and reformulated based on the basic mathematical operations discussed above.
- If we can build a computing device capable of doing basic mathematical operations, the machine can solve many computing and information processing problems.

The remaining question is how to make such a machine work faster. On a computing machine with only an addition and counting unit, if each step of an operation has to be enabled by a human, the computation would be very slow because it would have to wait for the human to input instructions and data. To speed things up, the entire problem solving process needs to be automated. This requirement has led to the introduction of memory to computing machines to store both the operating instructions (the program) and data needed during the computation. For the central processing unit (CPU) and memory (RAM) to communicate and work together, a communication channel and control unit need to be added. For human users to communicate with the computing machine, input and output units are also needed. Together these include everything described in Von Neumann's architecture for modern computers.

Computability and Computational Complexity

Now comes the question of how powerful the computing machine described above can be. In particular, we should ask,

- What problems can or cannot be computed with a computing machine?
- How difficult is it to solve a given problem with a computing machine, assuming that the machine can take as many steps and use as much memory as needed to solve the problem?

Answering the first problem is the study of computability. Answering the second question is the study of computational complexity.

A well-known computability problem was raised by David Hilbert, a very famous German mathematician in the 19th century. The 10th problem in his list of 23 important mathematical problems he published in 1900 is the decidability problem, which is about finding an algorithm that can determine whether a

https://doi.org/10.15215/remix/9781998944088.01

Diophantine equation such as $x^n + y^n = z^n$, where n is an integer and greater than 2, has integer solutions. It was proved in 1970 that such an algorithm doesn't exist, meaning that the problem of deciding whether a Diophantine equation has integer solutions is unsolvable, or incomputable.

Between the 1930s and 1940s, before the existence of modern computers, Alan Turing invented a simple yet powerful abstract machine called the Turing machine. He proved that the abstract machine could simulate the logic of any given algorithm and further determine whether an algorithm is computable. The operation of the machine will halt if the algorithm is incomputable. During the same time, computability was also well studied by Alonzo Church based on lambda calculus. The two works later intertwined in a formal theory of computation known as the Church-Turing thesis.

In computer science, the computational complexity of an algorithm devised to solve a problem is about the number of resources—CPU time and memory, in particular—needed to run the algorithm on a computer. A problem may be solved in different ways using different algorithms with different computational complexities. Sometimes it is necessary to find a better algorithm that runs faster and/or uses less memory.

The time required to run an algorithm is also related to the size of the problem to be solved. For example, finding someone in a classroom would be much easier than finding someone in a city. If you denote the size of a problem with n, then in computer science, O(f(n))—called the big O of f(n)—is used to describe asymptotically the upper bound of running time: the time complexity of an algorithm for solving the problem.

For example, assume you want to sort the numbers in a list. The size of the problem will be the number of numbers in the list. One algorithm is called selection sort, which begins with selecting the smallest number from the list, then selecting the smallest number from the remaining of the list, and so on. Assume the size of the list is n. The first time it will need to do n − 1 comparisons. The second time it will need n − 2 comparisons. When these are only two numbers left in the list, only one comparison is needed to finish. So the complexity of the algorithm in terms of the total number of comparisons needed will be

$$(n - 1) + (n - 2) + (n - 3) \ldots + 2 + 1 = (n - 1 + 1) / 2 * (n - 1) = n * (n - 1) / 2 = (n^2 - n) / 2 = O(n^2)$$

You may have noted in the big-O notation that we have kept only the highest-order term and have removed the constant ½ as well. This is because, with the big-O asymptotic notation, only the scale of complexity increased when the size of the problem increases is of interest.

https://doi.org/10.15215/remix/9781998944088.01

In computer science, problems can be classified as P (for polynomial), NP (for nondeterministic polynomial), NP-complete, or NP-hard problems. A problem is said to be in P if it can be solved using a deterministic algorithm in polynomial time—that is, if the complexity is a big O of a polynomial. A problem is said to be in NP if it can be solved with a nondeterministic algorithm in polynomial time. A problem is NP-complete if a possible solution can be quickly verified but there is no known algorithm to find a solution in polynomial time. A problem is NP-hard if every NP problem can be transformed or reduced to it within polynomial time.

In the above, a deterministic algorithm refers to the one in which, given the same input, the output would always be the same, whereas a nondeterministic algorithm may give different outputs even for the very same input.

The complexity of algorithms, which is closely related to the response and processing speed of programs and computer systems, is the concern of good programmers, especially when dealing with resource-intensive applications. It is also an important topic to be studied in senior or graduate courses in computing and computer science. When writing a program or just a function for assignments or an application, always think about the time and space complexity and ask yourself, Is there a better way to get the job done?

The Construction of Modern Computers

In general, computers can be categorized into two types: analog computers and digital computers. The computers in use today are almost all digital computers. But analog computers do have their advantages.

ANALOG COMPUTERS

As a computing machine, an analog computer uses the properties of certain objects or phenomena to directly analogize the values of variables (such as the speed of a vehicle) of a problem to be solved. Examples of the properties include the voltage and amperage of electricity, the number of teeth of a gear, or even the length of a wood or metal stick.

Analog computing machines were often built for some special purpose, with a very wide range of difficulties and complexities. A simple analog computer could be built with three gears for addition and subtraction. For addition, using both operands to drive the third gear in the same direction will yield the sum of the two operands on the third gear, whereas for subtraction, the difference can be calculated by driving the two gears in two different directions.

Historically, even as early as 100 BC, complicated analog computing machines were built for various applications, from astronomy in ancient Greece to the differential machine for solving differential equations in the late 1800s

https://doi.org/10.15215/remix/9781998944088.01

and early 1900s. Some of the most complicated analog computing machines in modern history include those for flight simulation and gunfire control. Analog computing machines continued well into the early age of modern digital computers because the special applications analog computers were developed for were still too hard for digital computers in the 1960s or even 1970s.

DIGITAL COMPUTERS

Different from analog computers, digital computers use sequences of digits to represent the values of variables involved in the problems to be solved. In digital computers, the machine representation of a problem is often abstract, with no analogy to the original problem.

Theoretically, in accordance with the earlier discussion about general number systems, digital computers' digits can be in any base, from 2 up. Hence a digital computer could be base-2, base-3, ... base-10, base-16, and so on. The digital computers we are using today are all base-2, or binary, computers, although it has been proved that base-3 computers would be even more efficient than base-2 computers.

This is because it is more convenient and cheaper to build components with the two easily distinguishable states needed to represent base-2 numbers.

Also, in a binary system, signed numbers can be naturally represented using the highest sign bit: 0 for positive numbers and 1 for negative numbers, for example. Moreover, the addition and subtraction of signed binary numbers can be easily done by using 2's complements to represent negative numbers.

For example, to do −2−3, we would do

$$\mathbf{(-2) + (-3)}$$

or

$$\mathbf{(-00000010)_b + (-00000011)_b}$$

Next, we need to convert $(-00000010)_b$ and $(-00000011)_b$ into their 2's complement representations. According to the two steps we described in the section above, "Number Systems and the Foundation of Computing," with Q = 2, you first get 1's complement of the magnitude of each negative number above by flipping each bit, then adding 1 to 1's complement, as shown below:

$$\mathbf{(00000010)_b\ (1\text{'s complement by flipping each bit}) \rightarrow (11111101)_b}$$
$$\mathbf{(then + 1) \rightarrow (11111110)_b}$$

$$\mathbf{(00000011)_b\ (1\text{'s complement by flipping each bit}) \rightarrow (11111100)_b}$$
$$\mathbf{(then + 1) \rightarrow (11111101)_b}$$

https://doi.org/10.15215/remix/9781998944088.01

The addition above will become

$$\mathbf{(11111110)_b + (11111101)_b}$$

which is equal to

$$\mathbf{(11111011)_b}$$

The 1 on the highest bit of the result means that the result is 2's complement representation of a negative number. Interestingly, the magnitude of the negative number is the complement of $(11111011)_b$, which is

$$\mathbf{(11111011)_b\ (1\text{'s complement by flipping each bit}) \rightarrow (00000100)_b}$$
$$\mathbf{(then + 1) \rightarrow (000000101)_b,}$$

which is 5, meaning that the result of −2−3 is −5.

It is very easy to calculate the 2's complement of a binary number. Adding a negative number can be done by adding its 2's complement. That is the advantage that the binary system has offered for the construction and development of modern digital computers.

As such, in principle, the key component needed in our modern digital computers for computing and information processing is a unit to add two binary numbers, since the three other basic arithmetic operations can be done based on addition, using respective algorithms or routines such as the ones mentioned above. In the real implementation and construction of digital computers, all basic arithmetic operations, including bitwise operations on integer binary numbers such as the operations needed to find 2's complements, are hardwired into a unit called the arithmetic logic unit (ALU), which is the core of the central processing unit (CPU) that you have heard about often.

A digital computer also needs to compute real or floating-point numbers as well as process graphics. To get these two jobs done more efficiently, modern computers also have a floating-point unit (FPU) for floating-point numbers, and a graphics processing unit (GPU) for graphics.

MECHANIC-BASED COMPONENTS

Mechanical components such as levels, gears, and wheels can be used to build analog computers. In fact, digital computers can also be built with mechanical components. The earliest special-purpose mechanic-based digital computer was the difference engine designed by Charles Babbage. A general-purpose mechanic-based digital computer, called the analytical engine, was also first proposed by Charles Babbage. In the design, the machine has memory and CPU. It would have been programmable and, therefore, would have become a general-purpose digital computer, if ever actually built.

https://doi.org/10.15215/remix/9781998944088.01

VACUUM TUBE–BASED COMPONENTS

Invented in 1904 by John Ambrose Fleming of England, a vacuum tube is a tube made of glass with gas/air removed, with at least two electrodes inserted for allowing or controlling the flow of electrons. Vacuum tubes can be made to be switches or amplifiers. As switches, the two states of on and off can be used to represent the 0 and 1 of binary arithmetic or Boolean logic. When the state of output can be controlled by input or inputs, the switch is made to be a gate to perform different logical operations. That's how vacuum tubes could be used to implement modern computers; as amplifiers, they found their uses in many electronic devices and equipment such as radio, radar, television, and telephone systems. It was the vacuum tube that made electronic computers and other electronic devices possible for the first time. Vacuum tubes played very important roles in the development and construction of all electronic devices and systems in the first half of the 20th century.

However, vacuum tubes disappeared from computers and most other electronic devices soon after transistors were developed at the Bell Laboratories by John Bardeen, Walter Brattain, and William Shockley in 1947 because, compared to transistors, vacuum tubes were too big, too heavy, too unreliable and consumed too much power to operate. That limited the construction and uses of vacuum tube–based computers and devices.

TRANSISTORS

Recall that in the construction of computers, the ALU or CPU requires two-state switches to represent 0 and 1 and logical gates to perform additions and subtractions of binary number systems. Vacuum tubes could be made to construct electronic computers, but the computers became too bulky and too heavy to be more useful.

INTEGRATED CIRCUITS AND VERY LARGE-SCALE INTEGRATED CIRCUITS

Integrated circuits (ICs) are chips with many electronic circuits built in, while very large-scale integrated (VLSI) circuits are those with millions and even billions of electronic circuits built into very small semiconductor chips. According to Moore's law (attributed to Gordon Moore, the cofounder and former CEO of Intel), the number of transistors in a dense integrated circuit doubles about every two years. This has been the case over the last few decades since 1975, though advancement in VLSI technology has slowed since 2010.

Today, computers are all using VLSI chips. That is why computers have become smaller and smaller, while computing power has increased exponentially in accordance with Moore's law.

https://doi.org/10.15215/remix/9781998944088.01

1.3 Programming and Programming Languages

One of the key principles of modern computers is using stored programs, which are sequences of instructions telling computers what to do. The task of writing programs for computers is called programming.

During the first generation of computers, machine languages and assembly languages were used to write programs. Machine languages use sequences of 0 and 1 to code instructions for computers. Programs in a machine language can be directly understood by the CPU of the target computer. The following is an example of machine language for the Intel 8088 CPU:

```
00000011 00000100
10001000 11000011
```

As you can see, although machine language is friendly to computers, it is not friendly to people, because the instruction itself does not tell us what it does in a human-readable language. To solve this problem, assembly language was invented. The two instructions above are written in assembly language as follows:

```
ADD AX, [SI]
MOV BL, AL
```

The instruction in assembly language is much more friendly to people because one can more easily tell what each instruction does. However, assembly language is not as friendly to computers. For computers to understand programs in assembly language, an assembler is needed to translate the programs into code in machine language.

With assembly language, programming is still a difficult task because instructions in an assembly language are mostly direct mapping of their machine-language equivalent and only describe fine and tiny steps of CPU operations. It is difficult to program in assembly language to solve real-world problems.

The idea of using a high-level, more human-friendly programming language was attributed to Konrad Zuse for his work between 1942 and 1945. He developed a high-level programming language called Plankalkül for a non–Von Neumann computer called Z4 that he made during the same period of time. The language was not implemented and used on Z4 or any other real computer during that time, but Konrad Zuse did write a computer chess program in the language.

https://doi.org/10.15215/remix/9781998944088.01

The first high-level programming language for our modern computers (Von Neumann machines) is ALGOL 58, but it was soon surpassed by ALGOL 60, a structured programming language.

During the second generation of modern computers, more than 100 high-level programming languages were developed. The most popular programming languages include COBOL, for programming business systems; Fortran (an imperative procedural programming language), for scientific calculation; ADA, used by the military and National Defense in the US; Lisp (a functional programming language), used for symbolic or information processing; and Prolog (a logic programming language), for logic programming, especially during the first revival of artificial intelligence in the 1980s. There have been thousands of programming languages since the invention of the modern computer, according to the list at https://en.wikipedia.org/wiki/List_of_programming_languages.

However, most of these programming languages have never been used in real application development. The most popular programming languages used today include Python, Kotlin (for Android applications), Java, R (for data analysis and machine learning), JavaScript, C++, C#, PHP, Swift (for iOS applications), Go, and C, which has been a popular language for more than half of the century and is considered a foundational language of many other popular programming languages.

High-level programming languages are friendly to people but not friendly to computers. For computers to understand and execute programs written in a high-level programming language, they must be translated into machine language. Based on how programs in high-level languages are translated into target machine language, high-level programming languages are either compiled or interpreted. A program written in a compiled programming language such as C or C++ needs to be compiled into its target machine's codes and linked with copies of code in static libraries, or linked with references to code in dynamic or shared libraries, so that the resulting object can then be executed by the target computer.

Programs written in an interpreted programming language can be executed directly by the interpreter of the programming language, in view of programmers and users. Behind the scenes, however, to speed up the execution of programs written in an interpreted programming language, the programs are often translated first into some intermediate codes, often called bytecodes. The bytecodes are then executed within a so-called virtual machine built into the interpreter of the programming language.

More readings about the history of computer programming languages can be found on the internet by searching for "history of computer programming." The following are some of the articles:

https://doi.org/10.15215/remix/9781998944088.01

https://en.wikipedia.org/wiki/History_of_programming_languages
http://www.softschools.com/inventions/history/computer_programming_history/369/
https://www.datarecoverylabs.com/company/resources/history-computer-programming-languages
https://www.computerhistory.org/timeline/software-languages/
https://www.thecoderschool.com/blog/the-history-of-coding-and-computer-programming/

A good overview of programming languages can be found at https://en.wikipedia.org/wiki/Programming_language.

It is worth noting that although well-formed computer programming languages only came into existence in the late 1940s, Ada Lovelace, an English mathematician who lived between 1815 and 1852, is considered the first computer programmer. When she was commissioned to translate an article based on Charles Babbage's speech about his analytical engine, she wrote extensive notes alongside the translation to explain the principles and mechanism of the analytical engine. Those notes were later considered to be the first computer programs in history. For example, one set of those notes would calculate a sequence of Bernoulli numbers if executed on the analytical engine.

1.4 Python Programming Language

Python is an interpreted, high-level, general-purpose programming language. It is one of the most popular programming languages in the world, second only to Java. The 2023 rankings for programming language popularity can be found at https://www.hackerrank.com/blog/most-popular-languages-2023/.

The Development and Implementation of Python

Python was originally conceived and developed in the Netherlands by Guido van Rossum in the late 1980s. Its first release was Python 0.9.0 in 1991. Python 2.0 was released in 2000, nine years after the release of Python 0.9.0.

Most of the development of Python was accomplished between 2005 and 2013, when Guido van Rossum was working at Google, where he spent half of his time on the development of the programming language. The latest Python 2 release is Python 2.7.16, whereas the latest release of Python 3 is Python 3.9.0, at the time of writing.

Python is an interpreted programming language rather than a compiled programming language. Programs written in Python don't need to be compiled into target machine code. Instead, they only need to be translated into

https://doi.org/10.15215/remix/9781998944088.01

bytecodes, then executed by a Python Virtual Machine (PVM), which is built into the Python interpreter.

Python language has different implementations with different names. CPython, written in C and known simply as Python, is what we will be using, and it is the implementation you will get from Python standard distribution at python.org by default. In addition to CPython, the following are some alternative implementations of Python:

1. JPython or Jython, and implementation for running on Java Virtual Machine (JVM)
2. IronPython, an implementation for running on .NET
3. PyPy, an implementation for working with a just-in-time (JIT) compiler
4. Stackless Python, an implementation of a branch of CPython supporting microthreads
5. MicroPython, an implementation for running on microcontrollers

These implementations may be of interest only for some special applications.

Advantages of Python

Python was designed with the principle of "programming for everyone." Programs written in Python are clean and easy to read. Yet Python is a general-purpose programming language and very powerful to code in. Programming in Python has the least overhead, especially when compared to Java. Let's take the famous "Hello World!" application as an example. In Java, to create this application, you need to write a class with a main method like the following:

```
class first {
    public static void main(String args[]){
    System.out.println("Hello World!");
    }
}
```

Then save it to a file and compile the file into Java bytecode by running the following command, provided you have a JDK correctly installed:

```
javac myhelloworld.java
```

This command will generate a file named first.class. You can then run command java first to have "Hello World!" spoken out.

https://doi.org/10.15215/remix/9781998944088.01

In Python, however, you only need to write the following code and save to a file, say, myhelloworld.py:

```
print("Hello World!")
```

Then run the following command:

```
python myhelloworld.py
```

More importantly, Python supports different programming paradigms, including structured programming, imperative programming, object-oriented programming, functional programming, and procedural programming. Logic programming is probably the only exception; it is not supported by Python.

Python is very powerful for two reasons. The first reason, as you will see later, is that Python has many powerful language constructs and statements to represent data and operations on various data. The second reason is that there are thousands of third-party packages/libraries/modules available for all kinds of programming needs in addition to the large standard libraries included with every release of Python. For example, Python is widely used for data analytics and machine learning with the support of the NumPy, SciPy, Pandas, Matplotlib, and TensorFlow libraries.

Resources for Python and Python Education

The ultimate resource for Python and API libraries is the official website of the Python software foundation at https://www.python.org/. From that website, you will be able to access all the releases of Python and other packages and libraries that you may be interested in, in addition to the standard libraries that come with the official release of Python.

At the time of writing, the latest release of Python is 3.11.1, but you can always check and locate the latest release for your platform at https://www.python.org/downloads/. However, for your convenience and for your future study and use of Python, we will use a package management system called Anaconda to install and manage Python and all tools and libraries that you will need for this textbook, as detailed later.

There are plenty of resources on the internet for eager learners to learn almost anything, including programming in Python. To learn effectively on the internet, however, a learner must be able to choose the right materials from thousands—even millions—of choices, and that has proved to be a very difficult task. That's probably one of the reasons why universities and professors are

https://doi.org/10.15215/remix/9781998944088.01

still needed: to set the right learning paths and to select the right resources for learners.

On the internet, the most authoritative learning source about the Python language is the documentation (https://docs.python.org/3/), including language references (https://docs.python.org/3/reference/index.html) and library references (https://docs.python.org/3/library/index.html), although they may not be articulated for beginners.

For learners who prefer to learn by reading, the following are some of the best resources on the web:

- https://docs.python.org/3/: This resource is from the official Python website, so the terms and jargons used should be deemed official at least within the Python community. It is a comprehensive section as well.
- https://www.w3schools.com/python/: This is a very popular site that offers resources covering many different languages.
- https://www.learnpython.org/: This is another good site for learning Python. it has many resources on specific topics.

For learners who love to watch videos and lectures, the following are recommended:

- Python Crash Course for Beginners, at https://www.youtube.com/watch?v=JJmcL1N2KQs, by Traversy Media. This video uses VS Code IDE to illustrate the program examples so that you can follow along once you have the required programming environment set up.
- Learn Python—Full Course for Beginners, at https://www.youtube.com/watch?v=rfscVS0vtbw, by freeCodeCamp.org and narrated by Mike Dane. This video is one of the best. Please note that the video lecture recommends the installation and use of PyCharm IDE. However, the installation and use of VS Code IDE is required in this book.

1.5 Getting Ready to Learn Programming in Python

To learn the content covered in the rest of this book, you first need to have the following installed on your computer:

1. A newer version of Python from https://www.python.org/. The latest version, at the time of writing, is Python 3.11.1. The standard

https://doi.org/10.15215/remix/9781998944088.01

distribution includes everything you need to interpret and execute Python programs, which are also called Python scripts.
2. A newer version of Jupyter Notebook, from https://jupyter.org/. Jupyter Notebook is a powerful interactive programming environment supporting up to 40 programming languages, including Python.
3. A newer version of Visual Studio Code (VS Code) from https://code.visualstudio.com/. VS Code is a free and open-source code editor developed by Microsoft, and it will be used as an integrated development environment (IDE) and to host Jupyter Notebook for interactive programming.

In this section, we will learn how to install Python, Jupyter Notebook, and Visual Studio Code IDE on our computer. A standard distribution of Python includes a large volume of libraries for general programming needs. Special applications such as those in data science and machine learning will require special libraries to be installed. We will learn about that in later units when we need them.

Installing and Setting Up the Python Programming Environment

There are different ways of setting up your Python programming environment. The method we present below is believed to be more reliable and less subject to future changes.

INSTALLING PYTHON

The first package we need to install is Python. The steps are as follows:

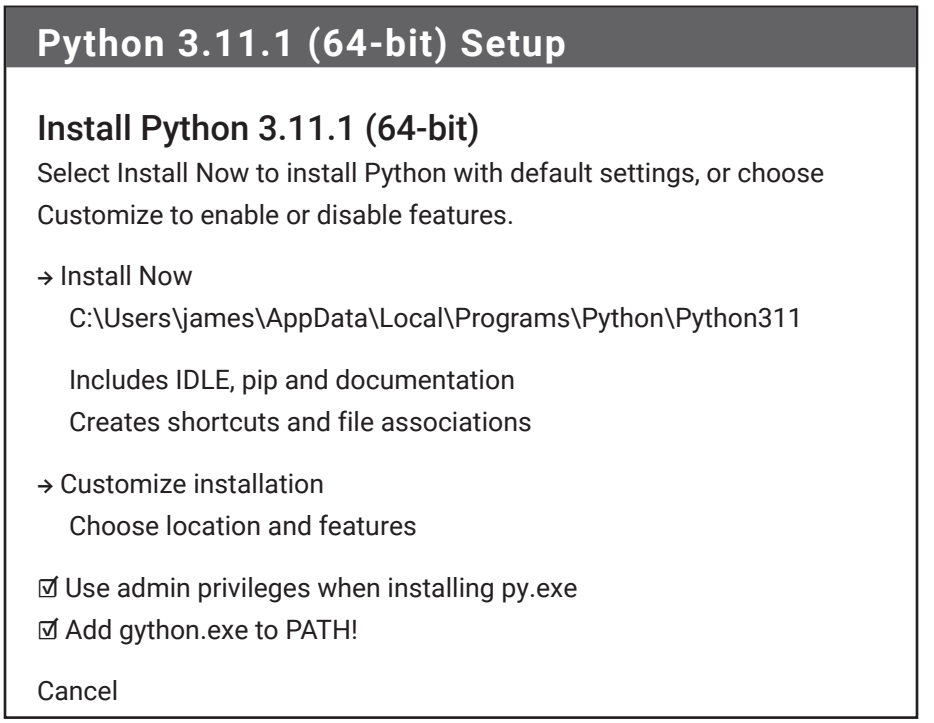
1. To ensure a clean installation of Python, uninstall all old versions of Python that may have previously been installed on your computer and check the value of the PATH environment variable to delete all those, and only those, related to Python. To ensure that all have been deleted properly, open a shell terminal such as CMD or PowerShell on Windows and check if a Python installation still exists by trying to run it. You should be told that Python is not recognized.
2. Go to https://www.python.org/downloads/ and look for the newest release of Python for your platform. For example, if you want to install Python on your Windows machine, download the latest release for Windows. Normally, the website should be able to detect the type of system you are using to access the website and show you the right

https://doi.org/10.15215/remix/9781998944088.01

Python package for you to download. You should, however, be aware of what you will be getting from the website.

3. Run the installer to install it. If installing Python on a Windows platform, all you need to do to start the installation is to double-click the downloaded file or press the run button if it does not start the installation automatically.

   After the installation has started, a window will pop up, as shown here:



   You can either let it install Python automatically for you by clicking "Install Now" or choose "Customize installation." When you choose "Customize installation," you have the opportunity to choose where Python will be installed on your computer and what optional modules and features will be installed.

   In either case, you must check the box to add Python to the PATH environment variable. Choose yes to add Python to the system PATH environment variable so that Python and other related commands can be found.

   Do a customized installation so that you will see what is to be installed and where. When doing customized installation, remember to check

https://doi.org/10.15215/remix/9781998944088.01

the box to install pip, Python package manager, and other components within the distribution, as shown here:

Python 3.11.1 (64-bit) Setup

Optional Features

☑ Documentation
Installs the Python documentation files.

☑ pip
Installs pip, which can download and install other Python packages.

☑ td/tk and IDLE
Installs tkinter and the IDLE development environment.

☑ Python test suite
Installs the standard library test suite.

☑ py launcher ☑ for all users (requires admin privileges)
Use Programs and Features to remove the 'py' launcher.

Back Next Cancel

Check the following advanced options, as shown here:

Python 3.11.1 (64-bit) Setup

Advanced Options

❑ Install Python 3.11 for all users
☑ Associate files with Python (requires the 'py' launcher)
☑ Create shortcuts for installed applications
☑ Add Python to environment variables
❑ Precompile standard library
❑ Download debugging symbols
❑ Download debug binaries (requires VS 2017 or later)

Customize install location
d:\Programs\Python\Python311
You will require write permissions for the selected location.

Back Install Cancel

https://doi.org/10.15215/remix/9781998944088.01

To check whether Python has been installed successfully on your computer, open a shell terminal and run the following command:

```
$ python --version
```

If the installation was successful, the version of Python will be displayed and should be the same as the one you have downloaded from python.org.

SETTING UP A VIRTUAL ENVIRONMENT FOR A PYTHON PROJECT

When programming in Python, you will need a Python interpreter and any additional libraries required for the project. Libraries required for one project are different from those for other projects. To ensure that each project has the right programming environment without interfering with others, Python uses a technology called a virtual (programming) environment, which can be created and activated for each project. When you work in a virtual environment for a specific project, all installations of additional libraries will be used only for that specific project.

To create and use a virtual environment, take the following steps:

1. Check whether you have a tool called pipenv installed with your Python installation by running the following command on Windows:

```
where.exe pipenv
```

or the following command on Linux or Mac:

```
which pipenv
```

If it is installed, the command will tell you where it is located on your computer, and you can go to step 3.

2. If the command cannot be located on your system, you will need to install it by running the following command on a shell terminal:

```
pip install pipenv
```

If you are running the shell terminal as a regular user, pipenv will be installed under your home directory, so that you will have to add the location to the environment variable PATH. It is better to start a shell

https://doi.org/10.15215/remix/9781998944088.01

terminal as an administrator, then run the above command as root/administrator to ensure that pipenv will be installed globally and be accessible for all users with the already set value for PATH.

3. Once you are sure that pipenv is installed, create a directory for your new project and change the work directory to that directory, say c:\dev\testproject as an example, by running the following commands in sequence:

```
c:\dev> mkdir testproject
c:\dev> cd testproject
```

4. Within the testproject directory, run the following command:

```
c:\dev\testproject> pipenv install
```

This will create a virtual environment for your project rooted at c:\de\firstproject.

5. To work on the project with the virtual programming environment, you need to activate the virtual environment for the project by running the following command within the project directory:

```
c:\dev\testproject> pipenv shell
```

Once the virtual environment has been activated, the prompt of the shell terminal will become something similar to the following:

```
(testproject-UCP5-sdH)c:\dev\testproject>
```

Please note the text within the parentheses. It contains the name of the project directory. It means that you are now working in a subshell terminal invoked by pipenv.

6. From now on, any package installed by running the pip command on this subshell terminal will be only part of this virtual environment, without interfering with installations elsewhere for other projects.
7. To get out of the virtual environment, simply type “exit” to close the subshell, as shown below:

```
(testproject-UCP5-sdH)c:\dev\testproject>exit
```

https://doi.org/10.15215/remix/9781998944088.01

The prompt will become

```
c:\dev\testproject>
```

8. If you want to remove the virtual environment created for a project, run the following command within the subshell on the root of the project directory:

```
(testproject-UCP5-sdH)c:\dev\testproject> pipenv
-- rm
```

INSTALLING JUPYTER NOTEBOOK

The second package that needs to be installed is Jupyter Notebook. The steps are as follows:

1. After you have successfully installed Python, install Jupyter Notebook with pip, the Python package manager that comes with the Python installation, by running the following command at a shell terminal:

```
pip install Jupyter
```

or use the following command if pip itself is not recognized as a command, but only installed as a module of Python:

```
python -m pip install jupyter
```

To see if it has been successfully installed, run the following command from a shell terminal:

```
jupyter-notebook
```

The command will start a web service on your computer, then launch Jupyter Notebook Service within your default web browser.

Start a new notebook by clicking "New" and "Choose Python 3." You can then start the program interactively within the notebook, as shown in Figure 1-3.

https://doi.org/10.15215/remix/9781998944088.01

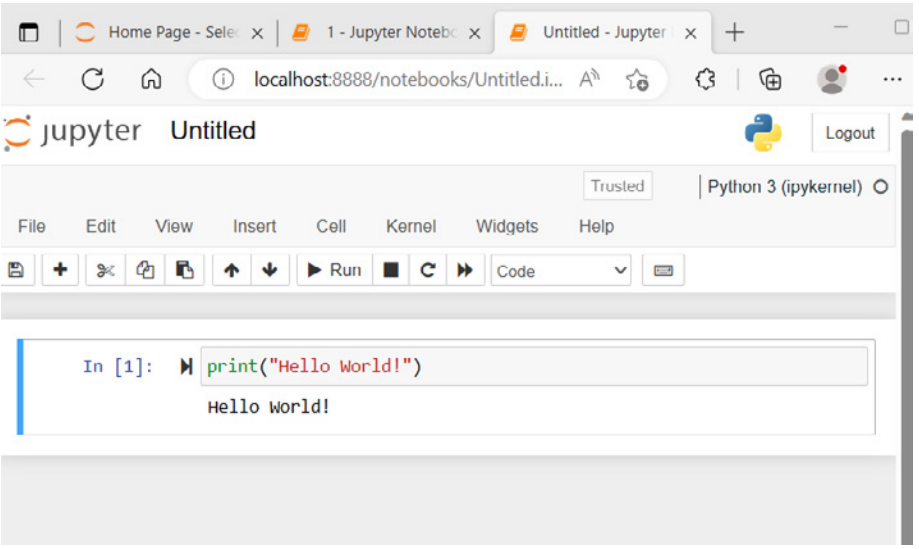


Figure 1-3: Jupyter Notebook in browser

INSTALLING VISUAL STUDIO CODE

The last package to be installed is Visual Studio Code, a free and open-source IDE developed by Microsoft that can be used for Python programming. The steps are as follows:

1. Go to https://code.visualstudio.com/, click "Download," and download the installer for your platform.

   If you want to see other download options, such as a version for a platform other than the one identified by the VS Code website, you can scroll down further to check.
2. Double-click "Downloaded Installer" for your platform to run and install Visual Studio Code.

To run Visual Studio Code on Windows, click the Start menu, scroll down to find Visual Studio Code, and click.

Additional Tools Supporting Software Development in Python

To make software development more efficient, the Python community has made the following frameworks or systems available for Python developers.

BUILDBOT

Buildbot is a framework intended to automate all aspects of software development. It supports parallel and distributed job execution across multiple

https://doi.org/10.15215/remix/9781998944088.01

platforms, with version control and job status reporting and automated job scheduling when required resources become available. It is hosted at https://buildbot.net/. An introductory section can be found at http://docs.buildbot.net/current/tutorial/firstrun.html.

TRAC

Compared to Buildbot, Trac is a simpler web-based software project management system that can track issues and bugs. It requires access to a database such as SQLite or MySQL. Trac is hosted at https://trac.edgewall.org/, but the latest release can be found at https://pypi.org/project/Trac/.

ROUNDUP

Compared to Trac, Roundup is an even simpler tool to track issues in a software development project. You can do so via the command line, the web, or email. The system can be found at https://pypi.org/project/roundup/.

1.6 Getting a Taste of Programming with Python

In this section, you will see how to solve simple problems or do simple tasks with Python. At this time, you do not need to understand every piece of code because you will learn all those details later.

Program Interactively with Python Interactive Shell

Programming for computers is writing instructions for computers to execute. Interactive programming is programming in an environment in which you can interact with a computer, the interpreter of the programming language, more precisely, instruction by instruction. The opposite of interactive programming is batch programming, in which you will need to write a complete program and then feed the entire program to a language engine such as an interpreter, language runtime, or virtual machine in order to execute it. In interactive programming, a piece can be a single statement or a group of statements, but not a complete program. One of the advantages of interactive programming is immediate feedback on the code pieces from the interpreter.

The simplest interactive programming environment for Python is the Python interactive shell. It can be quickly started by running command Python at a command prompt such as within a Windows PowerShell, as shown here:

```
PS S:\Dev> python
```

https://doi.org/10.15215/remix/9781998944088.01

The started interactive Python programming environment should look like this:

```
PS S:\Dev> python
Python 3.11.1 (tags/v3.11.1:a7a450f, Dec 6 2022,
19:50:39) [M.06 v.1934 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more
information.
>>>
```

Here, >>> is the prompt of the Python interactive shell, waiting for your input, which will then be evaluated by the interpreter. This is how you program interactively within this environment.

As the de facto standard, our first program in the interactive Python programming environment is to say "Hello World!" as shown here:

```
PS S:\Dev> python
Python 3.11.1 (tags/v3.11.1:a7a450f, Dec 6 2022,
19:58:39) [MSC v.1934 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more
information.
>>> print("Hello World!")
Hello World!
>>>
```

As can be seen above, this program in Python needs only one statement made of one line of code, whereas in C/C++ or Java, it would need a dozen lines of code.

Please note that in the previous example, the characters behind the Python prompt >>> are Python statements you need to type, and the rest are output from Python interpreter or Python Virtual Machine (PVM).

Our next sample program within the Python interactive shell is to assign 8 to variable x and assign 9 to variable y, then print out the sum of x and y, as shown here:

```
PS S:\Dev> python
Python 3.11.1 (tags/v3.11.1:a7a450f, Dec 6 2022,
19:58:39) [MSC v.1934 64 bit (AMD64)] on win32
Type .help., "copyright", "credits" or "license" for more
information.
```

https://doi.org/10.15215/remix/9781998944088.01

```
>>> print("Hello World!")
Hello World!
>>> x = 8 # assign 8 to variable x
>>> y = 9 # assign 9 to variable y
>>> print(f'{x} + {y} = {x+y}')
8 + 9 = 17
>>>
```

Within this interactive programming environment, you can type any Python expression directly behind >>>, and it will then be evaluated. An example is shown here:

```
PS S:\Dev> python
Python 3.11.1 (tags/v3.11.1:a7a450f, Dec 6 2022,
19:58:39) [MSC v.1934 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more
information.
>>> 1350-789*3/26
1258.9615384615386
>>>
```

You may have realized that the interactive Python programming environment can be used as a powerful scientific or financial calculator, with all the functions you need, as long as you know where to find the functions and how to import and use them to write formulas.

Throughout the textbook, we will occasionally give examples in Python interactive shell when it is more convenient and proper in context. However, our preferred interactive programming environment is Jupyter Notebook within VS Code IDE, as you will see later in this section.

Program with VS Code IDE

Interactive programming environments like Python Shell and Jupyter Notebook are good for testing Python statements, code blocks, and some real programming tasks such as data analytics, where interactive programming is more suitable. However, writing programs that contain thousands of lines of code in a Python interactive shell or Jupyter Notebook is inconvenient.

As previously mentioned, we will be using VS Code as our IDE. Our first small project is to write a program that takes an integer from the user input

https://doi.org/10.15215/remix/9781998944088.01

and tells whether it is a prime number or not. The first step to do this is to start VS Code from the Windows Start menu or desktop. In Windows or iOS, you may type "VS Code" in the search bar to launch the application.

To create a Python program file within VS Code, choose "New File" from File, and then save the file as xyz.py, where xyz is your preferred name for the small project. Here, we use "primetest.py." VS Code will ask you to choose a folder for the program, and you may create one and then select it to open after the file is created. The program in VS Code will look like the one in Figure 1-4.

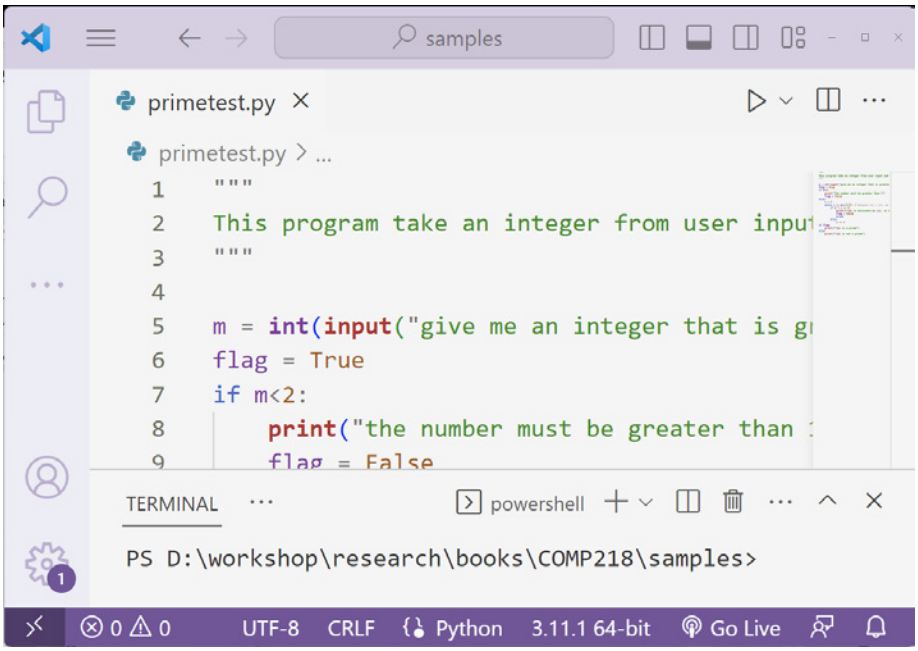


Figure 1-4: Python program in VS Code

At this time, you are not required to fully understand the program, but if you are interested, you may type the code line by line into your VS Code and run it by clicking the play button at the top-right corner of the VS Code window, just to get a taste of programming in VS Code with Python.

Please note that if you have multiple Python program files open in VS Code and want to run a particular one within the IDE, you will need to click the file to make it active; then, within the editing area, click the right button of the mouse to pop up a menu and select run for your particular program. This is even more important if you have multiple editing tabs open for different programs.

https://doi.org/10.15215/remix/9781998944088.01

Use Jupyter Notebook Within VS Code to Program Interactively

You may recall that Jupyter Notebook can be launched and run in a web browser, but for the purposes of this text, Jupyter Notebook will be run and used within VS Code to program interactively to allow you to benefit from many of the features that VS Code has.

Because Jupyter Notebook is natively supported within Visual Studio Code (as long as Jupyter Notebook is installed in your Python environment), all you need to do to get Jupyter Notebook running within VS Code is to open or create a notebook file, with extension ipynb in VS Code.

There are two ways to create a Jupyter Notebook file. One is to press Ctrl+Shift+P to search for the command "create new blank Jupyter Notebook," then click the command. A new blank Jupyter Notebook will be created within VS Code.

Alternatively, we can also simply create a new text file, and then save the file as xyz.ipynb, where xyz is a name you prefer and ipynb is the file's extension. Please note that to ensure the file type is ipynb, you have to select Jupyter (.ipynb) or all files (*.*) from the "Save as Type" list.

In either case, you may be asked to install an extension for Jupyter Notebook support. Click yes when you are asked to. You may also be asked to install Python if you have not done so yet or if the installation path has not been added to the environment variable. Click yes too to make sure you have Python properly installed on your computer.

Once a new Jupyter Notebook is created, you can then use it to program interactively within VS Code and enjoy many of the smart features of VS Code, such as those offered by IntelliCode.

Another advantage of using Jupyter Notebook within VS Code is that you can export a notebook as an HTML, PDF, or Python Script file. If you use Markdown cells to document your coding in code cells, you can produce a very nice document as a PDF or in HTML.

Later on, you will be required to create a Jupyter Notebook for each chapter or section, if there are many coding examples to work on in a section. You will use the Jupyter Notebook to interactively program all sample codes in that chapter or section or to test your own code to reinforce your learning. For chapter x, the Jupyter Notebook should be named chapter-x.ipynb, and for section x.y, the notebook should be named section-x.y.ipynb.

In the next section, we will provide a brief introduction to Markdown language that you can use to document your work either within Markdown cells of Jupyter Notebook or within a Markdown file, a file with md as the file extension.

https://doi.org/10.15215/remix/9781998944088.01

Write Documentation in Markdown

Markdown is a simple and straightforward markup language with a plaintext-formatting syntax. It was created by John Gruber and Aaron Swartz in 2004. With Markdown, you can write a document in an easy-to-write and easy-to-read format, then convert it to HTML. Within VS Code IDE, a Markdown file, with md as the extension, can be converted to PDF as well as HTML. Moreover, documentation written in Markdown cells of a Jupyter Notebook can be automatically converted into docstrings when the Jupyter Notebook is exported into a Python script file, and when the notebook is exported into PDF, scripts in code cells will be nicely embedded into documentation written in Markdown cells. Together with markup syntax for program code, Markdown cells within Jupyter Notebook running in VS Code provide a much more user-friendly way to include program code in rich and formatted documents.

The Markdown language has been extended since its initial release by John Gruber. You will be introduced to both the basic syntax and some extended syntax (but only those supported by Jupyter Notebook within VS Code) to serve our purposes in this book.

HEADINGS

In Markdown documents, headings are simply led by one or more hash symbols. As we have seen in previous sections, a Level 1 heading is led by a single hash symbol, a Level 2 heading is led by two hash symbols, and a Level 3 heading is led by three hash symbols. You can have up to six levels of headings in a document, as shown below:

```
# This is a Level 1 heading
```

This is a Level 1 heading

```
## This is a Level 2 heading
```

This is a Level 2 heading

```
### This is a Level 3 heading
```

This is a Level 3 heading

```
#### This is a Level 4 heading
```

This is a Level 4 heading

```
##### This is a Level 5 heading
```

This is a Level 5 heading

https://doi.org/10.15215/remix/9781998944088.01

```
###### This is a Level 6 heading
```

This is a Level 6 heading

Please note the space between the formatting symbol or symbols and the text to be formatted. A single space should be placed between the formatting symbol and the text being formatted in Markdown.

PARAGRAPHS

In Markdown, paragraphs are separated with one or more blank lines.

```
This is a paragraph.

This is another paragraph.
```

This is a paragraph.

This is another paragraph.

NEW LINES

In Markdown, to break a line like you would with <br/> in HTML, use more than one single space to break the line.

```
This line will break.  This line starts on a new line.
```

This line will break.
This line starts on a new line.

ITALIC, BOLD, AND STRIKETHROUGH TEXTS

To format text to be italic in Markdown, simply lead it with a * or underscore _ and use another * or _ to indicate the end of the text; to make text bold, use ** or __; to make text both bold and italic, use ***; to have a line strikethrough the text, use ~~. The following are examples:

```
_ Italic_
```

Italic

```
** Bold **
```

Bold

https://doi.org/10.15215/remix/9781998944088.01

```
*** both italic and bold ***
```

Italic and bold

```
~~ strikethrough ~~
```

~~Strikethrough~~

HORIZONTAL RULES

To add a horizontal line within a document, like <hr /> in HTML, use three hyphens ---. Most of the time, this has the same effect in Word.

```
---
```

KEYBOARD KEYS

In computing documentation, we often need to explain what key is used on the keyboard. To represent a key on the keyboard, we use HTML kbd tags directly, as shown below:

```
<kbd> Ctrl </kbd> <kbd> A </kbd>
```

Ctrl+A

```
<kbd> Ctrl+Shift+F3 </kbd>
```

Ctrl+Shift+F3

UNORDERED LISTS

With Markdown, writing an unordered list is rather straightforward, as shown below:

```
* first list item
* second list item
 * first item of sublist
 * second item of sublist
* third list item
```

The rendered result will be the following:

https://doi.org/10.15215/remix/9781998944088.01

- first list item
- second list item
  - first item of sublist
  - second item of sublist
- third list item

We can also use - in place of *.

ORDERED LISTS

To write an ordered list in Markdown is straightforward too, as shown below:

```
1. first list item
2. second list item
 * first item of sublist
 * second item of sublist
3. third list item
```

The rendered result will be

1. first list item
2. second list item
   - first item of sublist
   - second item of sublist
3. third list item

DEFINITION LISTS

The simple Markdown syntax for a definition list does not work in Jupyter Notebook within VS Code. However, we can use HTML <dl> tags directly to make such a list, as shown below:

```
<dl>
<dt> Python </dt>
<dd> It is a popular programming language, widely used in
AI and Data Science. </dd>
<dt> AI </dt>
<dd> Short for Artificial Intelligence.
It is the study of how to design and develop smart
artifacts.</dd>
</dl>
```

https://doi.org/10.15215/remix/9781998944088.01

The rendered result will be:

Python
: It is a popular programming language, widely used in AI and Data Science.

AI
: Short for Artificial Intelligence. It is the study of how to design and develop smart artifacts.

LINKS

To add a link with Markdown, put the anchor name in a square bracket, and put the URL in a pair of parentheses, as shown below:

```
[Markdown Home](https://www.markdownguide.org/)
```

The rendered result will be the following text, which will take the user to https://www.markdownguide.org/ when clicked:

Markdown Home

As in Word and some other editors, legitimate URLs are usually automatically linked without any markup tags. In Markdown, if you don't want a URL to be automatically linked, you can enclose it with a pair of backticks, just treating it as program code, as shown below:

```
`https://www.markdownguide.org/`
```

LINKS TO INTERNAL SECTIONS

A specific ID can be added to each header. Such IDs can be used as internal anchors in a link, as shown below:

```
[Assignment 1] (#assignment_1)
```

Assignment 1

https://doi.org/10.15215/remix/9781998944088.01

IMAGES

To add an image to your documentation, use a syntax similar to that used for adding links, but with an exclamation mark at the front of the open square bracket, as shown below:

```
![Markdown logo](AU_and_50_logo.png)
```

The image will be rendered as

BLOCKQUOTES

To include a blockquote in your documentation, use an angle bracket at the start of each line, as shown in the following example:

```
> COVID-19 UPDATES
>
> EXAMS
>
> HELP & SUPPORT
>
> FACULTY OF BUSINESS STUDENTS
>
> FACULTY OF SCIENCE STUDENTS
```

> COVID-19 UPDATES
> EXAMS
> HELP & SUPPORT
> FACULTY OF BUSINESS STUDENTS
> FACULTY OF SCIENCE STUDENTS

https://doi.org/10.15215/remix/9781998944088.01

TABLES

To create a table in Markdown, use the pipe character | to divide columns and a sequence of dashes/hyphens to separate the header of a table, as shown below:

```
| Markdown symbol | Description | HTML equivalent  |

| :----------------| :-----------: | -----------: |

| # | Level 1 heading | h1 |

| ## | Level 2 heading | h2 |
```

The table will be rendered as

Markdown symbol	Description	HTML equivalent
#	Level 1 heading	h1
##	Level 2 heading	h2

Please note the colons used in the formatting syntax. A single colon to the left of dashes means to align all the text in the column to the left, a single colon to the right of dashes means to align all the text in the column to the right, and adding a colon to both sides means to align the text at centre. You can also use other Markdown syntax on the text in the table, such as italic, bold, and so on.

INLINE PROGRAM / SCRIPT CODE

When writing a report on a software project, you may need to include code samples in the report. To include a code sample within a single sentence, enclose the code within a pair of backticks `, as shown below:

```
The ` range(start, end, step) ` function is used to
produce a sequence of integer numbers.
```

The rendered result will be:

The range(start, end, step) function is used to produce a sequence of integer numbers.

https://doi.org/10.15215/remix/9781998944088.01

CODE BLOCK

In Markdown, a block of program code can be marked up using a pair of triple backticks ```, as shown below:

````
```Python
for i in range(1, 10):
for j in range(1, i + 1):
print(f'{i} * {j} = {i * k}')
```
````

The rendered result will be:

```
for i in range(1, 10):
  for j in range(1, i + 1):
    print(f'{i} * {j} = {i * k}')
```

Please note that the name of the programming language right behind the opening triple backticks is optional, though with the name, the code will be automatically highlighted in a way specific for the language.

MATHEMATICAL FORMULAS AND EXPRESSIONS

With Markdown in Jupyter Notebook, you can embed LaTeX representation mathematical formulas directly within your text. LaTeX is a typesetting system used for scientific publications. It would take you some time to learn the complete system. The following examples show you how to represent mathematical formulas in your documentation.

```
1.  $\hat{Y} = \hat{\beta}_{0} + \sum \limits _{j=1} ^{p}
X_{j}\hat{\beta}_{j} $
2.  $\frac{n!}{k!(n-k)!}$
3.  $\binom{n}{k}$
4.  $\frac{\frac{x}{1}}{x - y}$
5.  $\sqrt{k}$
6.  $\sqrt[n]{k}$
7.  $\sum_{i = 1}^{10} t_i$
8.  $\int_0^\infty \mathrm{e}^{-x},\mathrm{d}x$
9.  $f(x) = x^2 + 2, if\ x = 2$
10. $\oint_C x^3\, dx + 4y^2\, dy$
11. $2 = \left(
\frac{\left(3 - x\right) \times 2}{3 - x}
```

https://doi.org/10.15215/remix/9781998944088.01

```
\right)$
12. $\sum_{m = 1}^\infty\sum_{n = 1}^\infty\frac{m^2\, n}
{3^m\left(m\, 3^n + n\, 3^m\right)}$
13. $\phi_n(\kappa) =
\frac{1}{4\pi^2\kappa^2} \int_0^\infty
\frac{\sin(\kappa R)}{\kappa R}
\frac{\partial}{\partial R}
\left[R^2\frac{\partial D_n(R)}{\partial R}\right]\,dR$
```

The rendered result of the above Markdown code is shown in Figure 1-5. You can get the same result if you export the notebook file to PDF or HTML format.

1. $\hat{Y} = \hat{\beta}_0 + \sum_{j=1}^{p} X_j \hat{\beta}_j$
2. $\frac{n!}{k!(n-k)!}$
3. $\binom{n}{k}$
4. $\frac{\frac{x}{1}}{x-y}$
5. $\sqrt{k}$
6. $\sqrt[n]{k}$
7. $\sum_{i=1}^{10} t_i$
8. $\int_0^\infty \mathrm{e}^{-x}, \mathrm{d}x$
9. $f(x) = x^2 + 2, if\ x = 2$
10. $\oint_C x^3\, dx + 4y^2\, dy$
11. $2 = \left(\frac{(3-x)\times 2}{3-x}\right)$
12. $\sum_{m=1}^{\infty} \sum_{n=1}^{\infty} \frac{m^2\, n}{3^m (m\, 3^n + n\, 3^m)}$
13. $\phi_n(\kappa) = \frac{1}{4\pi^2\kappa^2} \int_0^\infty \frac{\sin(\kappa R)}{\kappa R} \frac{\partial}{\partial R} \left[R^2 \frac{\partial D_n(R)}{\partial R}\right]\, dR$

Figure 1-5: Rendered result of the Markdown code

https://doi.org/10.15215/remix/9781998944088.01

TO-DO LIST

You may wish to have a to-do list in your learning notebook. With Markdown, you can get that accomplished as follows:

```
- [ ] a bigger project
 - [x] first subtask
 - [x] follow-up subtask
 - [ ] final subtask
- [ ] a separate task
```

The rendered result will look like this:

- ☐ a bigger project
  - ☑ first subtask
  - ☑ follow-up subtask
  - ☐ final subtask
- ☐ a separate task

ESCAPE SEQUENCE FOR SPECIAL CHARACTERS

Because, as we have seen, some characters have special meaning in Markdown syntax, we need to use the backslash to allow these characters to keep their normal meaning. These characters include the backslash \, backtick `, asterisk *, underscore _, the pound hash symbol #, plus sign +, hyphen/dash -, period ., and exclamation mark !, as well as curly braces {}, square brackets [], and parentheses (). For example, if we want to have * in our documentation, we need to use * instead of a simple *. This is especially necessary since confusion may arise from using these symbols without the backslash. Otherwise, you can use a special character directly, such as in the following example:

```
# A heading with a plus sign \+
```

This will be rendered as:

A heading with a plus sign +

Programming Interactively with Jupyter Notebook Within VS Code

Earlier in this section, we learned how to program interactively within a Python Shell launched within a CMD or PowerShell window, and how to

https://doi.org/10.15215/remix/9781998944088.01

start Jupyter Notebook within VS Code. In this section, we will find out how to use Jupyter Notebook within VS Code to program interactively as well as how to document your work and learning journey in both code cells and Markdown cells of Jupyter Notebook within VS Code.

Compared to the Python interactive shell, Jupyter Notebook is a much better and more powerful environment for interactive programming and has the following advantages:

1. Everything you typed and the output from the Python interpreter are kept in a notebook file so that you can go back and review your work whenever needed. In the Python interactive shell, however, everything within the shell will be lost as soon as you exit from it.
2. Within a programming cell of Jupyter Notebook, you can write and edit as many Python statements as you want, and the interpreter will wait until you hit Shift+Enter to run all the statements within the active cell, whereas the simplest interactive programming environments run only one statement at a time.
3. You can go back to a previous programming cell and edit the code in it, and then rerun the code as you wish, which you cannot do in a Python Shell.

Again, our first example to program interactively in Jupyter Notebook is to say Hello World! But before we begin, we need to create a new Jupyter Notebook named section-1.6.ipynb for this section. If you are using this book for a course, we recommend that on your desktop, you create a folder using the course name or number, and then create this and all the Jupyter Notebook files within this folder or subfolders to better organize all the files for the course. For the purposes of this textbook, we will give this folder the name "comp218." In the example shown here, the notebook file section-1.6.ipynb is under a subfolder named "VS Code."

With Jupyter Notebook in VS Code, you can use Markdown cells to present your ideas and thoughts about the program you are to write, and use code cells to write program code and Python documentation on your code. In this first example, we first write the following in a Markdown cell:

```
# First program in Jupyter Notebook within VS Code
As is tradition in teaching computer programming, our
first program in Jupyter Notebook is to say Hello World!.
```

https://doi.org/10.15215/remix/9781998944088.01

```
## How to start a program
As always, a program should begin with a brief
description of the program, including what it does,
who wrote it and when, and how it works and/or should
be used. This is especially necessary for independent
program files such as the Python script files you will
be developing using VS Code IDE. Within a code cell of
Jupyter Notebook, brief documentation is still needed
if the code within the cell is complicated either
grammatically or logically. You do not need to document
if the code is only a scribble for testing.

In Python, brief documentation at the beginning of a
program file is enclosed with a pair of triple single/
double quotation marks, such as

"""
brief documentation
"""

or

'''
brief documentation
'''
This is called a docstring. Different from comments made
on program code using a single hash symbol #, docstrings
are meant to be formal documentation of the code that can
be retrieved from an object.
```

We then write the following in a code cell:

```
    """
    This simple program is just to say Hello world!
    Everything between the two triple quotation marks is
    treated as documentation about the program.
    """

    print('Hello world!')   # a single statement of the
program - inline comment
```

https://doi.org/10.15215/remix/9781998944088.01

This results in the notebook printing, outside of the code cell, the following:

```
Hello world!
```

In Jupyter Notebook, there are two types of cells for input. One is the code cell for you to write actual program code in. The other is the Markdown cell for you to write more detailed notes or reports about your work within the notebook using the Markdown language we have introduced in the previous section.

When a new cell is created in a Jupyter Notebook within VS Code, by clicking the plus sign + on the left side of the notebook window, you will get a code cell by default. To change a code cell to a Markdown cell, click the M↓ button at the top of the cell; to change a Markdown cell back to code cell, click the {} button at the top of the cell.

Please note that when using Jupyter Notebook within a web browser, to switch a code cell to Markdown cell you will need to click "Code" at the top of the cell, and then choose "Markdown" from the pop-up menu.

In the Jupyter Notebook example above, two cells are used. The first is a Markdown cell, in which we explain in more detail what we are going to do for our first Python program and how we will do it, whereas the second cell is a code cell, in which actual Python code is written, together with docstring and inline comments, for the program.

In the simple program above, a pair of triple quotation marks is used to enclose some string literals, called docstrings, as the formal documentation for the program or module. In addition to each program file or each module file, a docstring is also recommended, and even required, for each function, class, and method. Docstrings should be placed at the very beginning of a program file, module file, or right after the header of the definition of a class, function, or method. The docstrings will be retrieved and stored as the __doc__ attribute of that respective object and can be displayed when help is called on that object. Python also has a utility program called pydoc that can be used to generate documentation from Python modules by retrieving all the docstrings.

Right after the docstring is a print statement that will print out Hello World! when the program is executed by pressing Shift+Enter while the cell is still active. A cell is active if there is a vertical blue bar on the left side of the cell. To execute statements inside an active cell, we can also click the play button (the thick right-facing arrow) at the top of the cell.

Our next sample program is to assign integers to two variables and then print the sum, difference, product, quotient, integer quotient, remainder, power, and root. The Python statements to accomplish these operations are as follows:

https://doi.org/10.15215/remix/9781998944088.01

```
#Operators and expressions in Python
In the next cell we will show the use of operators and
expressions in Python.

#Operators
   + operator for addition
   - operator for subtraction
   * operator for multiplication
   / operator for division
   % operator for modulus
   ** operator for exponentiation
   // operator for floor division
```

We then write the following in a code cell:

```
i = 5 # assign 5 to variable i. Everything behind the
hash mark is comment
j = 3 # assign 3 to variable j
print(f"{i} + {j} = {i + j}") # print i + j
print(f"{i} - {j} = {i - j}") # print i - j
print(f"{i} x {j} = {i * j}") # print i * j
print(f"{i} / {j} = {i / j}") # print i / j (/ is division)
print(f"{i} // {j} = {i // j}") # print i // j (// is
quotient)
print(f"{i} % {j} = {i % j}") # print i % j (% is modulus)
print(f"{i} ** {j} = {i ** j}") # print the result of i
power of j
print(f"root {j} of {i} = {i ** (1/j)}") # print the
result of root j of i
```

This prints the following:

```
5 + 3 = 8
5 - 3 = 2
5 x 3 = 15
5 / 3 = 1.6666666666666667
5 // 3 = 1
5 % 3 = 2
5 ** 3 = 125
root 3 of 5 = 1.7099759466766968
```

https://doi.org/10.15215/remix/9781998944088.01

As you can see, in this interactive programming environment, you can write and edit many Python statements within a single cell. You may simply write some statements to accomplish certain calculation or data analysis tasks that cannot be done even on an advanced scientific finance calculator.

This next example in Jupyter Notebook within VS Code calculates the sum and the product of 1, 2, 3, ... 100:

```
"""
This is to calculate the sum of 1,2,…,100.
"""

s=0
for i in range(100):
    s+=i+1
print(f"Sum of 1,2…,100 is {s}")

"""
This is to calculate the product of 1,2,…,100
"""

p=1
for i in range(100):
    p*=i+1
print(f"The product of 1,2,…,100 is {p}")
```

This prints the following:

```
Sum of 1,2,…,100 is 5050
The product of 1,2,…,100 is 933262154439441526816992388562667004907159682643816214685929638952175999932299156089414639761565182862536979208272237582511852109168640000000000000000000
```

Our next sample program in Jupyter Notebook within VS Code creates a simple data visualization to showcase how Python can be used for that purpose.

The data for x-axis are a list of letter grade in a grading system; the data for y-axis contain a list of numbers representing how many students received each corresponding grade in a class. The purpose of visualization is to see how the grades are distributed within the class.

https://doi.org/10.15215/remix/9781998944088.01

```
import matplotlib.pyplot as plt

x = ['D','C-','C','C+','B-','B','B+','A','A','A+']
y = [6,9,12,19,23,28,15,13,7,5]
plt.bar(x,y)
plt.title('A showcase')
plt.xlabel('Letter Grade')
plt.ylabel('# of Students')
plt.show()
```

Here we use a module called Matplotlib to visualize the data by plotting graphs. The result of the program as shown in Figure 1-6.

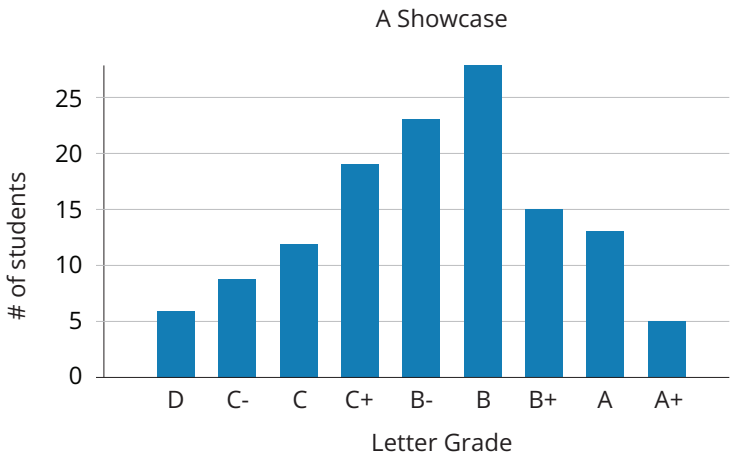


Figure 1-6: Graph produced by a Python script in Jupyter Notebook

As you can see, with only 11 lines of code, you can produce a nice graph to visualize the data in Python.

Run Python Programs Outside IDE

The VS Code IDE we used in previous sections is good for developing programs or applications, but it is impractical to start an IDE each time you need to run a Python program.

So how can we run a Python program or scripts stored in a file? In a previous section, we wrote a program called primetest.py. To run the program without invoking VS Code, we need to take the following steps:

1. Start Windows PowerShell or Windows Command Prompt by typing "terminal" in the search field on the Windows taskbar if you are on

https://doi.org/10.15215/remix/9781998944088.01

other platforms such as Linux. A shell terminal will come up; type the following:

```
PS S:\Dev\Learn_Python>
```

2. Within the terminal, go to the directory where the Python program file is located.
3. Change the working directory to that folder by typing the following PowerShell command:

```
cd  S:\Dev\Learn_Python\samples\
```

4. Run the Python program file by typing the following command at the PowerShell prompt:

```
python .\primetest.py
```

the result is shown below:

```
PS S:\Dev\Learn_Python> cd .\samples\
PS S:\Dev\Learn_Python\samples> python .\primetest
.py
give me an integer that is greater then 1, and I
will tell you if it is a prime: 23
23 is a prime
PS S:\Dev\Learn_Python\samples>
```

Note that to run a Python program from a terminal, two conditions must be met: (1) The location of the Python interpreter (python.exe) must be in the PATH system variable, so that Windows is able to find it. (2) The Python program file should be in the current working directory of the terminal. If that is not the case, you must either change your current working directory to where the Python program file is located or specify the path to the program file. Assume we change the current working directory from the one shown at the end of the list above to the one shown here using the command cd:

```
PS S:\Dev\Learn_Python>
```

https://doi.org/10.15215/remix/9781998944088.01

Because now the primetest.py program is one-level down from the current working directory at .\samples, where the leading dot (.) refers to the current directory, to run the program, you will have to specify the path to the program, as shown here:

```
PS S:\Dev\Learn_Python> python .\samples\primetest.py
give me an integer that is greater than 1, and I will
tell you if it is a prime: 91
91 is divisible by 7, so that 91 is not a prime
PS S:\Dev\Learn_Python> python .\semples\primetest.py
give me an integer that is greater than 1, and I will
tell you if it is a prime: 23
23 is a prime
PS S:\Dev\Learn_Python>
```

Make the Python Program File Executable

Sometimes, we may still consider it inconvenient to run a Python program file from a terminal by feeding the file to the Python interpreter and rather prefer to make the program executable on its own so that it can be run by clicking the file within Windows File Explorer. Luckily, a tool called pyinstaller can be installed and used to do that. To install it, you need to run PowerShell or another shell terminal as an administrator. Within the terminal, run the command shown here to install pyinstaller using the pip tool:

```
PS S:\Dev\Learn_Python> pip install pyinstaller
Requirement already satisfied: pyinstaller in s:\python\
python311\lib\site-packages (5.7.0)
Requirement already satisfied: setuptools>=42.0.0 in s:\
python\python311\lib\site-packages (from pyinstaller)
(67.4.0)
Requirement already satisfied: altgraph in s:\python\
python311\lib\site-packages (from pyinstaller) (0.17.3)
Requirement already satisfied: pyinstaller-hooks-
contrib>=2021.4 in s:\python\python311\lib\site-packages
(from pyinstaller) (2022.15)
Requirement already satisfied: pefile>=2022.5.30 in s:\
python\python311\lib\site-packages (from pyinstaller)
(2023.2.7)
```

https://doi.org/10.15215/remix/9781998944088.01

```
Requirement already satisfied: pywin32-ctypes>=0.2.0 in
s:\python\python311\lib\site-packages (from pyinstaller)
(0.2.0)
PS S:\Dev\Learn_Python>
```

Before you can use the pyinstaller tool to make the conversion, you will need to copy the Python library into Windows' system32 directory, as shown below:

```
PS C:\WINDOWS\system32> cp S:\Python\Python311\python311.
dll .
PS C:\WINDOWS\system32>
```

Now you can simply run the command pyinstaller <Python program file name> to make the conversion, as shown here.

```
PS S:\Dev\Learn_Python\samples> pyinstaller
.\primetest.py
1435 INFO: PyInstaller: 5.7.0
1435 INFO: Python: 3.11.1
1445 INFO: Platform: Windows-10-10.0.22621-SPO
1445 INFO: wrote S:\Dev\Learn_Python\samples\primetest.
spec
1460 INFO: UPX is not available.
1460 INFO: Extending PYTHONPATH with paths ['S:\\Dev\\\
Learn_Python\\samples']
pygame 2.3.0 (SDL 2.24.2, Python 3.11.1)
Hello from the pygame community. https://www.pygame.org/
contribute.html
5243 INFO: checking Analysis
5254 INFO: Building because inputs changed
5254 INFO: Initializing module dependency graph…
5254 INFO: Caching module graph hooks…
5285 WARNING: Several hooks defined for module 'numpy'.
```

The created executable is under .\build\primetest, named primetest.exe.

Please note, however, that the executable file requires the Python DLL library in order to run, so that you can either (1) make the DLL library searchable and accessible by Windows OS or whatever OS you are using, if you have many executables generated from Python program files, or

https://doi.org/10.15215/remix/9781998944088.01

(2) copy the Python DLL to where the executable file is located. In this case, since we are using Python3.11, the library is named python311.dll—which is located within the installation directory of Python3.11, which is s:\python\python311—we can simply copy the DLL file to .\build\primetest for a test, as shown below:

```
-a---    2022-12-06  8:12 PM        1463681    NEWS.txt
-a---    2022-12-06  8:10 PM         101752    python.exe
-a---    2022-12-06  8:10 PM          65912    python3.dll
-a---    2022-12-06  8:10 PM        5761912    python311.dll
-a---    2022-12-06  8:10 PM         100216    pythonw.exe
-a---    2022-12-06  8:10 PM
         49488    vcruntime140_1.dll
-a---    2022-12-06  8:10 PM
         109392    vcruntime140_1.dll

PS S:\Dev\Learn_Python\samples> cp S:\Python\Python311\
python3.dll .\build\primetest\
PS S:\Dev\Learn_Python\samples>
```

Once you have done all the steps above, you can run the program like all other apps on your computer, as shown here:

```
PS S:\Dev\Learn_Python\samples> .\build\primetest\
primetest.exe
give me an integer that is greater than 1, and I will
tell you if it is a prime: 31
31 is a prime
PS S:\Dev\Learn_Python\samples>
```

Errors in Programs

Errors in programs are hardly avoidable, and you should be prepared to see a lot of errors in your programs, especially at the beginning. The nice thing about programming within an IDE such as Visual Studio Code is that the IDE can point out many syntax errors and coach you to code along the way by showing you a list of allowable words or statements that you may want to use in that context, although you will have to decide which one to use for your program by highlighting the words that are problematic.

https://doi.org/10.15215/remix/9781998944088.01

Syntax errors often include the following:

1. Missing, misspelled, or misplaced keywords such as ***for***, ***while***, ***if***, ***elif***, ***else***, ***with***, ***class***, ***def***, and so on. Remember that Python language is case-sensitive so *for* and *For* are totally different words to Python Virtual Machine (PVM).
2. Missing or misspelled operators such as >=, <=, +=, -=, *=, /=, and so on.
3. Missing symbols, such as a colon, comma, square or curly bracket, or parenthesis.
4. Mismatched parentheses, double quotation marks, single quotation marks, curly brackets, and square brackets.
5. Incorrect indentation because Python uses indentation to form code blocks.
6. Empty code blocks. If you do not know what to write in a code block, you can simply use a pass statement as a placeholder, as shown in the following examples.

```
"""
This function will find the best student in a class based
on their over performance, but at this time we don't know
how.
"""

def findBest():
 pass   # the pass statement is used to hold the place
for a code block
```

The second type of errors you may encounter are runtime errors. While syntax errors may be easily avoided when programming within VS Code IDE, runtime errors can only be found during runtime. Examples of runtime errors include dividing a number with 0 and using a variable that has not been defined or that has no value before it is used. The discussion of error and exception handling, covered in Chapter 4, is mostly about runtime errors.

The following are some common runtime errors in Python programs:

1. ZeroDivisionError, when dividing something by 0.
2. TypeError, when an operation is performed on incompatible data types.

https://doi.org/10.15215/remix/9781998944088.01

3. ValueError, when an incorrect value is used in a function call or expression.
4. NameError, when an identifier is used that has not been defined. In Python, *define* means that it has been assigned a value, even if the value is None.
5. IndexError, when the index used to access a sequence is out of boundary.
6. KeyError, when an incorrect key is used to access a dictionary value.
7. AttributeError, when an object attribute is sued that does not exist.
8. FileNotFoundError, when a file that one is trying to open to read does not exist. It is OK to open a file to write that does not exist. The operation will create a new file in that case.

In addition to syntax and runtime errors, you can also have logical errors in your programs. These errors can be found only by you or users of the program. An example of logical errors is when you are supposed to add two numbers, but you subtract one from the other instead.

Throughout the text, you will learn and gradually gain the ability to make your programs logically correct and to identify logical errors in your programs.

1.7 Essentials of Problem Solving and Software Development

Before you learn how to program in Python, you need to gain a basic understanding of the steps taken by computers to solve a problem and the steps taken by programmers to develop a software system for real-world application. The former, steps taken by computers to solve a problem, is called an algorithm. The latter, steps taken by programmers to develop an information system for real-world application, is in the area of system analysis and design.

Design Algorithms to Solve Problems

An algorithm is a sequence of instructions showing steps to solve a problem or get something done. The recipe for cooking a dish is an example of an algorithm from our everyday lives. Unlike algorithms for people to follow, algorithms for computers must show definitive steps of explicit operation.

Consider a very simple task for a computer to complete: give a computer two numbers and ask the computer to find the sum and print out the result. The algorithm can be described as follows:

https://doi.org/10.15215/remix/9781998944088.01

[algorithm 1] Get two numbers from user, calculate, and print out the sum.
Step 1. Get the first number to n1.
Step 2. Get the second number to n2.
Step 3. Calculate the sum of n1 and n2, and store the result to s.
Step 4. Print out s.
[end of algorithm 1]

Describing the steps of operations as shown above is just one way to present algorithms. In computing and software development, algorithms can also be, and more often are, presented using pseudocode or a flowchart.

Pseudocode is a language that is easier for humans to understand but that is written like a computer program. There is no widely standard vocabulary and grammar for pseudocode. However, within an organization or even a class, the same pseudocode language should be used for collaboration and communication.

The above algorithm can be described in pseudocode as follows:

```
Start
Input from user → n1 # get an input from user and
 assign it to n1
Input from user → n2 # get another input from user
 and assign it to n2
n1 + n2 → s # n1 + n2 and assign the sum to s
print(s)
End
```

This simple algorithm can also be described using a flowchart, as in Figure 1-7.

In problem solving and computing, conditional operations and repetitive operations are often needed. An example is to calculate the sum of number 1, 2,…10000. One might think that the sum could be calculated by writing 1 + 2 + 3 + … + 10000, but that is wrong because that mathematical expression cannot be precisely understood by computers. The correct algorithm should be:

[algorithm 2] Calculate the sum of all positive integers that are no greater than 10000.
Step 1. 1 → i, 0 → s
Step 2. s + i → s, i +1 → i
Step 3. If i <= 10000 go to step 2 *# loop back and make repetition under condition*

https://doi.org/10.15215/remix/9781998944088.01

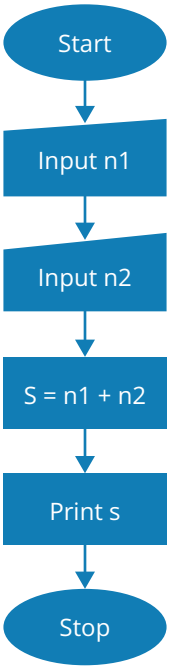


Figure 1-7: Flowchart of a simple algorithm

Step 4. Print s
[end of algorithm 2]

In pseudocode, the algorithm can be described as follows:

```
Start
Initialization: i = 1, s = 0
Repeat
  s = s + i
  i = i + 1
Until i > 10000
End
```

The algorithm can be depicted using a flowchart, as in Figure 1-8.

When programming to solve a problem, first develop an algorithm describing the steps of how the problem can be solved using computers, even if it is not explicitly and formally formulated as above.

Phases of Software System Development

Algorithm design and representation are essential when programming for computers to solve a problem or complete a task, but it is only part of software

https://doi.org/10.15215/remix/9781998944088.01

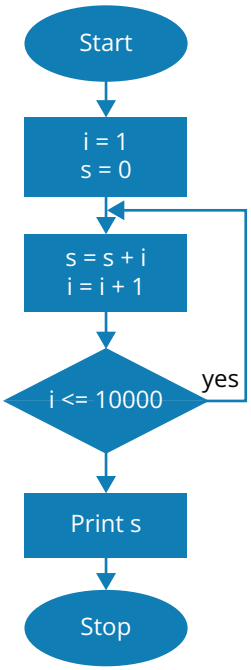


Figure 1-8: Flowchart of an algorithm

system development. Developing a software system to satisfy the requirements of a client requires several phases.

PHASE 1. UNDERSTAND THE PROJECT

When developing a software system at the request of a client or customer, the first thing to do is to understand what the client really needs so that you can get a clear definition of the problems to be solved and the tasks to be completed by the software system.

PHASE 2. ANALYZE THE REQUIREMENTS TO IDENTIFY COMPUTER-SOLVABLE PROBLEMS AND TASKS

This phase is very important in turning people problems into computer problems, problems that can be solved by computers. Your clients and customers often have limited knowledge of what computers can do, and their requirements may be clear for humans, but not directly doable for computers. Requirement analysis will turn client's requirements into problems and tasks suitable for computers to solve or complete. This often involves a strategy called divide and conquer, which means dividing big problems or tasks into smaller ones to solve. This strategy is widely used not only in computing but also in other disciplines and practices. Your understanding of the application domain and your knowledge of all aspects of computing are very important during this phase of software system development.

https://doi.org/10.15215/remix/9781998944088.01

PHASE 3. DESIGN THE SYSTEM

System design involves the identification of modules and their connections. Module identification can be based on an understanding of the project and analysis of the requirements. The problems and tasks identified in phase 2 should be clustered into different modules of the system, though the same problem or task may exist in different modules. The next important thing at the system design phase is to design algorithms for each of the identified problems and tasks. Your knowledge and skill in system design, problem solving, and mathematical and logical thinking play important roles at this phase.

PHASE 4. IMPLEMENT THE SYSTEM

System implementation is programming, which turns the algorithms into computer programs. Your knowledge of the language chosen to implement the system is vital. Implementation involves programming to solve the problems and complete the tasks identified at phase 2, based on the structure of the system and algorithms designed at phase 3, as well as integration of all the subsystems and modules to make them work together.

PHASE 5. TEST THE SYSTEM

The system implemented at phase 4 may not correctly reflect what the client wants and may have errors and bugs. A thorough test is needed to find the bugs and close up the gaps between what has been implemented and what the client really wants.

PHASE 6. MAINTAIN THE SYSTEM

After the system has been fully tested and accepted by the client, the system will be delivered to the client. But that is not the end of the project. Maintaining the system is often an even bigger task after the release and delivery of the software because there is no guarantee that all bugs were found during the test, no guarantee that the implementation genuinely reflects the needs of the client, and no guarantee that the client will not come up with a "better" idea or requirements after the fact.

1.8 Manage Your Working Files for Software Development Projects

When working on a software development project, you need to deal with many files and make many changes to the files. Later, you might realize that some changes to a file are incorrect or do not improve the file, so you might want to

https://doi.org/10.15215/remix/9781998944088.01

go back and use a previous version of the file. It may not be a big issue if you are only dealing with one or just several files, but it will be very difficult when many files are involved in a project. That is why a version-control system is needed. Use a version-control system called Git to manage all your working files on the exercises and projects included in this text.

Originally developed by Linus Torvalds, Git is an open-source-distributed version-control system for tracking changes in any set of files, including Jupyter Notebook files with the extension ipynb.

In this section, you will learn

1. how to install and set up Git on your computer and use it to version-control your project files locally.
2. how to set up an account at GitHub and version-control your project files using Git and remote repositories on GitHub.
3. how to use Git and GitHub within VS Code to version-control your project files.

Set Up Git on Your Computer and Version-Control Locally

Git can be freely downloaded from https://git-scm.com/downloads and installed on your computer. After installing Git, you can create a local Git repository on your computer by running the git init command within a shell window such as CMD or PowerShell on Windows. A preferred one is Git-Bash shell, which comes with Git. It can be started within Windows File Explorer.

For example, if you want to start a project called Learn_Python, you can create a folder called Learn_Python on your desktop within Windows File Explorer, then right-click the folder icon, and choose from the pop-up menu. A Git-Bash window will open with Learn_Python as the current working directory, as shown here:

```
james@trustshop MINGW64 /s/Dev/Learn_Python
$
```

Within this Git-Bash shell window, you can run the following command to initialize:

```
$ git init
```

You have to run this Git command once to create a local repository for a project. It will create an empty Git repository on your local computer, which is a directory named .git with specific subdirectories and files that Git will use to

https://doi.org/10.15215/remix/9781998944088.01

version-control your project files. It will also create a master/main branch for the project. Other branches can be created if the project is divided into smaller ones to be worked on in parallel and then later merged.

Before running the command, you can set up an environment variable called $GIT_DIR to determine where the local repository will be placed on your computer. If $GIT_DIR is not set, the local Git repository—that is, the .git subdirectory—will be created right under the current working directory; if $GIT_DIR is set in a different location of your choosing, .git will be created right under $GIT_DIR. The common location of .git for a project is right under the project directory.

For the system to know who has made changes to the files and how to communicate with them, the following two commands need to run to config the Git that you installed on your computer:

```
$ git config -global user.name "Jon Doe"
$ git config -global user.email "Jon.Doe@gmail.com"
```

These commands will configure the user's name and email address. Please keep in mind that this username is different from the username that you will use to sign up for your account with GitHub later in this section.

When changes have been made to a file and you want Git to manage the changes to the file, the first step is to stage the file by running the following command:

```
$ git add <file/files>
```

This is called "staging changes to the file," which is the first step Git takes to version-control files.

For example, if you want to stage changes to all Python program files under the current project directory, run the following command:

```
$ git add *.py
```

To stage changes to all files and even subdirectories under the current working directory, run the following command:

```
$ git add .
```

The next step Git will take to version-control your project files is called commit. It is done by running the following command:

https://doi.org/10.15215/remix/9781998944088.01

```
$ git commit -m "A brief message about this commitment"
```

Now changes to the files have been committed to the local repository. Please note that the stuff inside the quotation marks behind option -m are the comments or notes about the changes. If you run Git commit without option -m, a text editor will pop up to let you enter comments.

You can check the status of your Git system by running the following command:

```
$ git status
    On branch master
    No commits yet
    Changes to be committed:
    (use "git rm --cached <file>…" to unstage)
       new file: start.py
```

This shows that the file start.py has been staged but not committed yet. You can still use the git rm command to unstage the file (remove it from the staged list).

In the above, we basically described a process of making changes to a file, staging the files for the changes, and committing the changes to the repository. You may, and most likely want to, reverse the process to undo some changes.

To a file that has been staged but not committed, use the following command to unstage it:

```
$ git rm -cached <file>
```

or

```
$ git reset HEAD <file>
```

To a file that has not been staged, use the following command to discard the changes to the file:

```
$ git checkout -- file
```

To revert a commit that has been made, run the following command:

```
$ git revert HEAD
```

https://doi.org/10.15215/remix/9781998944088.01

To restore a file to one that is n versions back in the master branch of the repository, run the following command:

```
$ git restore –source master~n <file>
```

There are some other Git commands available for various needs of version control, and each has many options providing powerful functionalities. A list of these commands can be found at https://git-scm.com/docs/git#_git_commands. You can also get the GitHub-Git cheat sheet at https://training.github.com/downloads/github-git-cheat-sheet/.

For Git, files that have been staged or committed for changes are called tracked, and those that have not been staged or committed are called untracked. In software development, some files, such as those objects and executable files derived from code files, do not need to be tracked for changes, so they should be ignored by Git. To tell Git what files under a project directory can be ignored, especially when running the command to stage all, you can add the file names or ignore patterns to a special file called .gitignore right under the project directory. You need to manage the list in the .gitignore file by editing the file using a text editor. Details of ignored file patterns can be found from Git documentation at https://git-scm.com/docs/gitignore. For our purposes, it may be sufficient to just describe the files based on what you already know, such as a file name or file names with a wildcard such as *.log.

Note that unlike other version-control systems, Git does not work with files for different versions, but works only with the changes that led to the current version of the file. For each file, Git will only keep one copy of the file for its current version in its repository and keep only the changes that led to the current version for all older versions.

Set Up an Account on GitHub and Version-Control with Remote Repositories

With Git, you can also use remote repositories on GitHub, which is a web-based system that you can use to create, access, and manage your remote repositories on a GitHub server. The benefits of working on a remote repository like GitHub for a software development project are twofold: the first is that you do not need to worry about the possible loss of your computer or the corruption of the file system; the second is that a team of programmers can work on a project at the same time efficiently and globally.

To use the remote repository on GitHub, first create an account with GitHub .com. GitHub offers students use of their repositories free of charge but also provides a pack of paid professional tools for software development.

https://doi.org/10.15215/remix/9781998944088.01

Once you have signed into GitHub.com, you can create a repository for each of your projects under your account. Once created, each repository will have a unique URL where project files can be synchronized between the local repository and the remote repository, which can be cloned/downloaded. A repository on GitHub can be either public or private. A private repository can be accessed only by the owner, whereas a public repository can be accessed by everyone on the internet. Because you will be using the remote repository for your assignment projects, you should choose private so that your assignment work will not be shared with any others. You can create public repositories for projects that are substantially different from projects in the assignments and interesting enough for others to collaborate with you on the projects.

Now you know how to create both a local repository and a remote repository, and you've learned how to use Git commands to move project files to and from the local repository.

To continue work on a remote repository for the version-control of project files, you need to tell the computer where the remote repository is by using the following Git command:

```
$ git remote add <remote repository name> URL
```

Git supports several network protocols to communicate between your local computer and remote repository servers such as GitHub, including SSH and HTTPS. Because using SSH requires additional setup on your computer, we recommend HTTPS. The following is an example of adding the remote repository previously created for Jupyter Notebooks to our local notebook repository for Jupyter Notebooks:

```
$ git remote add notebooks https://github.com/jamesatau/
comp218-notebooks.git
```

You can also use the following to delete a remote repository:

```
$ git remote remove <remote repository name>
```

And you can use

```
$ git remote rename <remote repository old name> <remote
repository new name>
```

to rename a remote repository.

https://doi.org/10.15215/remix/9781998944088.01

You can also make changes to the URL of an existing repository, as shown in the following example:

```
$ git remote set-url origin https://github.com/jamesatau/
allgitcases.git
```

To view which remote repository is configured for the current project, use the following Git command:

```
$ git remote -v
```

The command below initializes a git repository and adds remote notebooks:

```
$ git init
Initialized empty Git repository in S:/Dev/Learn_Python/.
git/

james@trustshop MINGW64 /s/Dev/Learn_Python (master)
$ git remote add notebooks https://github.com/jamesatau/
comp218-notebooks.git

james@trustshop MINGW64 /s/Dev/Learn_Python (master)
```

The text below shows a git command to display the remote connections configured after running the commands above:

```
$ git remote -v
notebooks   https://github.com/jamesatau/comp218
-notebooks.git (fetch)
notebooks   https://github.com/jamesatau/comp218
-notebooks.git (posh)

james@trustshop MINGW64 /s/Dev/Learn_Python (master)
$
```

Between the local repository and the remote repository, project files can go in two directions. Moving files from the local to the remote repository is referred to as a *push*; moving files from the remote to the local repository is a *pull* or *fetch*. Between pull and fetch, pull is the default mode of moving files

https://doi.org/10.15215/remix/9781998944088.01

from remote to local, and fetch provides additional power, such as is needed when moving files from several repositories at the same time.

Please note that pull or push operations are not just transferring files. Key points of the operation are merging the changes to files in the target repository and versioning. Git has special commands for those key operations, but those operations often run behind the scenes without anyone noticing.

After you have added a remote repository to your project, you can push the already committed changes to the local repository by running the git push command in a shell terminal within the project directory, as shown here:

```
$ git push --set-upstream notebooks master
Enumerating objects: 3, done.
Counting objects: 100% (3/3), done.
Delta compression using up to 6 threads
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 545 bytes | 272.00 KiB/s,
done.
Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
remote:
remote: Create a pull request for 'master' on GitHub by
visiting:
remote:   https://github.com/jamesatau/comp21B-notebooks/
pull/new/master
remote:
To https://github.com/jamesatau/comp218-notebooks.git
* [new branch] master -> master
branch 'master' set up to track 'notebooks/master'.

james@trustshop MINGW64 /s/Dev/Learn_Python (master)
$ git push notebooks
Everything up-to-date

james@trustshop MINGW64 /s/Dev/Learn_Python (master)
$
```

Sometimes you might be interested in a project in a repository and would like to make a local copy of the entire project. The operation of copying an entire project in a remote repository onto a local computer is called cloning. For example, say there is a repository called learn-python3 in GitHub containing

https://doi.org/10.15215/remix/9781998944088.01

sample code, and you want to download all the code to play with it. You can do that by doing the following:

1. Open a terminal on a file folder in which the project is to be placed—say, the desktop.
2. From GitHub.com, copy the HTTPS URL from the "Code" dropdown menu of the learn-python3 repository and run the following Git command:

   ```
   $ git clone https://github.com/michaelliao/learn--
   python3.git
   ```

   You will then have all the code samples on your computer.

To work with Git and GitHub, you can use Git GUI, a GUI-based system, by invoking Git GUI on the project folder within Windows File Explorer.

Another way, our preferred way to use Git and GitHub for versioning on both local and remote repositories, is to operate within VS Code, which has native support for Git and GitHub, as long as Git is properly installed on the local computer, as shown earlier in this section.

1. Create a new folder on your local drive for a project.
2. Open the folder from VS Code. The new folder can also be created when you open the file folder in VS Code.
3. Initialize the local repository for the project by running git initialize in VS Code command palette by pressing Ctrl+Shift+P. Type "git" and click "git initialize." You will be asked to do the following:
   a. Choose a local workspace folder to initialize.
   b. Set the email address of the programmer/developer/author.
   c. Set the name of the programmer/developer/author.

   Now the local repository is ready for versioning your project files locally.
4. Create or copy project files under the project folder.
5. Click the source control icon on the left navigation bar, type a brief message about the changes to the files or project, and press Ctrl+Enter or click the commit icon above.
6. To push the changes to the project files and the project to the remote repository in GitHub, click the Git Sync icon at the bottom-left of the VS Code window.

https://doi.org/10.15215/remix/9781998944088.01

a. If this is the first time you are pushing the project to your GitHub account, the command palette dialogue will pop up and ask you to choose whether to publish the project into a public repository or private repository under your account. The project folder name will be used as the repository name by default, but you can use a different name.
b. You may be asked to sign into your GitHub account if you have not accessed GitHub from VS Code before.
c. You may also be asked to either use the existing username and email associated with your GitHub account or provide a different email and/or username to identify yourself as the author of the project being pushed.

If you have already created a repository for the project on GitHub, you can copy the URL of the repository and add the remote repository to the project in VS Code by running git add remote in the command palette. You will also be asked to provide a name for the remote repository after entering the URL.

Chapter Summary

- The first chapter has introduced you to computers and programming languages and guided you in setting up the programming environment.
- The introduction to modern computers included a brief history as well as a description of the fundamental architecture of modern computers. Knowing who made significant contributions to the development of computers, what contributions they made, and how they made those contributions can be very inspiring.
- The computers we use today are called Von Neumann machines because they are based on Von Neumann architecture, which consists of a control unit, an arithmetic logic unit (ALU), registers, and a memory unit (collectively called CPU) plus inputs and outputs. That's why Von Neumann is often credited as one of the fathers of modern computers.
- The key features of modern computers are that they are digital, programmable, and automatic, with stored programs, although these features were already in the design of the analytical engine by British mathematician Charles Baggage in the 1800s. Hence Charles Babbage is also credited as a father of modern computers, although his machines were not electronic.

https://doi.org/10.15215/remix/9781998944088.01

- Binary or Boolean algebra is one of the important theories behind today's modern computers. It can be proved that computing with a ternary number system would be more efficient, but it would be more costly to make computing components to build computing machines based on a ternary number system than on a binary number system.
- Signed numbers, including both integers and real numbers, need to and can be represented by a sequence of digits, using the highest bit to represent the sign (0 for positive, 1 for negative). A certain number of digits are assigned for decimals.
- To apply arithmetic operations more easily and efficiently on computers, 2's complements are used to represent signed numbers. With 2's complements, the addition of signed numbers can be easily done.
- How do you know what modern computers are and are not capable of? Alan Turing, independently from Alonzo Church, laid the foundational work on computability with his Turing machine. Because of that, Alan Turing is credited as a father of modern computing.
- There is also the question of how difficult it is to solve a problem with computers, which is the study of computational complexity. The computational complexity of a problem is often measured in terms of the total number of basic computations, such as addition and multiplication, which can be converted into the time needed to solve the problem on specific computers. The space required to solve a problem can be a concern as well, but most of the time, when people are talking about computational complexity, they are talking about the steps or time required to solve the problem.
- Problems to be solved on computers are often divided into three classes of problems: P, NP-complete, and NP, in which P is short for *polynomial* and NP is short for *nondeterministic polynomial*.
- If, on a deterministic and sequential machine like a computer, for a problem of size n, if the time or number of steps needed to find the solution is a polynomial function of n, the problem is said to be in the P class. If a problem can be solved in polynomial time on a nondeterministic machine, the problem is in the NP class. A problem is said to be NP-complete if proposed answers can be verified in polynomial time, and if an algorithm can solve it in polynomial time, the algorithm can also be used to solve all other NP problems.
- Programs are the computers' soul. The task of writing programs for computers is called programming. Languages in which programs can be written for computers are programming languages.

https://doi.org/10.15215/remix/9781998944088.01

- Programming languages play important roles in building soul into computers. Programming languages can be machine languages, assembly languages, and high-level languages.
- Ada Lovelace, who wrote code for Charles Babbage's analytical engine, was credited as the first programmer of modern computers. The Ada programming language was named in her honour.
- An algorithm describes the steps a computer needs to take to solve a problem or complete a task.
- Pseudocode and flowcharts are two ways of describing algorithms.
- System analysis and design are the steps taken to design and develop an information system for real-world application.
- The official website for the Python programming language is at www .python.org. Use the Anaconda package to install Python, Jupyter Notebook, Visual Studio Code IDE, and other tools for your study of programming with Python.
- Python interactive shell and Jupyter Notebook are recommended for learning Python programming interactively.
- Create a Jupyter Notebook for each chapter and/or section, and work through all the examples within that part of the course in that notebook to keep a record of your learning activities for review and evaluation.
- Visual Studio Code (VS Code) is the IDE recommended for you to complete the projects and programming projects in the assignments.

Exercises

1. Convert the following numbers in their respective bases into their binary equivalence:

$(78)_{10}$	$(1F)_{16}$	$(27)_{8}$
$(121)_{10}$	$(3E)_{16}$	$(33)_{8}$
$(29)_{10}$	$(CD)_{16}$	$(52)_{8}$

2. Complete the following binary operations:

10111 + 1101	1101010 – 101101	10101 + 1110
10101 – 1111	1101 * 101	111011 / 11

https://doi.org/10.15215/remix/9781998944088.01

3. Complete a thorough review of computer history by reading library books and online articles. Compile a table showing the significant developments of theories and technologies of modern computers in Europe and a similar table for America. For each development, the table should show the time, a description of the development, the key players/contributors, and its impact on the later development of modern computers.
4. Investigate the classification and development of computer programming languages and explain the features of the following programming languages:
   a. Machine code
   b. Assembly languages
   c. High-level programming languages
   d. Procedural programming languages
   e. Structured programming languages
   f. Imperative programming languages
   g. Functional programming languages
   h. Logic programming languages
   i. Object-oriented programming languages
5. Create a new folder called comp218, open it with VS Code, then create a new file and rename it comp218start.ipynb, which is recognized by VS Code as Jupyter Notebook. Start working in the cells and use it as a calculator to see what expressions and statements you can perform correctly with Python, based on what you have learned so far. Please note that you may wish to choose a Python interpreter in order to run the code or make sure the notebook is working properly.
6. In a cell of a newly created notebook comp218start.ipynb, type the following code and then press Shift+Enter to run it to see what you will get:

```
In [ ]:  first_name = 'John'
         last_name = 'Doe'
         full_name = first_name + ' ' +last_name
         print(full_name)
```

7. In a new cell of notebook comp218start.ipynb, type the following code and then press Shift+Enter to run the code to see what you will get.

```
In [ ]:  for i in range(11):
          p = (2**i)
           print(f'{bin(p)}')
```

https://doi.org/10.15215/remix/9781998944088.01

8. Write an algorithm in pseudocode describing the steps to make a pizza.
9. Write an algorithm in pseudocode describing the steps to cook rice.
10. Write an algorithm in a flowchart describing the steps you usually take in the morning, from getting up until leaving home to go to work or school.

Projects

1. Research the history of computers online, then write a summary about each of the following:
   a. The computing machines designed and developed in history that are significant to the development of today's computers
   b. The people in history who have made significant contributions to the development of computers
   c. The concepts, theories, designs, and technologies in history that are important to the development of modern computers
2. Research the history of programming languages online, then write a summary for each of the following:
   a. Machine languages that have been important to the development of computer systems
   b. The differences between machine languages, assembly languages, and high-level programming languages
   c. The essence (what defines it and differentiates it from others) of structural programming, the pros and cons of structural programming, and some well-known programming languages that support structural programming
   d. The essence of imperative programming, the pros and cons of imperative programming, and some well-known programming languages that support imperative programming
   e. The essence of declarative programming, the pros and cons of declarative programming, and some well-known programming languages that support declarative programming
   f. The essence of functional programming, the pros and cons of functional programming, and some well-known programming languages that support functional programming
   g. The essence of logical programming, the pros and cons of logical programming, and some well-known programming languages that support logical programming
   h. The essence of object-oriented programming, the pros and cons of object-oriented programming, and some well-known programming languages that support object-oriented programming

https://doi.org/10.15215/remix/9781998944088.01

This page intentionally left blank

Chapter 2

Essential Building Blocks of Computer Programs

This chapter introduces the fundamental elements and building blocks of computer programs in the Python language. These fundamental building blocks include the words, keywords, and reserved words that can be used in Python programs. You will also learn the primary data types and data models, the operators, and some built-in functions that can be applied to the data and data models in Python, as well as some important statements or sentences that can be used to instruct computers.

Learning Objectives

After completing this chapter, you should be able to

- make and use names correctly to identify various items in your Python programs.
- use different types of data and data models correctly.
- use the proper operators for different types of data and data models.
- correctly compose expressions using variables, data, operators, and the built-in functions of Python.
- write an assignment statement to assign values to variables.
- write augmented assignment statements.
- write ***input*** statements correctly to get input from users.
- write correct statements using ***print***.

https://doi.org/10.15215/remix/9781998944088.01

2.1 Primary Constructs of Computer Programs in Python

Computer programs tell computers how to complete specific tasks. These specific tasks may be as small as what you would do on a calculator or as big as mission control for space exploration. Believe it or not, computer programs for all these different tasks are composed of the same types of statements on the same types of primary data. The difference is in how the data are structured and how the statements are organized. In computing, the study of the first "how" is the subject of data structures, whereas the study of the second "how" is the subject of algorithms. In computing, there are dedicated courses providing in-depth coverage on data structures and algorithms. While studying introductory computer programming, keep in mind the importance of data and algorithms in solving problems and try to identify the data involved and describe the algorithms needed to solve a problem using the methods introduced in 1.7.

Vocabulary of the Programming Language

Vocabulary is the foundation of any language. In computer languages, identifiers are an important part of the vocabulary used to write computer programs. As in all programming languages, identifiers in Python are used to identify variables, functions and methods, classes, objects, and modules. They are called identifiers because, for computers, the only purpose of these names is to identify specific locations of computer memory that hold or can be used to hold specific data or code blocks.

Figure 2-1 illustrates how a variable named "grade" is identified as the memory location that holds the integer 98. Please note that we use 98 in the diagram for illustration purposes, although, in fact, both data and program codes are stored as sequences of binary (0s and 1s).

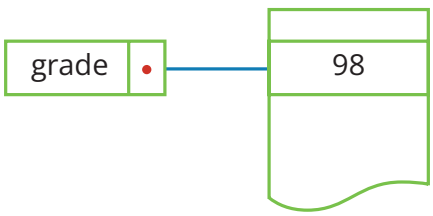


Figure 2-1: The variable "grade" and its memory location

Although an identifier in computer programs does not need to be meaningful to humans, it must be unique in a given context, which is often called namespace. In Python, a namespace is a mapping from names to objects and is implemented as a Python dictionary.

https://doi.org/10.15215/remix/9781998944088.01

In a program, different identifiers may be used to refer to the same memory location and hold the same data or program code, as shown in Figure 2-2. This is accomplished through the following Python code:

```
In [ ]:   x = 'John'
          y = x

Out [ ]:  John
```

To check what x and y hold, we use the ***print*** statement as follows:

```
In [ ]:   print('x holds ', x, ', y holds', y)

Out [ ]:  x holds John, y holds John
```

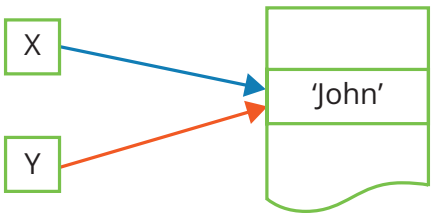


Figure 2-2: X and Y point to the same memory location

If we change one identifier to refer to something else, the value referred to by the other identifier will remain the same, as shown in Figure 2-3. This is done with the following code:

```
In [ ]:   x = 'Smith'
          print('x holds ', x, ', y holds ', y)

Out [ ]:  x holds Smith, y holds John
```

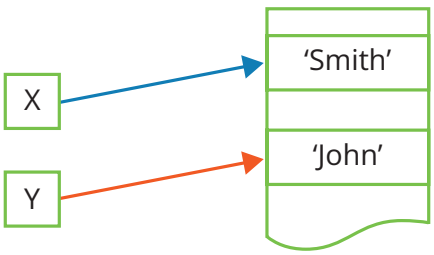


Figure 2-3: Now X and Y point to different memory locations

https://doi.org/10.15215/remix/9781998944088.01

Please note that in the examples above and other examples in the remainder of the textbook, the box behind In [ ] represents a code cell in Jupyter Notebook, whereas everything behind Out [ ] is the output from the code contained in the code cell when you press Shift+Enter or click the play button.

RULES OF NAMING IDENTIFIERS

In addition to uniqueness, identifiers used in Python programs must be named according to the following rules:

1. An identifier can be a combination of letters (a–z, or A–Z), numbers (0–9), and underscores (_).
2. It must begin with a letter (a–z or A–Z) or underscore (_).
3. In Python, identifiers are case-sensitive, so x and X are different identifiers.
4. Identifiers can be of any length.
5. User-defined identifiers cannot be the same as words reserved by the Python language.

According to these rules, the following are legitimate user-defined identifiers in Python:

```
AB, zf, cd, hz, d_2, c5E, falling, to_be
```

Whereas the following are not:

```
1d, d/, f-, g.d
```

In Python, identifiers shown in Table 2-1 are reserved and hence called reserved words or keywords, with predefined special meaning in the language, which means that you must not use them to name your variables, functions/methods, classes, or modules.

Table 2-1: Reserved words in the Python programming language

Reserved word	Special meaning	Reserved word	Special meaning
and	logical *and*	***if***	conditional statement
as	used together with ***import*** and ***with*** to create new alia	***import***	to import modules

https://doi.org/10.15215/remix/9781998944088.01

Table 2-1: Reserved words in the Python programming language *(continued)*

Reserved word	Special meaning	Reserved word	Special meaning
assert	to make an assertion for handling possible errors	***in***	membership test
break	to get out of some code blocks such as during iteration	***is***	identity test
class	to define ***class***	***lambda***	to create a lambda function
continue	to omit the rest of the code block and continue the loop	***not***	logic negation
def	to define functions	***or***	logical *or*
del	to delete an object	***pass***	to pass the code block
elif	used together with ***if*** to create a conditional statement	***print***	to output
else	used together with ***if*** to create a conditional statement	***raise***	to raise an exception intentionally
except	used together with ***try*** to handle errors	***return***	to return values from a function in function definition
exec	to execute some dynamically generated code in a string or object	***try***	used for exception handling
finally	used together with ***try*** and ***except*** to handle errors	***while***	to make loop/iteration statements
for	to create a loop	***with***	to introduce context for a code block
from	used together with import	***yield***	used in place of ***return***, to turn a function into a generator
global	to access a global variable from inside of a function		

In addition to the reserved words in Table 2-1, you should also avoid using names that have been used by Python for built-in types and built-in functions,

https://doi.org/10.15215/remix/9781998944088.01

classes, and modules. These names are collectively called built-in names. It is grammatically fine to use built-in names as user-defined identifiers, but using them may cause confusion.

Furthermore, Python also uses the underscore _ as an identifier for a very special variable to hold the result of the last evaluation when Python is running in interactive mode, as shown in the following example:

```
>>> sum ([1,2,3,4,5,6])
21
>>> sum (list(range(1000)))
499500
>>> _
499500
>>> |
```

This use of _ as a special variable can also be seen in Jupyter Notebook, as shown in the following example:

```
In [ ]:  pow(9, 3)

Out [ ]: 729

In [ ]:  _

Out [ ]: 729
```

When Python is not running in interactive mode, _ has no predefined meaning, however. Even within Jupyter Notebook, the value of _ is often unpredictable unless explicitly assigned.

Syntactically, the special variable _ can be used in the same way as others, especially when the value is to be thrown away and not used, as shown below:

```
In [ ]:  p, n = 1, 10   # assign 1 to variable p, and assign 10
           to variable n
         for _ in range(n):   # note that _ is not used
           elsewhere
            p *= 2
         print(f'2^{n} = {p}')   # format the output with
           f-string

Out [ ]: 2^10 = 1024
```

https://doi.org/10.15215/remix/9781998944088.01

PYTHON NAMING CONVENTIONS

Although identifiers used in Python programs don't have to be meaningful to humans, you should always try to use more meaningful identifiers in your programs because it is not only easy for you to tell what the identifiers are used for but also easier for other programmers to understand your programs when you work in a team or want to share your code with others.

For the same reason, you should also follow common practices and widely accepted coding conventions when programming. In terms of composing identifiers these conventions include:

1. Lower case identifiers are usually used for variables and function names.
2. Capitalized identifiers are used for class names.
3. Upper case identifiers are used for constants, such as PI = 3.1415926, E = 2.7182, and so on.
4. When an identifier has multiple words, the underscore _ is used to separate the words. So we use *your_name* instead of *yourname*, use *to_be* instead of *tobe*. Some programmers prefer not to use an underscore, but to capitalize each word, except for the first word, when an identifier is used as a variable or the name of a function or method.

Along with the programming technologies, these practices and conventions have developed over the years and may further evolve in the future. A good Python programmer should follow the developments and trends of the Python programming community.

NAMES WITH LEADING AND/OR TRAILING UNDERSCORES

As mentioned above, identifiers for variables, function/method names, and class names may begin and/or end with a single underscore, double underscores, or even triple underscores, and those names may have special meanings.

When a name has both leading and trailing double underscores, such as __init__, it is called a dunder (double-underscore) name. Some dunder names have been given special meanings in Python Virtual Machine (PVM) or a Python interpreter. They are either reserved as special variables holding special data or as special function/method names.

The following are some special dunder names used as special values or special variables holding special data.

https://doi.org/10.15215/remix/9781998944088.01

__MAIN__

Used as a special value. When a Python program/script file runs as the main program other than a module, the special variable __name__ will be assigned special value __main__.

__NAME__

Used as a special variable in a Python program to hold special value indicating how the program file is called or used. If the program file is used as the main program, __name__ will hold __main__. If it is used as a module, __name__ will hold the name of the module.

__PACKAGE__

Used as a special variable to hold the package's name if a module imported is a package. Otherwise, __package__ will hold an empty string.

__SPEC__

Used as a special variable to hold the module specification used when the module is imported. If the Python program file is not used as a module, it will hold the special value None.

__PATH__

Used as a special variable to hold the path to the module in a package. If the module is not within a package, __path__ is not defined.

__FILE__

Used as a special variable to hold the name of a Python program file.

__CACHED__

A special variable often used together with the special variable __file__, referring to a precompiled bytecode. If the precompiled bytecode is not from a program file, __file__ is not defined.

__LOADER__

Used as a special variable to hold the object that loads or imports the module so that you would know who is using the module.

__DOC__

Used as a special variable to hold the documentation of a module or Python program if it is used as the main program, documentation of a class, or function or method of a class.

https://doi.org/10.15215/remix/9781998944088.01

Dunder names used for special functions and methods will be discussed in Chapter 6 and Chapter 7.

RULES OF SCOPE RESOLUTION FOR IDENTIFIERS

Big programs for complicated applications often use hundreds or even thousands of identifiers to name variables, functions, classes, and other objects. When so many names are used, it is unavoidable that some names will be used more than once. How can we ensure in a program that each name can be mapped to an object without confusion and ambiguity? The answer is to follow the LEGB rule, in which L, E, G, and B refer to different scopes from small to big: L is for local, referring to the inside of a function or class; E is for enclosed, referring to the inside of a function enclosing another function; G is for global, referring to the space outside of all classes and function in a Python program file; and B is for built-in, referring to all the names defined within Python's built-in module. The relationships of LEGB scopes are illustrated in Figure 2-4.

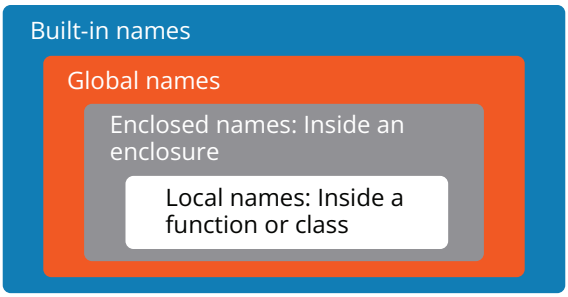


Figure 2-4: LEGB rules for scope resolution for names

The illustration above should be viewed in reference to a name used within a function or class. To resolve the name or to link it to a specific object, apply the following LEGB rules:

1. Look at the names defined locally (L) within the function/method or class. If not defined, proceed to the next rule.
2. Check whether the name has been defined in the enclosure (E) function (note that enclosures are not often used, so this is just for discussion right now). If not, proceed to the next rule.
3. Check whether it has been defined globally (G).
4. Check whether it is a built-in (B) name of Python.

The following sample shows how matching local and global names are resolved:

https://doi.org/10.15215/remix/9781998944088.01

```
In [ ]:   g_name = 'John'   # global name g_name
          l_name = 'Smith'   # global name l_name
          def display_names():   # define a function
            l_name = input('What is your name:')   # get input
            print(f'Local name is {l_name}')   # l_name is
           defined again
            print(f'Global name is {g_name}')   # g_name is
           global
          display_names()

Out [ ]:  What is your name: Kevin
          local name is Kevin
          Global name is John
```

You may have noted in the example above that variable l_name is defined both locally in definition of the function display_names() and globally. When it is used within the function, its local definition is used; variable g_name, on the other hand, is only defined globally, and when it is used within the function, it is resolved to its global definition.

In addition to the LEGB rules, please keep in mind the following:

1. A local name defined in a function will not be seen anywhere outside the function.
2. A local name defined in a class can be seen outside the class or its objects if the name is not a private member of the class, with an explicit reference to the name with dot notation. For example, a name X defined in class C can be accessed using C.X, or O.X if O is an object of C.
3. A name Nx globally defined in a Python script file named M1.py can be seen inside another Python script file by either importing the name explicitly from M or by importing M1 as a whole and using dot notation M1.Nx to access Nx.

Simple Data Types

Computers solve problems and accomplish various tasks by processing information. This information is represented in the form of data. It is important to know what data we can use or process and what operations we can apply to different types of data. In Python, there are simple data types and compound data types. The latter are also called models of data.

Simple data types include numbers, strings, and Boolean values. Numbers are used to represent numerical information, strings are used to represent

https://doi.org/10.15215/remix/9781998944088.01

literal information, and Boolean values are used to represent the result of tests, either True or False.

In Python, numbers can be signed integers, float numbers, and complex numbers. They all can be values of variables, as we shall see in the next section.

Although a string can be used conveniently as simple data, it has all the properties and supported operations of a compound data type. As such, it will be discussed in full detail later in this section.

SIGNED INTEGERS (INT)

Signed integers in Python are ...−2, −1, 0, 1, 2..., as examples. In Python 3, signed integers can be of arbitrary size, at least theoretically, as long as computer memory is not exhausted. In implementation, however, the biggest integer is defined by sys.maxsize, a variable named maxsize in a module called sys, which specifies the maximum number of bytes that can be used to represent an integer number. The notation of sys.maxsize here means maxsize, defined in module sys.

Operations on integer numbers include the following:

addition (x + y)
subtraction (x − y)
multiplication (x * y)
division (x / y)
negation (−x)
exponentiation (x ** y)
modular (x % y)
integer division (x // y)

You should be able to use the above operations, which you should already be familiar with, as well as the following bitwise operations you may never have heard about:

bitwise *or* (x | y)

```
>>> 1 | 4
5
```

bitwise *exclusive or*, often called *XOR* (x ^ y)

```
>>> 1 ^ 2
3
```

bitwise *and* (x & y)

```
>>> 1 & 5
1
```

https://doi.org/10.15215/remix/9781998944088.01

shifted left (x << n)

```
>>> 2 << 3
16
```

shifted right (x >> n)

```
>>> 256 >> 5
8
```

invert (~x)

```
>>> ~128
-129
```

The following are a few more samples from the Python interactive shell that show how these operators are used:

```
>>> 12 + 23
35
>>> 35 - 12
23
>>> -123
-123
>>> 123 * 567
69741
>>> 69741/123   # the result is a real or float-point
number
567.0
>>> 69741//123   # get the quotient of two integers
567
>>> 69741%12   # operation % will get the remainder
9
```

The next few examples are about bitwise operations. The first two operations on the first line show the binary form of the two numbers. In Python, you can have two or multiple statements on a single line, but you are not encouraged to do so.

```
>>> bin(123); bin(567)   # how to have two or more
statements on one line
'0b1111011'
'0b1000110111'
>>> bin(123 | 567)   # it will be 0001111011 | 1000110111
'0b1001111111'
```

https://doi.org/10.15215/remix/9781998944088.01

```
>>> bin(123 ^ 567)   # it will be 0001111011 ^ 1000110111
'0b1001001100'
>>> bin(123 & 567)   # it will be 0001111011 & 1000110111
'0b110011'
>>> bin(123 << 5)   # it will be 1111011 << 5
'0b111101100000'
>>> bin(123 >> 5)   # it will be 1111011 >> 5
'0b11'
>>> bin(~123)
'-0b1111100'
```

There are also many built-in functions available for operations on integers. All the built-in functions of Python will be discussed in detail below.

In addition, the following methods are also available to use for operations on integer objects.

N.BIT_LENGTH()

This returns the number of necessary bits representing the integer in binary, excluding the sign and leading zeros.

```
>>> n = -29
>>> print(f'Binary string of {n} is {bin(n)}')
binary string of -29 is -0b11101
>>> print(f'# of significant bits of {bin(n)} is
{n.bit_length()}')
# of significant bits of -0b11101 is 5
```

N.TO_BYTES(LENGTH, BYTEORDER, *, SIGNED=FALSE)

This returns an array of bytes representing an integer, where length is the length of bytes used to represent the number. byteorder can take Big-Endian or Little-Endian byte order, depending on whether higher-order bytes (also called most significant) bytes come first or lower-order bytes come first, and an optional signed argument is used to tell whether 2's complement should be used to represent the integer.

```
>>> n = 256
>>> n.to_bytes(2, byteorder = 'big') # big means higher
bytes first
b'\x01\x00'
```

https://doi.org/10.15215/remix/9781998944088.01

Recall that modern computers use 2's complements to represent negative numbers. So if the integer n is negative while ***signed*** remains False, an OverflowError will be raised, as shown below:

```
>>> n = -23567
>>> n.to_bytes(3, 'big')

OverflowError Traceback (most recent call last)
<ipython-input-4-66989275e22d> in <module>
1 n = -23567
----> 2 n.to_bytes(3, 'big')
OverflowError: can't convert negative int to unsigned
```

So a correct call of the method would be

```
>>> n = -23567
>>> n.to_bytes(3, 'big', signed = True)

b'\xff\xa3\xf1'
```

INT.FROM_BYTES(BYTES, BYTEORDER, *, SIGNED = FALSE)

This classmethod returns the integer represented by the given array of bytes.

```
>>> n = 256
>>> bin(n)
'0b100000000'
>>> # convert '0b100000000' in Big Endian to int
>>> int.from_bytes(n.to_bytes(2, byteorder = 'big'),
'big')
256
>>> # convert '0b100000000' in Little Endian to int
>>> n.from_bytes(n.to_bytes(2, byteorder = 'big'),
'little')
1
```

Note that when two bytes are used to represent integer 256, 0b100000000 will be expanded to 00000001 00000000. In Big Endian, it represents 256, but in Little Endian, 00000001 becomes the less significant byte, while 00000000 becomes the most significant byte, and the corresponding number for 256 becomes 00000000 00000001.

https://doi.org/10.15215/remix/9781998944088.01

For more advanced operations on integers, there are some special modules such as the standard math module, math; the free open-source mathematics software system SAGE; (https://www.sagemath.org/); SymPy (https://www.sympy.org/en/index.html); and, for operations in number theory, eulerlib (https://pypi.org/project/eulerlib/).

FLOAT (FLOAT)

Float numbers are numbers with decimals, in the form of 12.5 in decimal notation or 1.25e1 in scientific notation, for example. Operations on float numbers include addition (+), subtraction (–), multiplication (*), division (/), and exponentiation (**), as shown below:

```
In [ ]:   x = 12.5 * 32.7 / 23.9 - 53.6 + 28.9 ** 2.7
          print(f"x = {x}")   # prefix f tells to format the
            string

Out [ ]:  x = 8762.31728619258
```

In the example above, the equal sign (=) is an assignment operator in Python (and almost all other programming languages as well). We will explain all operators fully later in this section.

Python can handle very big integers and floating-point numbers. When a number is too big, it becomes difficult to count and check for accuracy. To solve that problem, Python allows using the underscore to separate the numbers, in the similar way that accounting and finance use a comma. The following is an example:

```
>>> 123_456_789_987.56+234_456_678_789
357913468776.56
```

Using an underscore to separate the digits has made it much easier to tell how big the number is.

R.AS_INTEGER_RATIO()

This returns a pair of integers whose ratio is exactly equal to the original float r and with a positive denominator. It will raise OverflowError on infinities and a ValueError on NaNs.

```
>>> r.as_integer_ratio()
(7093169413108531, 562949953421312)
```

https://doi.org/10.15215/remix/9781998944088.01

R.IS_INTEGER()

This returns True if r is finite with integral value, and False otherwise.

```
>>> r.is_integer()
False
```

R.HEX()

This returns a representation of a floating-point number as a hexadecimal string. For finite floating-point numbers, this representation will always include a leading 0x and a trailing p and exponent.

```
>>> r.hex()
'0x1.9333333333333p+3'
```

FLOAT.FROMHEX(S)

This returns the float represented by a hexadecimal string s. The string s may have leading and trailing whitespace.

```
>>> float.fromhex('0x1.9333333333333p+3')
12.6
```

BOOLEAN (BOOL)

Boolean data have only two values: True and False. They are used to represent the result of a test or an evaluation of logical expressions, as we will see. Technically, Python does not need a special Boolean data type, since it treats 0 and None as Boolean False, and treats everything else as Boolean True, as shown below:

```
In [ ]:  b = None   # assign None to variable b. None in Python
           represents 'no value'
         print(f"b = {b}")   # print out what b holds
         if not b:   # if (not b) is True then print
           print("Print this out when b is None!")

Out [ ]: b = None
         Print this out when b is None!
```

However, having a Boolean data type with two Boolean values of True and False does clearly remind Python programmers, especially beginners, that there are special types of data and expressions called Boolean data and Boolean expressions.

https://doi.org/10.15215/remix/9781998944088.01

COMPLEX (COMPLEX)

If you have never heard about complex numbers, quickly search the internet for complex numbers and read some articles or watch some videos.

Briefly, a complex number is a representation of a point on a plane with X and Y axes that take the form of x + yj, in which x and y are float numbers that represent and define the location of a point on the plane. Examples of complex numbers are 1 + 1j, 3 – 6j, 2.5 – 8.9j, and so on.

The same operations on float numbers can also be applied to complex numbers, as shown below:

```
In [ ]:  x = 3.5 + 6.7j   # assign a complex number to x
         y = 12.3 - 23.9j   # assign another complex number to y
         z = x + y   # assign the sum of x and y to z
         print(x, '+', y, '=', z)   # print out the value of the
           variable
         print(x, '-', y, '=', x - y)   # the difference between
           x and y
         print(x, '*', y, '=', x * y)   # the product of x and y
         print(x, '/', y, '=', x / y)   # the result of x
           divided by y

Out [ ]: (3.5 + 6.7j) + (12.3 - 23.9j) = (15.8 - 17.2j) (3.5 + 6.7j) - (12.3 - 23.9j)
           = (-8.8 + 30.599999999999998j) (3.5 + 6.7j) * (12.3 - 23.9j) =
           (203.18 - 1.2399999999999807j) (3.5 + 6.7j) / (12.3 - 23.9j) =
           (-0.16204844290657439 + 0.229840830449827j)
```

Compound Data Types

In previous sections, we saw data in pieces. Sometimes it is more convenient, more effective, and even necessary to use some data together to represent certain kinds of information. Examples are when we refer to the days of the week, months of the year, and so on. Courses in universities are often identified using compound data, a tuple made of a course number and course title or name.

In Python, compound data types provide means of structuring data. They are also referred to as data structures.

Compound data types in Python can be categorized into sequence and nonsequence data types. Items in sequence compound data are ordered and indexed. Sequence compound data types include *string*, *list*, and *tuple*. Items in a nonsequence compound data are not ordered. Nonsequence compound data include *set* and *dictionary*.

https://doi.org/10.15215/remix/9781998944088.01

STRING (STR)

Sequences are a group of data in order, and a string is a good example of a sequence.

Like numbers, strings are very important in all programming languages. It is hard to imagine what a number means without its context. For that reason, in most programming languages, strings are also considered a primary data type in terms of their importance and the role that they play.

In Python, strings are sequences of characters, symbols, and numbers enclosed in a pair of double quotation marks or a pair of single quotation marks. Table 2-2 is a list of ASCII characters that can be used in strings. The following are some examples of strings:

```
In [ ]:  s1 = "this is my first string in double quotes"
         s2 = 'string in single quotes'
         print('s1 = ', s1)   # print out s1 = the value of
          variable s1
         print('s2 = ', s2)   # print out s2 = the value of
          variable s2

Out [ ]: s1 = this is my first string in double quotes
         s2 = string in single quotes
```

When a string is too long and needs to span more than one line, a pair of triple quotation marks can be used, as shown in the following example:

```
In [ ]:  long_string = """ASCII stands for American Standard
          Code for Information Interchange.
         Computers can only understand numbers, so an ASCII code
          is the numerical representation of a character such
          as "a" or "@" or an action of some sort.
         ASCII was developed a long time ago, and now the
          nonprinting characters are rarely used for their
          original purpose. Below is the ASCII character table.
          The first 32 characters are nonprinting characters.
          ASCII was designed for use with teletypes, so the
          descriptions in ASCII are somewhat obscure.
         If someone says they want your CV in ASCII format,
          all this means is they want "plain" text with no
          formatting such as tabs, bold or underscoring—the raw
          format that any computer can understand.
         This is usually so they can easily import the file into
          their own applications without issues. Notepad.exe
          creates ASCII text, and MS Word lets you save a file
          as "text only."
         """
         print('long_string =', long_string)
```

https://doi.org/10.15215/remix/9781998944088.01

```
Out [ ]:  long_string = ASCII stands for American Standard Code for Information
            Interchange. Computers can only understand numbers, so an ASCII code
            is the numerical representation of a character such as "a" or "@" or an
            action of some sort. ASCII was developed a long time ago and now the
            nonprinting characters are rarely used for their original purpose. Below
            is the ASCII character table. The first 32 characters are nonprinting
            characters. ASCII was designed for use with teletypes and so the
            descriptions in ASCII are somewhat obscure. If someone says they want
            your CV in ASCII format, all this means is they want "plain" text with no
            formatting such as tabs, bold or underscoring—the raw format that any
            computer can understand. This is usually so they can easily import the
            file into their own applications without issues. Notepad.exe creates
            ASCII text, and MS Word lets you save a file as "text only."
```

This can be very useful in cases such as when you want to print out instructions for users to use with an application you developed.

Otherwise, you would need to use backslash at the end of each line except the last one to escape the invisible newline ASCII character, as shown below:

```
In [ ]:  s0 = 'Demo only. This string is not too long \
         to be put on one line.'

         print(f's0 = {s0}')
```

```
Out [ ]:  s0 = Demo only. This string is not too long to be put on one line.
```

This is OK if the string only spans across two or three lines. It will look clumsy if the string spans across a dozen lines.

In the example above, we use backslash \ to escape or cancel the invisible newline ASCII character. In Python and almost all programming languages, some characters have special meanings, or we may want to assign special meaning to a character. To include such a character in a string, you need to use a backslash to escape from its original meaning. The following are some examples:

```
In [ ]:  print("This string will be put \n on two lines")
         # \n will start a new line
         print("This string will add \t a tab - a big space")
         # \t will add a tab - big space
```

```
Out [ ]:  This string will be put
          on two lines
          This string will add    a tab - a big space
```

https://doi.org/10.15215/remix/9781998944088.01

In the examples above, putting a backslash in front of n assigns the combination \n a special meaning, which is to add a new line to the string; putting a backslash in front of t assigns the combination \t a special meaning, which is to add a tab (a number of whitespaces) to the string. The next example uses backslash to escape the quotation from its original meaning defined in Python.

```
In [ ]:  print("This is \"President\" quoted")
         # \" puts quotation marks in a string
```

```
Out [ ]: This is "President" quoted
```

Table 2-2: ASCII table showing the codes of characters

Dec	Char	Dec	Char	Dec	Char	Dec	Char
0	**NUL (null)**	32	SPACE	64	@	96	`
1	**SOH (start of heading)**	33	!	65	A	97	a
2	**STX (start of text)**	34	"	66	B	98	b
3	**ETX (end of text)**	35	#	67	C	99	c
4	**EOT (end of transmission)**	36	$	68	D	100	d
5	**ENQ (enquiry)**	37	%	69	E	101	e
6	**ACK (acknowledge)**	38	&	70	F	102	f
7	**BEL (bell)**	39	'	71	G	103	g
8	**BS (backspace)**	40	(	72	v	104	h
9	**TAB (horizontal tab)**	41	)	73	I	105	i
10	**LF (NL line feed, new line)**	42	*	74	J	106	j
11	**VT (vertical tab)**	43	+	75	K	107	k
12	**FF (NP form feed, new page)**	44	,	76	L	108	l
13	**CR (carriage return)**	45	-	77	M	109	m
14	**SO (shift out)**	46	.	78	N	110	n

https://doi.org/10.15215/remix/9781998944088.01

Table 2-2: ASCII table showing the codes of characters *(continued)*

Dec	Char	Dec	Char	Dec	Char	Dec	Char
15	**SI (shift in)**	47	/	79	O	111	o
16	**DLE (data link escape)**	48	0	80	P	112	p
17	**DC1 (device control 1)**	49	1	81	Q	113	q
18	**DC2 (device control 2)**	50	2	82	R	114	r
19	**DC3 (device control 3)**	51	3	83	S	115	s
20	**DC4 (device control 4)**	52	4	84	T	116	t
21	**NAK (negative acknowledge)**	53	5	85	U	117	u
22	**SYN (synchronous idle)**	54	6	86	V	118	v
23	**ETB (end of trans. block)**	55	7	87	W	119	w
24	**CAN (cancel)**	56	8	88	X	120	x
25	**EM (end of medium)**	57	9	89	Y	121	y
26	**SUB (substitute)**	58	:	90	Z	122	z
27	**ESC (escape)**	59	;	91	[	123	{
28	**FS (file separator)**	60	<	92	\	124	|
29	**GS (group separator)**	61	=	93	]	125	}
30	**RS (record separator)**	62	>	94	^	126	~
31	**US (unit separator)**	63	?	95	_	127	DEL

These 128 ASCII characters, including both printable and unprintable ones, are defined for communication in English between humans and machines. There is an extended set of ASCII characters defined for communication in other Western languages such as German, French, and others.

To enable communication between human and machines in languages such as Chinese, Unicode was designed. Details about Unicode can be found at https://unicode.org/. For information on how Unicode is used in Python, read the article at https://docs.python.org/3/howto/unicode.html.

https://doi.org/10.15215/remix/9781998944088.01

In earlier versions of Python, if you want to use a non-ASCII character encoded in Unicode in a string, you need to know the code, assuming it is NNNN, and use escape sequence \uNNNN within the string to represent the non-ASCII character. You can also use built-in function chr(M) to get a one-character string encoded in Unicode, where M is code of the character in the Unicode table. The reverse built-in function order(Unicode character) is used to get the code of the Unicode character in the Unicode table.

In Python 3.0, however, the default encoding of Python programs was changed to UTF-8, which includes Unicode, so you can simply include any Unicode character in a string and PVM will recognize and handle it correctly, as shown in the following example:

```
In [ ]:  print("秦时明月汉时关 can be directly included in a
           string")
         print(f"though you can still use chr(31206) for
           {chr(31206)}, chr(27721) for {chr(27721)}")

Out [ ]: 秦时明月汉时关 can be directly included in a string
         though you can still use chr(31206) for 秦, chr(27721) for 汉
```

When representing strings, some prefixes or flags can be put in front of the opening quotation mark. These flags are also called prefixes, used before the opening quote of a string. These prefixes are listed in Table 2-3 with their meaning and some coding samples.

Table 2-3: Prefixes used for string formatting and construction

Flag	What it does	Code sample in Jupyter Notebook

F, f	F/f for formatting. Causes the evaluation of expressions enclosed within {}.	In:
		`name="John"`
		`s0 = f"Your name is {name}."`
		`print(s0)`
		Out:
		Your name is John.

R, r	R/r for raw string. Nothing in the string is evaluated, not even \".	In:
		`name = "John"`
		`s0 = r"Your name is\t {name}`
		`  \""`
		`print(s0)`
		Out:
		Your name is \t {name} \"

https://doi.org/10.15215/remix/9781998944088.01

Table 2-3: Prefixes used for string formatting and construction *(continued)*

Flag	What it does	Code sample in Jupyter Notebook

U, u	U/u for Unicode, indicating Unicode literals in a string. It has no effect in Python 3 since Python 3's default coding is UTF-8, which includes Unicode.	In:
		`print(u"秦时明月汉时关", end=":")`
		`print("秦时明月汉时关")`
		Out:
		秦时明月汉时关:秦时明月汉时关

B, b	B/b for byte. Literals in the string become byte literals, and anything outside of ASCII table must be escaped with backslash.	In: `print(b"2005-05-26-10458.68")` Out: b"2005-05-26-10458.68"

LIST

List is a very useful compound data type built into Python. A list of elements, which can be different types of data, is enclosed in square brackets. The following are some examples of lists:

```
[1, 2, 3, 4, 5]
['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday']
```

In Python, an element in a list can be any type of data or object, to use a more common computer-science term. So an element can be a list too, such as,

```
[[1,2,3],[4, 5, 6]]
```

Assume that

```
week = ['Monday', 'Tuesday', 'Wednesday', 'Thursday',
'Friday', 'Saturday', 'Sunday']
```

The term week[0] refers to the first element of the list, and week[1] refers to the second element of the list, where 0 and 1 are called index. An index can be negative as well, meaning an item counted from the end of the list. For example, week[-1] will be Sunday, the first item from the end; week[-2] will be Saturday, the second item from the end.

https://doi.org/10.15215/remix/9781998944088.01

To get a sublist of a list L we use notation L[s: e], where s is the starting point in the list, e is the ending point within the list, and the sublist will include all items from s till e but excluding the item at e. For example, to get all weekdays from the list week, we use week[0,5], as shown below:

```
In [ ]:  week = ['Monday', 'Tuesday', 'Wednesday', 'Thursday',
           'Friday', 'Saturday', 'Sunday']
         week[0:5]

Out [ ]: ['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday']
```

Negative indexing becomes convenient when a list, or any sequence, is too long to count from the beginning. In the example above, it is much easier to count from the end to find out that the weekdays stop right before Saturday, whose index is −2, as shown below:

```
In [ ]:  week = ['Monday', 'Tuesday', 'Wednesday', 'Thursday',
           'Friday', 'Saturday', 'Sunday']
         week[0:-2]

Out [ ]: ['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday']
```

Within the notation of L[s:e], s or e or both may be missing. When s is missed, it means the sublist is indexed from the beginning of the list; when e is missed, it means the sublist is indexed from the end of the list. So week[:] will include all the items of the list week, as shown below:

```
In [ ]:  week[:]

Out [ ]: ['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday', 'Sunday']
```

Multiple lists can be joined together with operator +. Assume we have the following two lists:

```
weekdays = ['Monday', 'Tuesday', 'Wednesday', 'Thursday',
'Friday']
weekend = ['Saturday', 'Sunday']
```

We can then combine weekdays and weekend into one list and assign the new list to week using operator +, as shown below:

https://doi.org/10.15215/remix/9781998944088.01

```
In [ ]:  weekdays = ['Monday', 'Tuesday', 'Wednesday',
           'Thursday', 'Friday']
         weekend = ['Saturday', 'Sunday']
         week = weekdays + weekend
         print(week)

Out [ ]: ['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday', 'Sunday']
```

A copy of a list can be easily made by sublisting all its members, as shown below:

```
week0 = week[:]
```

We can create a list from a string using the built-in function list():

```
In [ ]:  l0 = list("How are you?")
         print(l0)

Out [ ]: ['H', 'o', 'w', ' ', 'a', 'r', 'e', ' ', 'y', 'o', 'u', '?']
```

We can also create a list using the built-in function list() and range():

```
In [ ]:  l1 = list(range(10))
         l2 = list(range(10, 20))
         print(l1, l2)

Out [ ]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9] [10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
```

We can use the built-in function len() to find out how many elements are in the list:

```
In [ ]:  print(len(week))

Out [ ]: 4
```

TUPLE

Tuple is another type of sequence, but members of a tuple are enclosed in parentheses. The following are some sample tuples:

```
(12, 35)
('Canada', 'Ottawa')
('China', 'Beijing')
```

https://doi.org/10.15215/remix/9781998944088.01

You can create a tuple from a list by using the tuple() function. For example,

TPL = TUPLE(WEEK)

This will create a tuple, as shown below:

```
('Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday',
'Saturday', 'Sunday')
```

Similarly, you can create a list from a tuple, as shown below:

```
week = list(tpl)
```

Members of a tuple can be accessed in the same way as lists because the members are also indexed. For example, tpl[0] refers to Monday.

Moreover, most of the operations used on lists can be applied to tuples, except those that make changes to the member items, because tuples are immutable.

Why are tuples immutable? You may consider it like this: A tuple is used to represent a specific object, such as a point on a line, or even a shape. If any member of the tuple is changed, it will refer to a different object. This is the same for numbers and strings, which are also immutable. If you change any digit of a number, or any character of a string, the number or string will be different.

SET

In Python, a set is a collection of elements or objects enclosed in a pair of curly brackets. Like sets in mathematics, members in a set are unordered and unindexed. The following are two examples of sets:

```
grades = {'A', 'A+', 'A-', 'B', 'B+', 'B-', 'C', 'C+',
'C-', 'D', 'D+', 'D-'}
My_friends = {'John', 'Jack', 'Jim', 'Jerry', 'Jeromy'}
```

You can use built-in function set() to build a set from a list or tuple:

```
week_set = set(('Monday', 'Tuesday', 'Wednesday',
'Thursday', 'Friday', 'Saturday', 'Sunday'))
```

Built-in functions set(), list(), tuple(), float(), int(), str() are also called constructors or converters, because they are used to construct or convert to one respective type of data from another type.

https://doi.org/10.15215/remix/9781998944088.01

You can use membership operator ***in*** to test if an object is a member of a set. For example,

```
'John' in My_friends
```

will give you a True value because John is a member of set My_friends constructed above.

You can use built-in function len() to get the size of a set—that is, how many members are in the set. So len(grades) will be 12.

DICTIONARY

In Python, a dictionary is a collection of comma-separated key-value pairs enclosed in curly brackets and separated by a colon :. The members of a dictionary are unordered and unindexed. The following is an example:

```
In [ ]:  weekday = {'Mon':'Monday', 'Tue':'Tuesday',
           'Wed':'Wednesday', 'Thu':'Thursday', 'Fri':'Friday'}
         print(weekday['Mon'], ' of ', weekday)

Out [ ]: Monday of {'Mon': 'Monday', 'Tue': 'Tuesday', 'Wed': 'Wednesday', 'Thu':
           'Thursday', 'Fri': 'Friday'}
```

Because the keys are used to retrieve the values, each key must be unique within a dictionary. For example, we use Weekday['Mon'] to retrieve its corresponding value, Monday.

In this case, you can also use integer numbers as keys, as shown below:

```
In [ ]:  weekday = {1:'Monday', 2:'Tuesday', 3:'Wednesday',
           4:'Thursday', 5:'Friday'}
         print(weekday[1], ' of ', weekday)

Out [ ]: Monday of {1: 'Monday', 2: 'Tuesday', 3: 'Wednesday', 4: 'Thursday', 5:
           'Friday'}
```

OBJECT

Although you can code almost any application in Python without object-oriented thinking, you should be aware that Python fully supports object-oriented programming. In fact, Python treats everything as an object, including classes, functions, and modules. For example, numbers are treated as objects in the following statements:

https://doi.org/10.15215/remix/9781998944088.01

```
In [ ]:  print((2.35).hex())
         print((23).bit_length())

Out [ ]: 0x1.2cccccccccccdp+1
         5
```

There are some special types of objects in Python, as shown in Table 2-4.

Table 2-4: Types of objects in Python

Name	Meaning	Example of usage
type	Because in Python everything is an object, so are classes, whose class is type, which is again an object as well as a special object.	It has no real usage in programming but is rather a philosophical and ideological concept.

None	An object representing "no value." It is the only object of data type NoneType. For a logical expression, None is the same as 0, null, empty string, and False.	In:
		`if not None:`
		`  print('there is no value')`
		Out:
		There is no value

file	A file object can be created by calling the file() or open() methods, which allow us to use, access, and manipulate all accessible files.	f = open("resume.xml", "r")
function	In Python, functions are first-class objects and can be passed as arguments in another function call.	sort(method, data_list)
module	A special type of object that all modules belong to.	It is more a philosophical and ideological concept.
class	A type of all classes.	It is more a philosophical and ideological concept.
class Instance	An individual object of a given class.	It is more a philosophical and ideological concept.
method	A method of a given class.	It is more a philosophical and ideological concept.
code	A code object is the internal representation of a piece of runnable Python code.	It is used in internal Python program running.

https://doi.org/10.15215/remix/9781998944088.01

Table 2-4: Types of objects in Python *(continued)*

Name	Meaning	Example of usage
frame	A table containing all the local variables.	It is used in function calling.
traceback	A special object for exception handling.	It provides an interface to extract, format, and print stack traces of Python programs, especially when an exception is raised.
ellipsis	A single object called ellipsis, a built-in name in Python 3.	It is rarely used except in slicing.

Variables and Constants

Data values, variables, and constants are important constructs of any programming language. We have presented all standard simple or primary data types and compound data types and their values in the previous section. In the following section, we will study variables and constants.

VARIABLES

A variable is a name that identifies a location in computer memory to store data. A variable must conform to the rules of naming discussed earlier in this section, and it also must not be used for other purposes within the same context of the program. For example, you cannot use reserved words or keywords as variables.

An important operation on a variable is to assign a value to it or put a value into the memory location identified by the variable. In Python, this is done by using the assignment operator =.

For example, the following statement assigns COMP 218 to the variable course, assigns Smith to the variable student, and assigns 99 to the variable grade:

```
In [ ]:  course = 'COMP 218'
         student = 'Smith'
         grade = 99
         print("course is {}, student is {}, grade is {}".
           format(course, student, grade))
```

```
Out [ ]: course is COMP 218, student is Smith, grade is 99
```

https://doi.org/10.15215/remix/9781998944088.01

Unlike other programming languages, in Python, there is no need to declare its type before a variable is introduced and used in your program for the first time. Its type is determined by the value assigned to it. In the above example, the type of grade is integer, because 99 is an integer.

```
In [ ]:  type(grade)

Out [ ]: int
```

You may convert the value of a variable into another type using specific built-in functions, such as int(), float(), str(), etc.

```
In [ ]:  marks = 90.8   # assign float mark to variable marks
         int_mark = int(marks)   # convert into integer mark and
           assign it to variable int_mark
         print(int_mark)

Out [ ]: 90
```

CODING PRACTICE

Task 1

In a Jupyter Notebook cell, type the following code and run it by hitting Shift+Enter to see what you will get:

```
pi = 3.1415
r = 6
area = pi * r ** 2
print(F"The type of pi is {type(pi)}, and its value is
{pi}")
print("The type of r is ", type(r), ", its value is ", r)
print("The type of area is {:^10}".
format(str(type(area))), end='')
print(", and its value is {:^12}.".format(area))
```

Task 2

In a new cell of Jupyter Notebook, write and run the following statements, which calculate the circumference of a circle with radius r = 10, and print the result:

https://doi.org/10.15215/remix/9781998944088.01

```
pi = 3.1415
r = 10
circumference = 2 * pi * r
print("The circumference of a circle with radius {:d} is
{:f}".format(r, circumference))
```

Please note that in Jupyter Notebook, if you need a new cell, simply click the plus sign button under the notebook menu.

Please also note that the type of variable area is the same as the type of variable pi, but not of variable r. We can check what type the result will be when a different arithmetic operator is applied to a pair of numbers of the same or different types, as shown in Tables 2-5a and 2-5b:

Table 2-5a: Semantics of operator +, -, *

+, -, *	integer	float	complex
integer	integer	float	complex
float	float	float	complex
complex	complex	complex	complex

Table 2-5b: Semantics of / (division) operator

/	integer	float	complex
integer	float	float	complex
float	float	float	complex
complex	complex	complex	complex

The following coding example in Jupyter Notebook confirms that the result will be a complex number when an integer is divided by a complex number:

```
In [ ]:  i = 12
         cx = 23 + 35j
         icx = i / cx
         print(icx, type(icx))

Out [ ]: (0.1573546180159635-0.2394526795895097j) <class 'complex'>
```

You may copy and modify the code above to check other combinations in Jupyter Notebook.

https://doi.org/10.15215/remix/9781998944088.01

BUILT-IN CONSTANTS

We have seen some values of various data types. Some values have special meanings in Python. We call them constants. The following table lists all the constants you may see and use when programming with Python.

Table 2-6: Python built-in constants

Constant name	Meaning	Code sample in Python interactive mode

True	Logical true	`>>> x = True`
		`>>> if x:`
		`… print(f"x is {x}")`
		`…`
		`x is True`

False	Logical false	`>>> x = False`
		`>>> if not x:`
		`… print(f"x is {x}")`
		`…`
		`x is False`

None	No value assigned	`>>> x = None`
		`>>> if not x:`
		`… print(f"{x} is treated as False")`
		`…`
		`None is treated as False`

Ellipsis	Same as …, often used in slicing multiple dimensional arrays	`>>> Ellipsis == …`
		`True`
		`>>> Ellipsis is …`

__debug__	Contains True if Python is started without the -O option	`C:\Users\comp218>python -O`
		`Python 3.7.2`
		`>>> __debug__`
		`False`

quit	Contains information displayed when Python quits and terminates	`>>> quit` `Use quit() or Ctrl+Z+Enter to exit`
exit	Contains information displayed when Python quits and terminates	`>>> exit` `Use exit() or Ctrl+Z+Enter to exit`

https://doi.org/10.15215/remix/9781998944088.01

Table 2-6: Python built-in constants *(continued)*

Constant name	Meaning	Code sample in Python interactive mode

copyright	Contains copyright information	`>>> copyright`
		`Copyright (c) 2001-2018`
		`  Python Software`
		`  Foundation.`
		`All Rights Reserved.`
		`Copyright (c) 2000`
		`  BeOpen.com.`
		`All Rights Reserved.`
		`Copyright (c) 1995-2001`
		`  Corporation for`
		`  National Research`
		`  Initiatives.`
		`All Rights Reserved.`
		`Copyright (c) 1991-1995`
		`  Stichting Mathematisch`
		`  Centrum, Amsterdam.`
		`All Rights Reserved.`

credits	Contains credit information	`>>> credits`
		`Thanks to CWI, CNRI,`
		`  BeOpen.com, Zope`
		`  Corporation and a`
		`  cast of thousands`
		`  for supporting Python`
		`  development. See`
		`  www.python.org for more`
		`  information.`

license	Contains licensing information	`>>> license`
		`Type license() to see the`
		`  full license text`

__name__	When a Python file is started, some special variables are initialized with specific values, and __name__ is one such variable. This variable will have value __main__ if the Python file is started as the main program; otherwise, it will contain the module name or the function or class name if that is imported from the module.	`C:\comp218> python prime.`
		`  py`
		`....`
		`>>> __name__`
		`__main__`

More information about these constants can be found at https://docs.python.org/3/library/constants.html.

https://doi.org/10.15215/remix/9781998944088.01

Operators

As we learned in previous sections, data are used to represent information, while variables and constants are used to refer to data stored in computer memory. In the following sections, we will learn about operators and built-in functions that can be used to process and manipulate data.

ARITHMETIC OPERATORS

Arithmetic operators are used on numbers. These operators are well-known. Table 2-7 provides a list of these operators, their meaning, and code samples you may take and practise in Python interactive mode. Please copy only the expressions or statements behind >>>, which is the Python prompt for your input.

Table 2-7: Arithmetic operators

Operator	Operation	Code samples in Python interactive mode

+	Add two operands
	or unary plus
	+ operator can be redefined in a class for its objects defining __add__
		`>>> x = 10`
		`>>> y = 20`
		`>>> x + y`
		`30`
		`>>> +y`
		`20`

-	Subtract right operand from the left or unary minus	`>>> x, y = 10, 20`
		`>>> x - y`
		`-10`
		`>>> -y`
		`-20`

*	Multiply two operands	`>>> x, y = 10, 20`
		`>>> y * x`
		`200`

/	Divide left operand by the right one (always results into float)	`>>> x, y = 10, 20`
		`>>> y / x`
		`2.0`

//	Floor division—division that results into integer number by omitting all the decimals	`>>> x, y = 32, 7`
		`>>> x, y`
		`(32, 7)`
		`>>> x // y`
		`4`

%	Modulus—remainder of the division of left operand by the right	`>>> x, y = 32, 7`
		`>>> x, y`
		`(32, 7)`
		`>>> x % y`
		`4`

https://doi.org/10.15215/remix/9781998944088.01

Table 2-7: Arithmetic operators *(continued)*

Operator	Operation	Code samples in Python interactive mode

**	Exponent—left operand raised to the power of right	`>>> x, y = 32, 7`
		`>>> x, y`
		`(32, 7)`
		`>>> x ** y`
		`34359738368`

When two or more of these arithmetic operators are used in an expression, the precedence rules you learned in high school or university math courses apply. In brief, the precedence rules for all the operators are as follows:

1. Exponent operation (*) has the highest precedence.
2. Unary negation (–) is the next.
3. Multiplication (*), division (/), floor division (//), and modulus operation (%) have the same precedence and will be evaluated next unary negation.
4. Addition (+) and subtraction (–) are next, with the same precedence.
5. Comparison operators are next, with the same precedence.
6. The three logical operators (*not*, *and*, *or*) are next, with *not* having the highest precedence among the three, followed by *and*, then *or*.
7. When operators with equal precedence are present, the expression will be evaluated from left to right, hence left association.
8. Parentheses can be used to change the order of evaluation.

The following are some examples of how expressions are evaluated.

```
>>> 12 + 3 * 21
75
```

In the expression 12 + 3 * 21, because * has higher precedence than +, 3 * 21 is evaluated first to get 63, and then 12 + 63 is evaluated to get 75.

```
>>> ((23 + 32) // 6 - 5 * 7 / 10) * 2 ** 3
44.0
```

In this example, because ** has the highest precedence, 2 ** 3 is evaluated first, to get

https://doi.org/10.15215/remix/9781998944088.01

```
((23 + 32) // 6 - 5 * 7 / 10) * 8
```

In this intermediate result, because of parentheses, we will need to evaluate ((23 + 32) // 6 – 5 * 7 / 10) first, and in this subexpression, because of parentheses, 23 + 32 will be evaluated first to get 55//6 – 5 * 7 /10. According to the precedence rules, this subexpression will be evaluated to 9 – 35 / 10, and then 9 – 3.5 = 5.5. Then the expression above will be

```
5.5 * 8
```

which is evaluated to be 44.0.

COMPARISON OPERATORS

Comparison operators are used to compare two objects. It is easy to understand how they work on numbers and even strings. When a comparison is applied to lists or tuples, the comparison will be applied to pairs of items, one from each list or tuple, and the final result is True if there are more Trues; otherwise it will be False, as shown in the example below:

```
In [ ]:  l1 = ['apple', 'orange', 'peach']
         l2 = ['tomato', 'pepper', 'cabbage']
         l1 < l2

Out [ ]: True
```

Table 2-8 explains all the comparison operators, with samples.

Table 2-8: Python comparison operators

Operator	Operation	Code sample in Python interactive mode

>	Greater than—True if left operand is greater than the right	`>>> x, y = 23, 53`
		`>>> x > y`
		`False`
		`>>> l1 = ('apple',`
		`'orange', 'peach')`
		`>>> l2 = ('tomato',`
		`'pepper', 'cabbage')`
		`>>> l2 > l1`
		`True`

https://doi.org/10.15215/remix/9781998944088.01

Table 2-8: Python comparison operators *(continued)*

Operator	Operation	Code sample in Python interactive mode

<	Less than—True if left operand is less than the right	`>>> x, y = 23, 53`
		`>>> x, y`
		`(23, 53)`
		`>>> x < y`
		`True`
		`>>> s1 = "Albert"`
		`>>> s2 = "Jeremy"`
		`>>> s1 < s2`
		`True`

==	Equal to—True if both operands are equal	`>>> x, y = 23, 53`
		`>>> x, y`
		`(23, 53)`
		`>>> x == y`
		`False`

!=	Not equal to—True if operands are not equal	`>>> x, y = 23, 53`
		`>>> x, y`
		`(23, 53)`
		`>>> x != y`
		`True`

>=	Greater than or equal to—True if left operand is greater than or equal to the right	`>>> x, y = 23, 53`
		`>>> x, y`
		`(23, 53)`
		`>>> x >= y`
		`True`

<=	Less than or equal to—True if left operand is less than or equal to the right	`>>> x, y = 23, 53`
		`>>> x, y`
		`(23, 53)`
		`>>> x <= y`
		`True`

LOGICAL OPERATORS

Logical operators are used to form logical expressions. Any expression whose value is a Boolean True or False is a logical expression. These will include expressions made of comparison operators discussed above. Table 2-9 summarize the details of these logical variables.

https://doi.org/10.15215/remix/9781998944088.01

Table 2-9: Python logical operators

Operator	Meaning	Code sample in Python interactive mode

and	A and B
	True if both A and B are true
		`>>> (x, y) = (23, 53)`
		`>>> (m, n) = (12, 9.6)`
		`>>> x, y, m, n`
		`(23, 53, 12, 9.6)`
		`>>> x > y and m > n`
		`False`

or	A or B
	True if either A or B is true
		`>>> (x, y) = (23, 53)`
		`>>> (m, n) = (12, 9.6)`
		`>>> x, y, m, n`
		`(23, 53, 12, 9.6)`
		`>>> x > y or m > n`
		`True`

not	not A
	True if A is false
		`>>> (x, y) = (23, 53)`
		`>>> x, y`
		`(23, 53)`
		`>>> not x < y`
		`False`

Logical expressions are often used in ***if*** and ***while*** statements, and it is important to ensure that the logical expressions used in your programs are correctly written. Otherwise, catastrophic results may occur in some real applications. Common errors in writing logical expressions include:

1. Using > instead of <, or using < instead of >
2. Using >= instead of >, or using > instead >=
3. Using <= instead of <, or using < instead <=

For example, suppose you are writing a program to control the furnace at home, and you want to heat the home to 25 degrees Celsius. The code for this should be

```
if t < 25 : heating()
```

However, if instead you wrote,

```
if t > 25 : heating()
```

the consequence would be either the home will not heat at all (if the initial temperature is below 25 when the program starts) or it will overheat (if the initial temperature is greater than 25).

https://doi.org/10.15215/remix/9781998944088.01

BITWISE OPERATORS

We know that data in computer memory are represented as sequences of bits, which are either 1 or 0. Bitwise operators are used to operate bit sequences bit by bit. These bitwise operations may look strange to you, but you will appreciate these operations when you need them. Table 2-10 provides a summary of bitwise operators. Please note that built-in function ***bin()*** converts data into their binary equivalents and returns a string of their binary expressions, with a leading 0b.

Table 2-10: Python bitwise operators

Operator	Meaning	Code sample in Python interactive mode

&	Bitwise *and*	`>>> m, n = 12, 15`
		`>>> bin(m), bin(n)`
		`('0b1100', '0b1111')`
		`>>> bin(m & n)`
		`'0b1100'`

|	Bitwise *or*	`>>> bin(m), bin(n)`
		`('0b1100', '0b1111')`
		`>>> bin(m | n)`
		`'0b1111'`

~	Bitwise *not*	`>>> ~ m`
		`-13`
		`>>> bin(~ m)`
		`'-0b1101'`

^	Bitwise *XOR*	`>>> bin(m), bin(n)`
		`('0b1100', '0b1111')`
		`>>> bin(m ^ n)`
		`'0b11'`

>>	Bitwise right shift	`>>> bin(m)`
		`'0b1100'`
		`>>> bin(m>>2)`
		`'0b11'`

<<	Bitwise left shift	`>>> bin(m)`
		`'0b1100'`
		`>>> bin(m<<2)`
		`'0b110000'`

https://doi.org/10.15215/remix/9781998944088.01

ASSIGNMENT OPERATORS

In Python, and all programming languages, the assignment operation is one of the most important operations because assignment operations store data in variables for later use.

In previous sections, we have seen many examples of using the assignment operator =, the equal sign. However, Python provides many augmented assignment operators. Table 2-11 lists all the assignment operators you can use in your Python programs.

Table 2-11: Python assignment operators

Operator	Operation	Code samples in Python interactive mode

x = e	Expression e is evaluated, and the value is assigned to variable x. Note that in Python and in previous code samples, assignments can be made to more than one variable with one statement.	`>>> m, n = 23, 12`
		`>>> m, n`
		`(23, 12)`
		`>>> x = m + n`
		`>>> x`
		`35`

x += e	Expression x + e is evaluated, and the value is assigned to variable x.
	x = x + e
		`>>> x, m, n = 35, 23, 12`
		`>>> x, m, n`
		`(35, 23, 12)`
		`>>> x += m + n`
		`>>> x`
		`70`
		`>>> x, m, n`
		`(70, 23, 12)`

x = e	Expression x − e is evaluated, and the value is assigned to variable x.
	x = x − e
		`>>> x, m, n = (70, 23, 12)`
		`>>> x, m, n`
		`(70, 23, 12)`
		`>>> x −= m + n`
		`>>> x, m, n`
		`(35, 23, 12)`

x *= e	Expression x * e is evaluated, and the value is assigned to variable x.
	x = x * e
		`>>> x, m, n = (35, 23, 12)`
		`>>> x, m, n`
		`(35, 23, 12)`
		`>>> x *= m + n`
		`>>> x, m, n`
		`(1225, 23, 12)`

https://doi.org/10.15215/remix/9781998944088.01

Table 2-11: Python assignment operators *(continued)*

Operator	Operation	Code samples in Python interactive mode

x /= e	Expression x / e is evaluated, and the value is assigned to variable x.
	x = x / e
		`>>> x, m, n = (1225, 23, 12)`
		`>>> x, m, n`
		`(1225, 23, 12)`
		`>>> x /= m + n`
		`>>> x, m, n`
		`(35.0, 23, 12)`

x %= e	Expression x % e is evaluated, and the value is assigned to variable x.
	x = x % e
		`>>> x, m, n = (28, 23, 12)`
		`>>> x, m, n`
		`(28, 23, 12)`
		`>>> x %= m + n`
		`>>> x, m, n`
		`(28, 23, 12)`

x //= e	Expression x // e is evaluated, and the value is assigned to variable x.
	x = x // e
		`>>> x, m, n = 989, 23, 12`
		`>>> x, m, n`
		`(989, 23, 12)`
		`>>> x //= m + n`
		`>>> x, m, n`
		`(28, 23, 12)`

x **= e	Expression x ** e is evaluated, and the value is assigned to variable x.
	x = x ** e
		`>>> x, m, n = 98, 3, 2`
		`>>> x **= m + n`
		`>>> x, m, n`
		`(9039207968, 3, 2)`

x &= e	Expression x & e is evaluated, and the value is assigned to variable x.
	x = x & e
		`>>> x, m, n = 9, 3, 2`
		`>>> bin(x), bin(m), bin(n)`
		`('0b1001', '0b11', '0b10')`
		`>>> x &= m * n`
		`>>> bin(x), bin(m), bin(n)`
		`('0b0', '0b11', '0b10')`

x |= e	Expression x | e is evaluated, and the value is assigned to variable x.
	x = x | e
		`>>> x, m, n = 9, 3, 2`
		`>>> bin(x), bin(m), bin(n)`
		`('0b1001', '0b11', '0b10')`
		`>>> x |= m * n`
		`>>> bin(x), bin(m), bin(n)`
		`('0b1111', '0b11', '0b10')`

(continued on next page)

https://doi.org/10.15215/remix/9781998944088.01

Table 2-11: Python assignment operators *(continued)*

Operator	Operation	Code samples in Python interactive mode

x ^= e	Expression x ^ e is evaluated, and the value is assigned to variable x.
	x = x ^ e
		`>>> x, m, n = 9, 3, 2`
		`>>> bin(x), bin(m), bin(n)`
		`('0b1001', '0b11', '0b10')`
		`>>> x ^= m * n`
		`>>> bin(x), bin(m), bin(n)`
		`('0b1111', '0b11', '0b10')`

x >>= e	Expression x >> e is evaluated, and the value is assigned to variable x.
	x = x >> e
		`>>> x, m, n = 9, 3, 2`
		`>>> bin(x), bin(m), bin(n)`
		`('0b1001', '0b11', '0b10')`
		`>>> x >>= m * n`
		`>>> bin(x), bin(m), bin(n)`
		`('0b0', '0b11', '0b10')`

x <<= e	Expression x << e is evaluated, and the value is assigned to variable x.
	x = x << e
		`>>> x, m, n = 9, 3, 2`
		`>>> bin(x), bin(m), bin(n)`
		`('0b1001', '0b11', '0b10')`
		`>>> x <<= m * n`
		`>>> bin(x), bin(m), bin(n)`
		`('0b1001000000', '0b11', '0b10')`

CODING PRACTICE

Type the following code to a scratch file in VS Code or a code cell in Jupyter Notebook and run. Explain why bin(8 << 2) is equal to bin(8 * 4).

```
print(bin(8))
print(bin(8 << 2), bin(8 * 4))
```

https://doi.org/10.15215/remix/9781998944088.01

IDENTITY OPERATORS

Identity operators are used to test if two operands, usually two identifiers, are identical, which in most implementations of Python means that they are referring to the same memory block of the computer. The examples Table 2-12 tell you more about this. Note that in the example, the built-in function id(o) is used to get the id of object o.

Table 2-12: Python identity operators

Operator	Meaning	Example

is	True if the operands are identical (refer to the same object)
	Note that 3 and 2 + 1 have the same id, and so does 6 // 2, but not 6 / 2 because 6 / 2 = 3.0, which is different from 3.
		>>> x = list(range(3))
		>>> y = list(range(3))
		>>> x, y
		([0, 1, 2], [0, 1, 2])
		>>> id(x), id(y)
		(10049456, 10049016)
		>>> x is y
		False
		>>> id(3)
		258398416
		>>> id(2 + 1)
		258398416

is not	True if the operands are not identical (do not refer to the same object).
	Note that when assigning variable y to variable x, the operation points x to the same memory block y is pointing to (so that the two have the id) and shares the same memory block. However, when a new assignment is made to x or y, the two will have different ids, unless they both hold *the same integer or string*.
		>>> x = list(range(3))
		>>> x = y
		>>> id(x), id(y)
		(10049016, 10049016)
		>>> x is y
		True
		>>> y = list(range(3))
		>>> id(x),id(y)
		(10090256, 10090336)
		>>> x, y
		([0, 1, 2], [0, 1, 2])
		>>> x is not y
		True

SEQUENCE OPERATORS

In 2.1, we learned that sequences include strings, lists, and tuples because elements in strings, lists, and tuples are ordered and indexed. Sets and dictionaries are not sequences because elements in dictionaries and sets are not ordered, or not in sequence.

Sequence operators are made available for operations on strings, lists and tuples, as shown in Table 2-13.

https://doi.org/10.15215/remix/9781998944088.01

Table 2-13: Python sequence operators

Operator	Operation	Code sample in Python interactive mode

*	Repeat a sequence such as string, list or tuple multiple times	`>>> "Python " * 3`
		`'Python Python Python '`
		`>>> [1, 2, 3] * 5`
		`[1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3]`

+	Join two sequences	`>>> [1, 2, 3] + [5, 6, 7]` `[1, 2, 3, 5, 6, 7]`

[n]	Slice out a single member of the sequence	`>>> name = 'John'`
		`>>> name[1]`
		`'o'`

[n:m]	Slice a sequence start from n to m.	`>>> name`
		`'John'`
		`>>> name[1:3]`
		`'oh'`

MEMBERSHIP OPERATOR

Membership operators are used to test whether an element is a member of a sequence. There are three membership operators in Python, and two of them are shown in Table 2-14.

Table 2-14: Python membership operators

Operator	Operation	Code sample in Python interactive mode

v in s	True if value v is found in sequence s	`>>> l = list(range(10))`
		`>>> l`
		`[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]`
		`>>> 3 in l`
		`True`
		`>>> 2 + 1 in l`
		`True`
		`>>> 'x' in "Welcome to COMP218"`
		`False`

https://doi.org/10.15215/remix/9781998944088.01

Table 2-14: Python membership operators *(continued)*

Operator	Operation	Code sample in Python interactive mode

v not in s	Checks whether value/variable is not found in the sequence	`>>> l`
		`[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]`
		`>>> 30 in l`
		`False`
		`>>> 30 not in l`
		`True`

The third membership operator is used to access members of objects, modules, or packages. It is the dot (.) operator. The example in Table 2-15 shows how to access a function in module math.

Table 2-15: Dot operator

Operator	Operation	Code sample in Python interactive mode

p.q	q is a member of p where p and q refer to an object (variable, constant, function or method name), package, or module name	`>>> import math`
		`>>> type(math)`
		`<class 'module'>`
		`>>> type(math.sqrt)`
		`<class 'builtin_function_`
		`or_method'>`
		`>>> math.sqrt(35)`
		`5.916079783099616`

Built-In Functions

As with operators, built-in functions are also important in processing and testing data in programs. As the name implies, built-in functions are built into Python Virtual Machine (PVM). A built-in function can be used directly without importing any module or noting what it belongs to.

Built-in functions, and functions or methods in general, can be put into two categories. One is based on the data they return, whereas the other is based on the operations performed, although sometimes the returned data from a built-in function in the second category may still be useful in subsequent executions of the program.

We will first explain built-in functions in the first category, followed by built-in functions in the second category. To best understand these functions, read through these built-in functions and test the sample code in a Python interactive shell or Jupyter Notebook.

https://doi.org/10.15215/remix/9781998944088.01

ABS(X)

This returns the absolute value of x, which can be any number.

```
>>> abs(-99)
99
>>> abs(-b110010)   # a binary number
50
>>> abs(-0o32560)   # an octal number
13680
>>> abs(0xdef21a)   # a hexadecimal(16) number
13610970
```

INT(S, BASE = 10)

This converts a number s in a specific base to an integer in base-10. The default base is 10. If the base is explicitly given, s must be a string containing a legitimate literal for the base.

```
>>> int(19.9)
19
>>> int("22", 8)   # in the quote must be a legitimate
literal for base-8
18
>>> int('0x123', base = 16)   # a number in base-16
291
```

POW(X, P)

This returns the value of x to the power of p.

```
>>> pow(2.9, 12.8)
829266.980472172
```

FLOAT(S)

This converts s to float. s can be a number, or a string of numbers.

```
>>> float('18.23')
18.23
>>> float(19)
19.0
```

https://doi.org/10.15215/remix/9781998944088.01

MAX(ITERABLE, *[, DEFAULT = OBJ, KEY = FUNC])
MAX(ARG1, ARG2, *ARGS, *[, KEY = FUNC])

These find and return the biggest element from an iterable such as a list, tuple or string, or from two or more arguments. The default keyword-only argument specifies what will be returned if the iterable is empty. The key keyword-only argument specifies how the maximum is defined if it is out of the ordinary.

```
>>> max(2, 1, 5, 65, 89)   # variable-length list of
arguments
89
>>> max("this")   # the given sequence is a string
't'
>>> max((2, 3, 5, 1, 78))   # numbers in a tuple
78
>>> max([2, 3, 5, 1, 78])   # numbers in a list
78
```

MIN(ITERABLE, *[, DEFAULT = OBJ, KEY = FUNC])
MIN(ARG1, ARG2, *ARGS, *[, KEY = FUNC])

These find and return the smallest number from an iterable such as a list, tuple or string, or from two or more arguments. The default keyword-only argument specifies what will be returned if the iterable is empty. The key keyword-only argument specifies how the minimum is defined if it is out of the ordinary.

```
>>> min(6, 5, 8, 3, 2)
2
>>> min([2, 3, 5, 1, 78])   # numbers in a list
>>> min([], default = 0)   # 0 will be returned because
the list is empty
0
```

ROUND(F)

This rounds number f to the closest integer and returns the integer.

```
>>> round(3.1415926)
3
```

ORD(C)

This finds and returns the order of a character, as a single char string, in the ASCII table.

https://doi.org/10.15215/remix/9781998944088.01

```
>>> ord('c')
99
```

SUM(...)

This calculates and returns the sum of numbers in a list, a tuple, or a range() call.

```
>>> sum([23, 56, 67, 12, 89])
247
>>> sum((23, 56, 67, 12, 89))
247
>>> sum(range(88))
3828
```

SET(S)

This converts a set from a list or tuple.

```
>>> set([23, 56, 67, 12, 89])
{23, 56, 67, 12, 89}
```

DICT()
DICT(ITERABLE)
DICT(A = V,...)

These convert an empty dictionary, construct a dictionary from the iterable of (k, v) tuples, and from key=value pairs, respectively.

```
>>> dict()
{}
>>> dict([(1,'Turing'), (2,'Bool'), (3,'Babbage'),
(4,'Neumann'), (5,'Knuth')])
{1: 'Turing', 2: 'Bool', 3: 'Babbage', 4: 'Neumann', 5:
'Knuth'}
>>> dict(a = 1, b = 2, c = 3)
{'a': 1, 'b': 2, 'c': 3}
```

BIN(N)

This converts a number to its binary equivalence as a string.

```
>>> bin(98)
'0b1100010'
```

https://doi.org/10.15215/remix/9781998944088.01

HEX(N)

This converts a number to its hex equivalence as a string.

```
>>> hex(19)
'0x13'
```

OCT(N)

This converts a number to its oct equivalence as a string.

```
>>> oct(28)
'0o34'
```

BOOL(O)

This converts o to Boolean True or False. In Python, 0, '', and None are equivalent to False, everything else is equivalent to True.

```
>>> bool(1)
True
>>> bool('school')
True
>>> bool(0)
False
```

TUPLE(S)

This constructs a tuple from a list, a string, or range() call.

```
>>> tuple("this is tuple")
('t', 'h', 'i', 's', ' ', 'i', 's', ' ', 't', 'u', 'p',
'l', 'e')
```

LEN(S)

This returns the length of a sequence.

```
>>> len(my_tuple)
13
>>> len("I like Python so much!")
22
```

LIST(S)

This constructs a list from a sequence or range() call.

https://doi.org/10.15215/remix/9781998944088.01

```
>>> list(range(5))
[0, 1, 2, 3, 4]
```

RANGE(START, STOP, STEP)

This returns a sequence of numbers starting from 0 by default, ending right before stop and increasing by one step by default.

```
>>> list(range(1, 9, 2))
[1, 3, 5, 7]
```

COMPLEX(A, B)

This constructs a complex number from a pair of numbers and returns the complex number.

```
>>> complex(1, 8)
1 + 8j
```

HASH(S)

This generates a hash for a given string s and returns the hash. One use is for transmitting and saving passwords.

```
>>> hash("Python is a great language for programming")
6145305589036905122
```

DIVMOD(A, B)

This returns a tuple of the quotient and the remainder of one integer or float number divided by another integer or float number.

```
>>> divmod(23, 5)
(4, 3)
```

STR(X)

This converts object x literally to a string and returns the converted string.

```
>>> str([23, 56, 67, 12, 89])
'[23, 56, 67, 12, 89]'
```

https://doi.org/10.15215/remix/9781998944088.01

CHR(N)

This returns the character n with its code in the Unicode table. Note that 0 <= n <= 0x10ffff as a legitimate code.

```
>>> chr(90)
'Z'
>>> chr(99)
'c'
```

TYPE(O)
TYPE(C, BASES, DICT)

type(o) returns the data type of object o, whereas type(C, bases, dict) will create a new type/class whose name is C and whose base classes are in bases, which is a tuple, and the dictionary defines the attributes of the new class, with assigned values. This gives programmers a way to dynamically define classes.

```
>>> type(list(range(9)))
<class 'list'>
```

In [ ]:
```
X = type('X', (object), dict(a = 1, b = 3))   # create
  a new class named X
x = X()   # create an instance of X
print(f'x.a = {x.a}, x.b = {x.b}')
x.a, x.b = 23, 35   # assign values to x's attribute a
  and b
print(f'x.a = {x.a}, x.b = {x.b}')
```

Out [ ]:
```
x.a = 1, x.b = 3
x.a = 23, x.b = 35
```

ALL(ITERABLE)

This returns True if all the elements of iterable are true.

```
>>> all(range(9))
False

>>> all(range(1,9))
True
```

https://doi.org/10.15215/remix/9781998944088.01

ANY(ITERABLE)

This returns True if any of the arguments true.

```
>>> any(range(9))
True
>>> any([0,0,0,0])
False
```

DIR()
DIR(O)

dir() returns a list of names in the current namespace. dir(o) returns a list of the attributes of object o.

```
>>> dir()
['__annotations__', '__builtins__', '__doc__', '__
loader__', '__name__', '__package__', '__spec__']

>>> dir(math)
['__doc__', '__loader__', '__name__', '__package__',
'__spec__', 'acos', 'acosh', 'asin', 'asinh', 'atan',
'atan2', 'atanh', 'ceil', 'copysign', 'cos', 'cosh',
'degrees', 'e', 'erf', 'erfc', 'exp', 'expm1', 'fabs',
'factorial', 'floor', 'fmod', 'frexp', 'fsum',
'gamma', 'gcd', 'hypot', 'inf', 'isclose', 'isfinite',
'isinf', 'isnan', 'ldexp', 'lgamma', 'log', 'log10',
'log1p', 'log2', 'modf', 'nan', 'pi', 'pow', 'radians',
'remainder', 'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'tau',
'trunc']
```

NEXT(IT)

This returns the next element of an iterable such as list, string, tuple, and so on.

```
>>> l=iter(list(range(3,99)))
>>> next(l)
3
>>> next(l)
4
```

ASCII(O)

This returns a string containing a printable ASCII representation of an object.

https://doi.org/10.15215/remix/9781998944088.01

```
>>> ascii(math)
"< module 'math' (built-in)>"
>>> ascii(int)
"<class 'int'>"
```

ID(O)

This returns object o's "identity," which is a unique integer within a given context, usually the address of the object in memory.

```
>>> i = 10
>>> id(i)   # return the id of variable i
263313728
>>> i *= 98
>>> id(i)   # id is different, but still the same i
2809440
```

SORTED(S)

This returns a new sorted list of elements in iterable s.

```
>>> il = [12, 0, 9, 32, 8, 5, 3, 99] # il is a list of
integers
>>> sorted(il)   # default is to sort in ascending order
[0, 3, 5, 8, 9, 12, 32, 99]
>>> sorted(il, reverse = 1)   # sorted in descending order
[99, 32, 12, 9, 8, 5, 3, 0]
```

REVERSED(S)

This returns a reversed iterator.

```
>>> il = [0, 3, 5, 8, 9, 12, 32, 99]
>>> list(reversed(il))
[99, 32, 12, 9, 8, 5, 3, 0]
>>> list(reversed(range(9))) # range(9) return a sequence
of 0,1,…9
[8, 7, 6, 5, 4, 3, 2, 1, 0]
```

ENUMERATE(S, START = 0)

This returns a list of tuples from a sequence in which the elements are counted and each element is paired with its count to form a tuple.

https://doi.org/10.15215/remix/9781998944088.01

```
>>> list(enumerate("this is"))   # default value for
optional argument start is 0
[(0, 't'), (1, 'h'), (2, '€'), (3, 's'), (4, ' '), (5,
'€'), (6, 's')]
>>> list(enumerate("this is", 2))   # now counting start
from 2
[(2, 't'), (3, 'h'), (4, '€'), (5, 's'), (6, ' '), (7,
'€'), (8, 's')]
>>>
```

EVAL(S, GLOBALS = NONE, LOCALS = NONE, /)

This evaluates the expression in s as a string or a compiled Python code in s and returns the value. Global and local namespaces can be specified for the expression or code object using the keyword arguments.

```
>>> eval("1 / (1 + (1 / math.e) ** 12)")
0.9999938558253978
```

EXEC(S)

This executes the statement in string s and provides a way to dynamically execute the Python code.

```
>>> exec("print('Hello World!')")
Hello World!
```

```
In [ ]:  def cubbie(n):
           return n * n * n
         src = "print(cubbie(23))"
         exec(src)

Out [ ]: 12167
```

ZIP(*ITERABLES)

This returns a list of tuples by taking one element from each of the iterables to make a tuple until reaching the end of the shortest iterable, and then returning the tuple. In Python, *p notation means p takes multiple arguments. In this case, multiple iterables such as lists, tuples, or strings are expected.

https://doi.org/10.15215/remix/9781998944088.01

```
>>> grade_n = [50, 70, 80, 90, 100]
>>> grade_c = ['F', 'D', 'C','B', 'A']
>>> list(zip(grade_n, grade_c))
[(50, 'F'), (70, 'D'), (80, 'C'), (90, 'B'), (100, 'A')]
```

```
In [ ]:  chars = [chr(i) for i in range(32,97)] # a list of chars
         asc_table = zip(range(32,97), chars)
         for coding in asc_table:
           print(coding)
```

```
Out [ ]: (32, ' ') (33, '!') (34, '"') (35, '#') (36, '$') (37, '%') (38, '&') (39, "'")(40, '(') (41, ')')
           (42, '*') (43, '+') (44, ',') (45, '-') (46, '.') (47, '/') (48, '0') (49, '1') (50, '2') (51,
           '3') (52, '4') (53, '5') (54, '6') (55, '7') (56, '8') (57, '9') (58, ':') (59, ';') (60, '<')
           (61, '=') (62, '>') (63, '?') (64, '@') (65, 'A') (66, 'B') (67, 'C') (68, 'D') (69, 'E')
           (70, 'F') (71, 'G') (72, 'H') (73, 'I') (74, 'J') (75, 'K') (76, 'L') (77, 'M') (78, 'N')
           (79, 'O') (80, 'P') (81, 'Q') (82, 'R') (83, 'S') (84, 'T') (85, 'U') (86, 'V') (87, 'W')
           (88, 'X') (89, 'Y') (90, 'Z') (91, '[') (92, '\\') (93, ']') (94, '^') (95, '_') (96, '`')
```

The code sample above prints a portion of ASCII table showing the codes from 32 to 96 and their corresponding characters. It is only to show how zip function is used. There is a much simpler way to print such a table using just one ***for*** loop.

MAP(F, *ITERABLES)

This applies function f to every item of an iterable and returns the resulted iterator.

```
>>> import math
>>> list(map(math.sqrt, range(17))
[0.0, 1.0, 1.4142135623730951, 1.7320508075688772, 2.0,
2.23606797749979, 2.449489742783178]
>>> list(map(sum, ([1, 2, 3], [4, 5, 6], [7, 8, 9, 10])))
[6, 15, 34]
```

GETATTR(O, ATTR)

This returns the value of object o's attribute attr, the same as o.attr.

```
>>> getattr(math, 'sqrt')
<built-in function sqrt>
>>> getattr(math, 'e')
2.718281828459045
```

https://doi.org/10.15215/remix/9781998944088.01

HASATTR(O, ATTR)

This tests if object o has attribute attr and returns True if it does.

```
>>> hasattr(math, 'e')
True
>>> hasattr(math, 'sqrt')
True
>>> hasattr(math, 'power')
False
```

SETATTR(O, A, V)

This sets or adds an attribute a to object o and assigns value v to the attribute.

```
>>> class Student:   # defining a class named Student. By
convention, class names should be capitalized
…pass   # this defines a class without any attribute
…
>>> s1 = Student()   # create an instance of Student
>>> setattr(s1, 'name', 'John')   # add an attribute
called name, and assign 'John' to it
>>> s1.name
'John'
>>> hasattr(s1, 'name')
True
```

DELATTR(O, A)

This deletes attribute a from object o.

```
>>> delattr(s1, 'name') # delete attribute name from
object s1
>>> hasattr(s1, 'name') # check if s1 has attribute name
False
```

ISINSTANCE(O, C)

This returns True if o is an instance of class c or a subclass of c.

```
>>> class Student:
…pass
…
>>> s1 = Student()
```

https://doi.org/10.15215/remix/9781998944088.01

```
>>> isinstance(s1, Student)
True
```

ISSUBCLASS(C, C)

This returns True if class c is a subclass of C.

```
>>> class Graduate(student):
…pass
…
>>> issubclass(Graduate, Student)
True
```

REPR(O)

This returns a string representation of object o.

```
>>> repr(Graduate)
"<class "__main__.graduate'>'
```

FILTER(F, ITERATOR)

This returns an iterator containing only the elements of the iterable for which the function returns true.

```
>>> def even(n):
…return not n%2 # return True if n can be divided by 2
…
>>> list(filter(even, range(9)))   # odd numbers will be
taken out
[0, 2, 4, 6, 8]
```

CALLABLE(O)

This returns True if o is a callable object such as a defined function.

```
>>> callable(even)   # it will return True since even is
defined
True
```

LOCALS()

This updates and returns a dictionary of local names/symbols.

https://doi.org/10.15215/remix/9781998944088.01

```
>>> locals()
{'__name__': '__main__', '__doc__': None, '__package__':
None, '__loader__': <class '_frozen_importlib.
BuiltinImporter'>, '__spec__': None, '__annotations__':
{}, '__builtins__': <module 'builtins' (built-in)>,
'math': <module 'math' (built-in)>, 'l': <list_iterator
object at 0x00370230>, 'student': <class '__main__.
student'>, 's1': <__main__.student object at 0x00CE3DB0>,
'graduate': <class '__main__.graduate'>, 'even':
<function even at 0x0029F7C8>}
```

VARS()
VARS(O)

vars() returns the same as locals(), whereas vars(o) returns the _dict_ attribute of object o.

```
>>> setattr(s1, 'name', 'John')
>>> vars(s1)
{'name': 'John'}
```

GLOBALS()

This updates and returns a dictionary of global names/symbols in the current scope.

```
>>> globals()
{'__name__': '__main__', '__doc__': None, '__package__':
None, '__loader__': <class '_frozen_importlib.
BuiltinImporter'>, '__spec__': None, '__annotations__':
{}, '__builtins__': <module 'builtins' (built-in)>,
'math': <module 'math' (built-in)>, 'student': <class
'__main__.student'>, 's1': <__main__.student object at
0x00CE3DB0>, 'graduate': <class '__main__.graduate'>,
'even': <function even at 0x0029F7C8>}
```

BYTEARRAY([SOURCE[, ENCODING[, ERRORS]]])

This returns a bytearray object which is an array of the given bytes.

```
>>> s = "Hello Python lover!"
>>> barry = bytearray(s, 'utf-8')
>>> print(barry)
bytearray(b'Hello Python lover!')
```

https://doi.org/10.15215/remix/9781998944088.01

BYTES([SOURCE[, ENCODING[, ERRORS]]])

```
>>> bs = bytes(s, 'utf-8')
>>> print(bs)
b'Hello Python lover!'
```

BREAKPOINT(*ARGS, **KWS)

This function break the program and takes it into debug mode, calls sys. breakpointhook(), and passes a list of arguments (args) and a list of keyword arguments (**kws) to the system function.

@CLASSMETHOD

The at sign @ is called a decorator in Python. This particular decorator is used to declare a method as class method, which receives the class as its first argument.

```
# define a class Person
class Person:
    # define a class attribute
    species = "human"
```

```
    # define an instance method
    def __init__(self, name, age):
        self.name = name
        self.age = age
```

```
    # define a class method using the @classmethod
decorator
    @classmethod
    def from_birth_year(cls, name, birth_year):
        # calculate the age from the birth year
        age = 2023 - birth_year
        # return a new instance of Person with the given
name and age
        return cls(name, age)
```

```
# create an instance of Person using the class method
p1 = Person.from_birth_year("Alice", 1995)
# print the instance attributes
print(p1.name) # output: Alice
print(p1.age) # output: 28
print(p1.species) # output: human
```

https://doi.org/10.15215/remix/9781998944088.01

The code above was taken from a code cell in Jupyter Notebook. The output is as follows when you hit the Ctrl+Enter key to run the code:

```
Alice
28
Human
```

COMPILE(SOURCE, FILENAME, MODE, FLAGS = 0, DONT_INHERIT = FALSE, OPTIMIZE = −1)

This is used to compile the source into a code that can be executed using eval() or exec().

FORMAT(VALUE[, FORMAT_SPEC])

This is used to convert a value to a "formatted" representation, as controlled by format_spec.

```
>>> print("Modern computers have over {h:3d} years of
history".format(h = 80))
Modern computers have over 80 years of history
```

FROZENSET([ITERABLE])

This returns a new frozenset object, with the option to display it with elements taken from an iterable. frozenset is also a built-in class.

```
>>> l = list(range(10))
>>> print(frozenset(l))
frozenset({0, 1, 2, 3, 4, 5, 6, 7, 8, 9}) # {…} is
a set
```

HELP([OBJECT])

This invokes the built-in help system on the object.

```
>>> help(frozenset)
```

Using help on class frozenset in module builtins displays the following:

```
class frozenset(object)
 |  frozenset() -> empty frozenset object
 |  frozenset(iterable) -> frozenset object
 |
```

https://doi.org/10.15215/remix/9781998944088.01

```
 |  Build an immutable unordered collection of unique
elements.
 |…
```

INPUT([PROMPT])

This is used to read a line from input, convert it to a string with trailing newline characters stripped, and return the string. The optional prompt argument will be displayed without a trailing newline character so that the cursor will just wait at the end of the prompt for input.

```
>>> s = input("please give me an integer:")
please give me an integer:
```

ITER(OBJECT[, SENTINEL])

This returns an iterator object. If the second argument doesn't exist, the first argument must be a collection object.

```
>>> for c in range(6): print(next(l))
…
P
y
t
h
o
n
```

MEMORYVIEW(OBJ)

This returns a "memory view" object of obj. Note that obj must be a bytes-like object.

```
>>> mv = memoryview(b"Hello Python Lover")
>>> print(mv)
<memory at 0x000001B932AD4A00>
```

OBJECT

This returns a new featureless object, a base for all classes.

```
>>> O = object
>>> print(O)
<class 'object'>
```

https://doi.org/10.15215/remix/9781998944088.01

OPEN(FILE, MODE = 'R', BUFFERING = −1, ENCODING = NONE, ERRORS = NONE, NEWLINE = NONE, CLOSEFD = TRUE, OPENER = NONE)

This opens the file in the desired mode and returns a corresponding file object. The default mode is read.

```
>>> f = open('scoresheet.txt', 'w')
```

The example opens a file named scoresheet.txt for writing and assigns the handle to f.

PRINT(*OBJECTS, SEP = ' ', END = '\N', FILE = SYS.STDOUT, FLUSH = FALSE)

This prints objects to the text stream file, separated by separator and followed by end. sep, end, file, and flush, if present, must be given as keyword arguments.

```
>>> print("Hello Python Lover!")
Hello Python Lover!
```

PROPERTY(FGET = NONE, FSET = NONE, FDEL = NONE, DOC = NONE)

This returns a property attribute. fget, sfet, and fdel take functions for getting, setting, and deleting an attribute value.

SLICE(STOP)
SLICE(START, STOP[, STEP])

These return a slice object representing the set of indices specified by range(start, stop, step).

```
>>> s = "Hello Python Lover!"
>>> slicing = slice(3)   # slicing the first 3 items out
of an object
>>> print(s[slicing])   # this will take the first three
characters from the s
Hel
```

@STATICMETHOD

This function decorator is used to declare a method as static. A static method can be called on the class or an instance.

https://doi.org/10.15215/remix/9781998944088.01

```
In [ ]:   class FTool():
              @staticmethod
              def percentage(a, b):
                  return a/b

          r = FTool.percentage(13, 265)
          print(f"{13}/{256} is {r}")
```

```
Out [ ]:  13/256 is 0.04905660377358491
```

SUPER([TYPE[, OBJECT-OR-TYPE]])

This returns a proxy object that delegates method calls to a parent or sibling type class. It is used for accessing inherited methods that have been overridden in a class.

```
In [ ]:   class FTool():
              @staticmethod
              def percentage(a, b):
                  return a/b

          print(super(FTool))
```

```
Out [ ]:  <super: <class 'FTool'>, NULL>
```

The superclass is NULL because FTool is not a subclass of any class except object, which doesn't count.

Expressions

Expressions are important program constructs. An expression is made up of data items, variables, constants, and function calls joined by proper operators. The precedencies of the operators are as follows:

1. Within arithmetic operators, other operators take precedence over addition and subtraction.
2. Arithmetic operators take precedence over comparison operators.
3. Membership operators, identity operators, and comparison operators take precedence over logic operators.
4. Among logic operators, the order of precedence, from high to low, is ***not*** > ***and*** > ***or***.

https://doi.org/10.15215/remix/9781998944088.01

CODING ALERT

Python language is case-sensitive. Be sure to use the right case when typing!

Expressions are used almost everywhere in a program and will be evaluated to a value or object in a general term. According to the type of the value or object from evaluation, an expression can be any of the following.

ARITHMETIC EXPRESSIONS

An arithmetic expression's value is always one of the following: an integer, float, or complex. An arithmetic expression can be made of data, variables, function calls, and arithmetic operators. When mixed data types appeared in an expression, the value type of the expression will be the most general data type. For example, the value of 3 + 5.6 will be a float number.

STRING EXPRESSIONS

The string expression's value is a string. String expressions are made of strings, string operators, functions and methods that return a string.

BOOLEAN EXPRESSIONS

The Boolean expression's value is either True or False. Boolean expressions can be made of data, comparison operators, and logical operators. Note that although Python has True and False defined as logical true and false, it treats 0, None, empty string, empty list, empty tuple, set, and dictionary as False and treats everything else as True.

OTHER EXPRESSIONS

The values of some expressions may be a list, tuple, set, dictionary or even a complex object. For example, some functions and methods can return a list or an object of a user-defined class, and the operator + can be used to combine two strings or lists together.
The following are some examples of expressions in Python:

```
12 + 35.6 - 36 * 3 + x   # integer and float numbers can
be mixed
235 + x ** k   # 235 plus x to the power of k
2 < j and j in list_x   # 2 is less than j and j is a
member of list x
```

https://doi.org/10.15215/remix/9781998944088.01

Expressions are often used on the right-side of an assignment operator, such as

```
total = a1 + a2 + a3
i *= j + 2   # the same as i = i * (j + 2)
```

CODING ALERT

In Python, # is used to add end-of-line comments. Anything behind # is ignored by PVM.

CODING PRACTICE

Within VS Code, create a new Jupyter Notebook and rename it section-2.1, or double-click the notebook name on the left-hand file navigation area to open the notebook if you have already created one. In a cell, type the following code and hit Shift+Enter to run the code. Then manually evaluate the expression within the curly braces of the ***print*** statement and compare your result to the one given by the code. Explain why the results are different.

```
x, y = 10.23, 5.2
m, n = 27, 8
print(f'{x * y + 27 // n}')
```

2.2 Higher-Level Constructs of Python Programs

The constructs you learned in section 2.1 are small and meant to be used as parts of bigger ones. The big constructs of programs are called statements, which give explicit instructions to computers to act upon.

Structure of Python Programs

Before diving into the details of writing statements in Python, this section will first look at the general structure of a Python program and the coding style widely agreed upon among the Python community, which will make it so the programs that you write are readable to others.

For a simple application, a single Python file (with py as its extension) may be enough to contain all the program code needed to implement the application;

https://doi.org/10.15215/remix/9781998944088.01

for a more complex application, however, several or even hundreds of Python files may be needed. Of these files, there will be only one Python file defining the starting point of the program that implements the application, while all other files are used as modules to be imported into the main Python file, either directly or indirectly. So essentially, the relationships of all the Python files used for an application can be depicted as a tree in which the root is the main Python file.

The modules used directly or indirectly by the main Python program may be from the standard libraries installed with Python or from those installed later using the conda or pip command as needed.

Regarding coding style, see PEP 8: Style Guide for Python Code (https://pep8.org/), which describes in great detail how Python code should be written. Read it thoroughly and review it whenever you are unsure. Below are the highlights:

1. A Python program/script file should begin with a docstring as the main documentation of the program file, stating the application and functionality of the program, as well as the author and revision history.
2. In a script file, use double blank lines to separate the actual program code from the documentation section at the beginning of the file.
3. Also use double blank lines to separate top-level function and class definitions.
4. Use a single blank line to surround the definition of a method in a class definition.

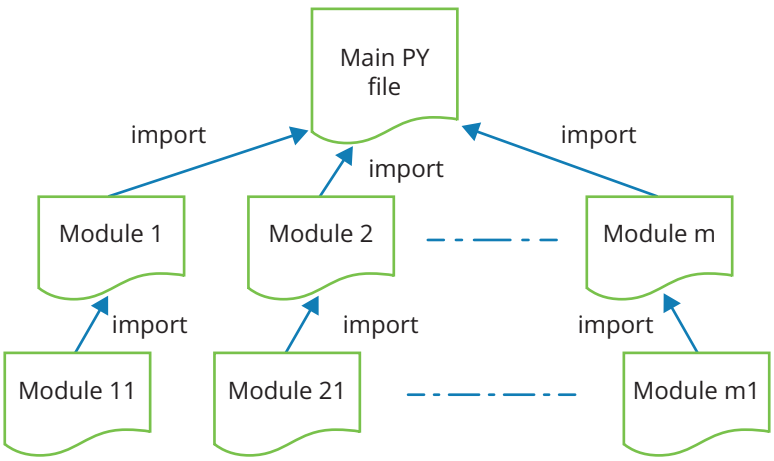


Figure 2-5: Illustration of the structure of files for a Python application

https://doi.org/10.15215/remix/9781998944088.01

5. Pay attention to indentation, especially when an expression, a simple statement, or the header of a compound statement is too long and needs to cross multiple lines.
   a. When an expression or a statement needs a closing brace, bracket, or parenthesis mark to complete it, there is no need to escape (\) newline at the end of an unfinished line.
   b. When a string needs to cross multiple lines, newline must be escaped by putting a backslash at the end of each unfinished line.
   c. The four-space rule is optional. The next line can be started wherever it makes more sense, such as in the column next to the opening delimiter.

In addition to the rules of coding, it's important to maintain a consistent coding style and to make sure that the programs not only are easy to read and understand but also look neat and nice.

Documentation and Comments

As mentioned above, some lines of documentation, called docstring, are needed at the very beginning of each Python script/program file to state the purpose and functionality of the program, who made it and when, and notes for you and others who may read the program.

The following sample program calculates the area of a circle for a given radius. It shows how docstring is used in the program file.

```
"""
This program is used to calculate the area of a circle. It
  will take an input as
a radius, then calculate and print out the area.

File name: circle.py
Author: John Doe
Date: March 30, 2019
Version: 1.0
"""


radius = int(input("tell me the radius:"))   # take input
  from user
area = 3.14 * radius ** 2   # calculate the area
print(f"The area of a circle with radius {radius} is
  {area}")   # print

```

https://doi.org/10.15215/remix/9781998944088.01

Please note the triple quotation marks on line 1 and line 9. The triple quotation marks on line 1 mark the start of the docstring, whereas the triple quotation marks on line 10 mark the end of the docstring. The quotation marks can be single or double, but they must be the same. You can also add docstrings for functions, classes, class methods, or other code blocks in the program as needed, but the opening triple quotation marks and ending triple quotation marks must be indented the same amount as the code block. You will see how this should be done in later chapters, with examples.

Please also note the comments starting with a # at the end of lines 13, 14, and 15. They are called end-of-line comments or block notes. An end-of-line comment is usually used to explain what the code on the line does. Everything behind the # mark on that line is ignored by Python Virtual Machine (PVM) and intended for only humans to read. An end-of-line comment can also be started at the beginning of a line.

The difference between docstrings and end-of-line comments is that docstrings are formal documentation of the program or module and are accessible through the built-in help() function, with the _doc_ variable automatically attached to each module, function, class and method, whereas end-of-line comments are not. As well, utility tools such as pydoc are available for generating formal documentation for a program or module from the docstrings within each Python file. The revised version of program circle.py is shown below, in which we defined a function named area, with docstrings added to the function and the program.

```
1  """
2  The purpose: revised circle.py program with more
     docstrings, to demonstrate how docstrings and
3  inline documentation is used in Python programs.
4
5  This program is designed to calculate the area of a circle.
     It takes an input from the user for the radius and
6  calculates and prints out the area.
7
8  File name: circle-v2.py
9  Author: John Doe
10 Date: March 30, 2019
11 Version: 1.0
12 """
13
14 def area(r):
15 """To calculate the area of a circle for a given
     radius."""
16    return 3.14 * r ** 2   # calculate the area
```

https://doi.org/10.15215/remix/9781998944088.01

```
17 """Add docstring after return statements?"""
18
19 radius = int(input("tell me the radius:"))   # take input
    from user
20 print(f"The area of a circle with radius {radius} is
    {area(radius)}")   # printout
21
22 """I want to add more docstrings."""
```

The following are some general rules for program documentation:

1. A docstring should also be written for every function, class, and public method right after the header of the definition. Docstrings must be indented the same amount as the suite of function or class definition.
2. Limit the maximum line length to 79 or 72 characters if the line is part of a docstring.
3. Use inline comments whenever necessary.
4. Some code may need more than one line of comments, which makes it a block comment. A block comment should be written right before the code and indented to the same level as the code below.

CODING TRICK

Document! Document! Document!

Documentation is necessary not only for users but for the programmer who made the program. The programmer may think they know everything about the program, but time can blur their memory.

Simple Statements

Normally, a simple statement is contained within a single logical line, though Python allows several simple statements to appear on one line separated by semicolons. There will be examples of this later.

EXPRESSION STATEMENT

Simply put, expression statements in Python programs are just expressions in mathematical terms. Any expression can be used as a statement, and PVM will evaluate every expression statement, but the result is only displayed in Python interactive mode, as shown in the following code sample in the Python Shell:

https://doi.org/10.15215/remix/9781998944088.01

Code sample in Python interactive mode

```
1
2  >>> 2 * 3.14 * 15
3  94.2
4  >>> 3.14 * 15 ** 2
5  706.5
6  >>>
7
8
```

So if you are using Python in interactive mode, you can simply type expressions without using the ***print*** statement to see the result of the calculation, just like a powerful calculator.

We can also have expression statements in Jupyter Notebook, but when there are several expression statements in the same cell, it will only show the result of the last expression, as shown in the following example:

```
In [ ]:   2 * 3.14 * 15
          3.14 * 15 ** 2

Out [ ]:  706.5
```

We mentioned earlier in this subsection that you can also put several simple statements on the same line, but you must separate them with semicolons. This is shown in the following examples:

Code sample in Python interactive mode

```
1
2  >>> 2 * 3.14 * 15; 3.14 * 15 ** 2
3  94.2
4  706.5
5  >>>
6
7
8
```

The expression statement above can also be given in Jupyter Notebook, as shown below. In Jupyter Notebook, however, you have to press Shift+Enter or click the play button to run the scripts in an active cell:

https://doi.org/10.15215/remix/9781998944088.01

```
In [ ]:  2 * 3.14 * 15; 3.14 * 15 ** 2

Out [ ]: 706.5
```

Again, in Jupyter Notebook, even if multiple expression statements are on the same line, only the result of the last expression statement will be displayed.

ASSIGNMENT STATEMENT

The assignment statement is one of the most important statements and is used most often in programs because it is a common requirement to keep the result of computing or information processing in the computer memory for future uses. It does so by assigning the result to a variable.

In section 2.1, we saw a list of assignment operators. Formally, an assignment statement is made of a variable on the left and an expression on the right of an assignment operator, either = or an augmented one such as +=. We have already seen some examples of assignment statements before, but the following code sample includes a few more examples:

```
In [ ]:  d = 15
         r = d / 2
         x = d * 3.14
         a = r * r * 3.14
         print(f"diameter = {d}; radius = {r}; circumference =
           {x}; area is {a}")

Out [ ]: diameter = 15; radius = 7.5; circumference = 47.1; area is 176.625
```

AUGMENTED ASSIGNMENT

In programming, we often take the value of a variable, perform an operation on it, then put the result back to the variable. That is when augmented assignment comes into play.

In section 2.1, we saw several augmented assignment operators. In general, for an assignment in the form of

```
x = x <operator> v
```

the augmented assignment can be used in the following form:

```
x <operator>= v
```

https://doi.org/10.15215/remix/9781998944088.01

The following are some examples of augmented assignments:

```
In [ ]:  y = 10
         n = 5
         print(f"y is {y}; n is {n}")
         y *= n   # y * n is assigned to y; it is equivalent to
          y = y * n
         print(f"y is {y}; n is {n}")
         n += 2
         y /= n
         print(f"y is {y}; n is {n}")

Out [ ]: y is 10; n is 5
         y is 50; n is 5
         y is 7.142857142857143; n is 7
```

To understand these augmented assignment statements, we need to consider the memory location referred to by a variable, such as y with respect to time. Take y *= n as an example. At time t0 before the actual assignment starts, the value of y (data stored in the memory referred to by y, which is 10 at time t0) is taken out, and multiplied by n, whose value is 5 at time t0; then time t1—the result, which is 50 (from 10 * 5)—is stored in the memory location referred to by variable y.

CODING ALERT

The number of value items must be enough for the variables to be assigned.

MULTIPLE ASSIGNMENTS

Also, for convenience and efficiency, you can assign values to multiple variables in a single assignment statement. There are several ways of doing multiple assignments, as shown in the following examples:

```
In [ ]:  x, y = 1, 2   # assign 1 to x, assign 2 to y
         (x, y) = (1, 2)   # it is the same as x, y = 1, 2
         x, y = [1, 2]   # it is the same as x, y = 1, 2
         k, *l = [1, 2, 3, 5]   # this will assign 1 to k, and
          assign the rest to l as list, because of *l
         x, *y, z = [1, 2, 3, 5, 6, 7]   # this will assign 1 to
          x, 7 to z, [2, 3, 5, 6] to y
         print(f"x = {x}; y = {y}; z = {z}")

Out [ ]: x = 1; y = [2, 3, 5, 6]; z = 7
```

https://doi.org/10.15215/remix/9781998944088.01

In the last example above, *l, *y tells PVM that variable l and y can take a variable-length list of values.

CONDITIONAL ASSIGNMENTS

Additionally, in Python, you may even assign different values to a variable under different conditions, as shown in the following example:

Code sample in Python interactive mode

```
>>> marks = 90
>>> gr = 'pass' if marks >= 60 else 'fail'   # note that
  elif clause does not work on the conditional assignment
>>> gr
'pass'
>>> marks = 50
>>> gr = 'pass' if marks >= 60 else 'fail'
>>> gr
'fail'
```

As you will see throughout the text, there are many fancy ways of making statements in Python, and that's why Python is a very powerful language. Indeed, the full power of a programming language can only be materialized by the best programmers.

ANNOTATED ASSIGNMENT STATEMENT

We know that variables in Python are dynamically typed. Sometimes, however, it is nice to indicate what type of data is expected for a variable. In Python, this is done by annotating the variable using a colon followed by the name of the data type, as shown in the following example:

```
marks: float = 86.5   # variable marks will hold a float
number
```

However, Python will not complain if other types of data are assigned, as shown below:

```
>>> marks: float = "Python"
>>> marks
'Python'
>>>
```

To annotate the type of value returned from a function, you need to use -> followed by the data type, right before the colon, as shown in the following example:

https://doi.org/10.15215/remix/9781998944088.01

```
In [ ]:  def literal_grade(marks:float) -> str:
           return 'fail' if marks < 60 else 'pass'

         print(literal_grade(88))

Out [ ]: pass
```

In this example, marks:float determines that a float number is expected for marks when calling the function, and -> str dictates that a string is to be returned.

It must be made clear that the actual type of a variable is still determined by the value it holds, and annotation added to a variable or function doesn't really change the type of the variable or the value returned from a function. So annotations are more for programmers as reminders.

PRINT STATEMENT

The ***print*** statement is another one of the most important statements. It is used to output (to a terminal by default) so that a program can tell human users the result of calculations or information processing, the value of an object, or the status of the program. The following are some examples of ***print*** statements:

```
In [ ]:  print("this is my first print statement.")

         print("it will take about ", round(300/110), "hours to
           drive from Edmonton to Calgary")

Out [ ]: this is my first print statement.
         it will take about 3 hours to drive from Edmonton to Calgary
```

The ***print*** statement can evaluate multiple arguments and print out the values. If the arguments are all constant strings, there will be no need to separate them into multiple arguments. Separate arguments are needed only when some arguments are expressions, like the round(300/110) in the above example. If you do not like separations, introduced in release 3.0, Python provides a neat way to include expressions all in a single pair of quotation marks, as shown in the following example:

```
In [ ]:  print(f"it will take about {round(300/110)} hours to
           drive from Edmonton to Calgary")

Out [ ]: it will take about 3 hours to drive from Edmonton to Calgary
```

https://doi.org/10.15215/remix/9781998944088.01

Please note the f—which is a flag or prefix—before the opening quotation mark and the curly brackets around the expression. Without the f flag, everything will be taken literally as part of the string, without evaluation.

If you want one portion of the string to be put on one line and the other portion on the next line, you may insert \n between the two portions in the string, as below:

```
In [ ]:  print(f"it will take about {round(300/110)} hours \nto
           drive from Edmonton to Calgary")

Out [ ]: it will take about 3 hours
         to drive from Edmonton to Calgary
```

In addition to \n, other escape sequences that we discussed previously may also be included in a string.

There may be times that you want to include an escape sequence such as \n in a string as is. To achieve that effect, you either use the r flag before the opening double quotation mark or use another backslash \ before the escape sequence to cancel the first backslash (escape), as shown in the following example:

```
In [ ]:  print(r"there will be no escape \n to newline")   #
           using the r flag
         print("there will be no escape \\n to newline")   #
           using double backslash

Out [ ]: there will be no escape \n to newline
         there will be no escape \n to newline
```

Without the r flag, the sentence will be printed on two lines, as shown in the following example:

```
In [ ]:  print("there will be no escape \nto newline")

Out [ ]: there will be no escape
         to newline
```

Normally, a \n (newline) will be automatically appended to the output from each ***print*** statement by default, so that the output from the next ***print*** statement will print on a new line. If you want the output from a ***print*** statement to end with something else rather than a new line, you may use the end keyword argument to specify how the output should be ended. In the following

https://doi.org/10.15215/remix/9781998944088.01

example, output from the first ***print*** statement will be ended with a whitespace, so that outputs from the two ***print*** statements will be on the same line:

```
In [ ]:  print("this is my first print statement.", end=" ")
         print("it will take about ", round(300/110), "hours to
           drive from Edmonton to Calgary")
```

```
Out [ ]: this is my first print statement. it will take about 3 hours to drive from
           Edmonton to Calgary
```

Formally, a ***print*** statement may take the following form when used:

```
print(value0, value1, value2,…, sep=' ', end = '\n', file
= sys.stdout, flush = False)
```

where value0, value1,... are values to be printed to a file stream; optional keyword argument sep defines what is used to separate value0, value1,...; optional keyword argument end tells how the output from the ***print*** statement should end; optional keyword argument file defines what file stream the output should be printed on; and optional keyword argument flush tells how the output should be flushed. The meanings and purposes of these optional keyword arguments are explained in Table 2-16.

Table 2-16: Key arguments of the *print* statement

Keyword argument	Values that can be taken	Default value
sep	it takes a string as its argument and inserts it between values	a space
end	it also takes a string as argument but appends it to the last value of the output stream	a new line
file	it is a file handle such as that returned by an ***open*** statement	sys.stdout
flush	it takes a Boolean value (True or False) indicating whether to forcibly flush the output stream	False

Please note that a program may need to output different types of data not only correctly but also nicely. In 5.1, we will learn how to construct well-formulated strings from various types of data.

https://doi.org/10.15215/remix/9781998944088.01

INPUT STATEMENT

The ***input*** statement is another important one you must learn and use correctly and effectively. Contrary to the ***print*** statement, the ***input*** statement is used to get information from users through the keyboard, as shown in the following example in Jupyter Notebook:

```
In [ ]:  your_age = input("Please tell me your age:")
         print("Now I know your age is", your_age)

Out [ ]: Please tell me your age: 39
         Now I know your age is 39
```

Please note that everything taken from users through the ***input*** statement is treated as a string. If you are expecting a number, such as an integer, you must convert the string into its respective data type, as shown in the following example:

```
In [ ]:  your_age = input("Please tell me your age:")
         your_age = int(your_age)
         print(f"In 50 years your age will be {your_age + 50}")

Out [ ]: Please tell me your age: 39
         In 50 years, your age will be 89
```

As you may have guessed already, the ***input*** statement takes one argument as a prompt to tell users what to do and what the program is expecting from the user.

If you want to provide more detailed instructions in the prompt to the user, you may use the triple quotation marks to include multiple lines of instruction as prompt:

```
In [ ]:  sl = input("""Your choices
         A: to get the average mark
         M: to get the mean of all marks
         H: to get the highest mark
         L: to get the lowest mark
         Q: to exit the program
         Please select___""")

Out [ ]: Your choices
         A: to get average the mark
         M: to get the mean of all marks
         H: to get the highest mark
         L: to get the lowest mark
         Q: to exit the program
         Please select___
```

https://doi.org/10.15215/remix/9781998944088.01

As you can see, this can be a good way to make a menu for some terminal-based applications.

ASSERT STATEMENT

The ***assert*** statement is used to test if a condition is true in a program. It may take one of two forms, as shown in Table 2-17.

Table 2-17: Semantics of *assert* statement

Syntax	Meaning
assert <condition>	if <condition> is false, the program will stop and raise AssertionError if <condition> is True, the program will run ahead
assert <condition>, <error message>	if <condition> is false, the program will stop and raise AssertionError, along with <error message>

The assertion statement is very useful in debugging your programs, because it can be used to check the value of a variable or a certain condition of the program. If the condition is not met as expected, the program would stop and let you check what's going on, as shown in the following examples:

```
In [ ]:  def average_incomes(incomes):
           assert len(incomes) != 0, "Error: need at least one
         income"
           return sum(incomes) / len(incomes)

         incomes = [35892, 13487, 56852, 135278, 87542]
         print("Average of incomes is",average_incomes(incomes))
         incomes = []
         print("Average of incomes is",average_incomes(incomes))
Out [ ]: Average of incomes is 65810.2
         ---------------------------------
         AssertionError Traceback (most recent call last)
         <ipython-input-2-201cb13363c6> in <module>
         6 print("Average of incomes is",average_incomes(incomes))
         7 incomes = [] --
         8 print("Average of incomes is","average_incomes(incomes))

         <ipython-input-2-201cb13363c6> in average_incomes(incomes)
         1 def average_incomes(incomes):
         -- 2 assert len(incomes) != 0, "Error: there must be at least one income"
         3 return sum(incomes)/len(incomes)
         4
         5 incomes = [35892, 13487, 56852, 135278, 87542]

         AssertionError: Error: there must be at least one income
```

https://doi.org/10.15215/remix/9781998944088.01

PASS STATEMENT

As the name implies, this statement does nothing. It is used as a placeholder in places where you don't have the actual code yet. As an example, assume you have the class Student in your design for a project, but the implementation details of the class, except the name, are yet to be worked out. You can use the ***pass*** statement to hold the place of the details, as shown below:

```
In [ ]:   class Student():
            pass

          s1 = Student()
          s2 = Student()
```

With the ***pass*** statement in place, this piece of code can run as part of a big program without raising an exception.

Note that a ***pass*** statement won't let you get out of a loop. You will need to use the ***break*** statement to get out of a loop.

DEL STATEMENT

This statement is used to delete an object. Because Python treats everything as an object, you can use this statement to delete everything you've defined.

```
In [ ]:   grade = 99
          print(f"grade = {grade}")

          del grade
          print(f"grade = {grade}")

Out [ ]:  grade = 99
          ---------------------------------
          NameError Traceback (most recent call last)
          <ipython-input-9-2b1bea6f987e> in <module>
          3
          4 del grade
          -- 5 print(f"grade = {grade}")
          NameError: name 'grade' is not defined
```

Deleting an object will free up the memory locations occupied by the object. Computer memory is a precious resource in computing. Python objects, even objects of built-in data types—such as list, tuple, set, and dictionary—can take up a great deal of memory. Deleting the objects that are no longer used will free up memory and make it available for other objects and other applications.

https://doi.org/10.15215/remix/9781998944088.01

RETURN STATEMENT

The ***return*** statement is one of the most important statements in Python (and some other languages as well). It is used to return a value from a function—a very important construct of all programs. The following is an example:

```
In [ ]:  def cube(n):
           return n ** 3

         print(f'The cube of 23 is {cube(23)}')
```

```
Out [ ]: The cube of 23 is 12167
```

This function simply takes a number and returns n to the power of 3.

Please note that in Python, the ***return*** statement doesn't have parentheses around the value to be returned. Even if you want to return multiple values from a function, you only need to use commas to separate the values behind the ***return*** keyword. The ***return*** statement will automatically pack all the values in a tuple and then return them.

In the following example, we define a function of a modular operation but return the quotient and the remainder at the same time:

```
In [ ]:  def modular(a, b):
           assert b != 0, 'Zero divisor'
           return a // b, a % b

         print(modular(13, 6))
```

```
Out [ ]: (2, 1)
```

OPEN STATEMENT

The ***open*** statement is used to open a file for writing, reading, appending, or updating (reading and writing). The following is an example:

```
f = open("c:\\workbench\\myprimes.txt", 'r')
```

This opens the file specified by c:\\workbench\\myprimes.txt and creates a file object ready for reading. Reading is the default mode when you open a file without the second argument. Hence the following statement does the same as the above statement:

```
f = open("c:\\workbench\\myprimes.txt")
```

https://doi.org/10.15215/remix/9781998944088.01

To open a file for writing, w is used for the second argument:

```
f = open("c:\\workbench\\myprimes.txt", 'w')
```

When a file is opened for writing, the old data will be overwritten if there is already data in the file. To keep the old data and append new data to the file, use a for the second argument instead:

```
f = open("c:\\workbench\\myprimes.txt", 'a')
```

If you want to create a file only if the file doesn't exist, use x for the second argument, to mean exclusive creation of the file:

```
f = open("c:\\workbench\\myprimes.txt", 'x')
```

This would avoid accidentally overwriting a file.

By default, data written to a file opened with w, a, or x is text. The data on a file can also be in binary format. To explicitly indicate whether data on a file are or should be text or binary, you can use t or b with r, w, a, or x, as shown in the following examples:

```
f3 = open("c:\\workbench\\mykey.dat", 'bw')
f5 = open("c:\\workbench\\mykey.dat", 'br')
```

YIELD STATEMENT

The ***yield*** statement is used in place of the ***return*** statement in some special circumstances when defining a function. When the ***yield*** statement is used in defining a function, the function becomes a generator in Python terms, as shown in the following example:

```
In [ ]:  def odds(n):
           for i in range(n):
             yield 2 * i + 1 # yield makes the function a generator

         odd_numbers = odds(12)
         print(f"type of object odd_numbers is
          {type(odd_numbers)}")
         for i in odd_numbers:
           print(i, end = ' ')

Out [ ]: type of object odd_numbers is <class 'generator'>
         1 3 5 7 9 11 13 15 17 19 21 23
```

https://doi.org/10.15215/remix/9781998944088.01

When we say a function becomes a generator, we mean that an object of the generator class is returned from the function.

What is a generator object? For now, you may consider it a dynamic list whose members are generated and used dynamically on the fly, without using a big bunch of memory to store the whole list. The following is an example of a generator:

```
Object_generator = (x ** 3 for x in range(13))
```

If we run

```
for i in Object_generator:
  print(i)
```

we will see the following members:

0, 1, 8, 27, 64, 125, 216, 343, 512, 729, 1000, 1331, 1728,

However, if we try to get the length of the generator object with the following statement:

```
print(f"{len(object_generator)}")
```

we will get the following error:

TypeError: object of type 'generator' has no len()

This confirms that an object of type generator has no length.

RAISE STATEMENT

When some errors occur in a Python program, exceptions will be automatically raised and the program will stop running, unless the exception is handled with the ***try-except*** statement. Such errors include operations deemed illegal by Python. In some cases, an exception needs to be explicitly applied when a certain condition is met. In the previous section, we saw how an exception can be raised with the ***assert*** statement. In the following example, we show how to raise an exception with the ***raise*** statement.

https://doi.org/10.15215/remix/9781998944088.01

```
In [ ]:  total = 0
         for i in range(39):
           mark = int(input("Tell me a mark:"))
           if mark < 0:
             raise Exception("No negative mark is accepted!")
             total += mark
         print(f'average mark is {total / 39}')
```

```
Out [ ]: Tell me a mark: -12
         ---------------------------------
         Exception Traceback (most recent call last)
         <ipython-input-13-f4aa3f6c4326> in <module>
         3 mark = int(input("Tell me a mark:"))
         4 if mark < 0:
         -- 5 raise Exception("No negative mark is accepted!")
         6 total += mark
         7 print(f'average mark is {total / 39}')

         Exception: No negative mark is accepted!
```

This piece of code is used to calculate the average marks of 39 students, but it considers a negative mark unacceptable and will raise an exception.

BREAK STATEMENT

The ***break*** statement is used to get out of a loop and continue to the next statement. Here is an example:

```
In [ ]:  for i in range(10):
           print(i, end = " ")
           if i == 8:
             print(f"\nget out of the loop when i = {i}")
           break
```

```
Out [ ]: 0 1 2 3 4 5 6 7 8
         get out of the loop when i = 8
```

CONTINUE STATEMENT

The ***continue*** statement is used within a loop code block to continue to the next iteration of the loop and ignore the rest of the code block. This statement can be very useful if you don't want to run some statements when some condition is met.

https://doi.org/10.15215/remix/9781998944088.01

IMPORT STATEMENT

The ***import*** statement is used to import modules into a program file or a specific class within the module. The following is an example of how to import the standard math module into the program:

```
In [ ]:  import math
         print(f"98 ** 3 = {math.pow(98, 3)}")
```

```
Out [ ]: 98 ** 3 = 941192.0
```

GLOBAL STATEMENT

A ***global*** statement simply declares, within a code block such as function or class, that some identifiers/names such as variables should be treated as globally writable. Without a ***global*** statement, a variable defined outside of a function or class may be read, but writing to the variable will raise an exception, as shown in the following examples:

```
In [ ]:  gravity = 9.807   # gravity on the earth's surface,
          global variable
         r_earth = 6371000   # the earth's mean radius 6371000
          metres

         def changed_gravity(m):   # m is the distance from the
          earth
           gravity = gravity * (r_earth/(r_earth+m)) ** 2   #
          decrease gravity by 2
           return gravity

         print(f'gravity at 99999 metres above sea level is
          {changed_gravity(99999)} or {gravity}')
```

```
Out [ ]: ---------------------------------
         UnboundLocalError Traceback (most recent call last)
         <ipython-input-36-7f10379e5fb0> in <module>
         6 return gravity
         7
         -- 8 print(g_force(99999))
         <ipython-input-36-7f10379e5fb0> in g_force(m)
         3
         4 def changed_gravity(m):
         -- 5 gravity = gravity * (r_earth/(r_earth+m))**2
         6 return gravity
         7 UnboundLocalError: local variable 'gravity' referenced before assignment
```

https://doi.org/10.15215/remix/9781998944088.01

In the code above, gravity is first introduced outside the function definition. It became a global variable by the rule of naming scopes. Inside the definition of function changed_gravity, the name was defined again by putting it on the left side of the assignment, but only locally by default, according to the rules. However, this local variable is used on the right side of the same assignment statement. That is how the exception has occurred.

Since what we actually want is to use the globally defined variable gravity on both sides of the assignment statement within the function definition, we need to explicitly declare that, as shown in the following example revised from above:

In [ ]:
```
gravity = 9.807   # gravity on the earth's surface,
  global variable
r_earth = 6371000   # the earth's mean radius is
  6371000 metres

def changed_gravity(m):   # m is the distance from the
  earth
   global gravity
   gravity = gravity * (r_earth / (r_earth+m)) ** 2
   return gravity

print(f'gravity at 99999 metres above sea level is
  {changed_gravity(99999)} or {gravity}')
```

Out [ ]:
```
gravity at 99999 metres above sea level is 9.506238807104731 or
  9.506238807104731
```

As you can see, the value of global variable gravity has now been changed within the function.

NONLOCAL STATEMENT

We have seen global variables and local variables, and how global variables are accessible globally, whereas local variables are only accessible within a local scope such as a function. There is something between global and local called nonlocal. The ***nonlocal*** statement can be used to declare a list of variables that are not local but refer to variables defined in the nearest enclosing scope but excluding global variables. This may happen when defining a function within another function, as shown in the following example:

https://doi.org/10.15215/remix/9781998944088.01

```
In [ ]:  def outer_logger():
           event = 'Something has happened.'

         def inner_logger():
           nonlocal event
           event += "Something else has happened as well."
           print("inner log:", event)

         inner_logger()
           print("outer log:", event)

         outer_logger()
```

```
Out [ ]: inner log: Something has happened. Something else has happened
           as well.
         outer log: Something has happened. Something else has happened as
           well.
```

HELP STATEMENT

The ***help*** statement is used to invoke a helping system and is often used in Python interactive mode to get help on modules, statements, functions, or methods.

Code sample in Python interactive mode

```
>>> help(str)
Help on class str in module builtins:

 class str(object)
 | str(object = '') -> str
 | str(bytes_or_buffer[, encoding[, errors]]) -> str
 |
 | Create a new string object from the given object. If
the encoding or
 | any errors are specified, then the object must expose a
data buffer
 | that will be decoded using the given encoding and error
handler.
 | Otherwise, returns the result of object.__str__() (if
defined)
 | or repr(object).
 | encoding defaults to sys.getdefaultencoding().
 | errors defaults to 'strict'.
 |
 | Methods defined here:
 |
```

https://doi.org/10.15215/remix/9781998944088.01

```
18  | __add__(self, value, /)
19  | Return self+value.
20  |
21  | __contains__(self, key, /)
22  | Return key in self.
23  |
24  | __eq__(self, value, /)
25  | Return self==value.
26  |
27  | __format__(self, format_spec, /)
28  | Return a formatted version of the string as described
    by format_spec.
29  |
30  |
31  | __ge__(self, value, /)
32  -- More --
```

Compound Statements

In the previous section, we studied individual statements that can be used in programming with Python. In this section, we study compound statements and ways to make various compound statements in Python.

In Python, a compound statement consists of at least one clause, and each clause is made of a header and a suite, or code block. A header starts with a keyword such as ***if***, ***for***, ***while***, ***class***, ***def***, ***try***, ***else***, ***except***, ***finally***, and so on and ends with a colon :, as described below:

```
<header>:
  <code block>
```

What can be on the header line depends on the keyword leading the header. You will learn more about this in the following chapters.

CODE BLOCKS

In programs, some statements are grouped and run in sequence as a unit or a suite. We call such a group of statements a code block.

Unlike C, C++, Java, and some other languages that use curly brackets to make code blocks, Python uses indentation to form code blocks. In Python, a program can have multiple code blocks, and code blocks can be nested with proper indentation. Statements intended to be in the same code block must use the same indentation. The following is an example:

https://doi.org/10.15215/remix/9781998944088.01

Code sample

```
1  i, s = 1, 1   # the first statement must be started at
   the very beginning
2
3  while I <= 100:   # this compound statement is in the
   same code block
4     s *= i   # this is the first statement in the code
   block/suite
5     i += 1   # this is the second statement in the code
   block/suite
6
7  print("the product of 1x2x3 … 100 is ", s)
```

The sample program above has two simple statements on lines 1 and 7, and one compound statement on lines 3 to 5. The header of the compound statement begins with the keyword ***while***, and its suite is a code block that consists of two simple statements. Because statements on lines 1, 3, and 7 are in the same code block, they must be indented the same, whereas statements on lines 4 and 5 must be further indented to form a code block as a suite for the ***while*** compound statement.

RULES OF INDENTATION

To ensure that your programs are properly indented, follow the following rules:

1. The first line of code of a program must start at the very first column of line, though there can be some blank lines before the first line of code, for better readability, if you like.
2. All lines of code in the same code block must be indented the same.
3. The suite of a compound statement must be indented further than the header of the compound statement.
4. All code blocks that are at the same level must use the same indentation.
5. All lines of code in the same suite must use the same indentation.

RULES OF SPACING

The rules of spacing are about how to space out words within a line of script or code and how to space lines of scripts. Some of the rules must be followed, while other rules are for readability or are merely convention among Python programmers:

1. There must be at least one space between two words.
2. As a convention, there should be only one space between two words.

https://doi.org/10.15215/remix/9781998944088.01

3. Also as a convention, there should be one space before each operator and one space behind each operator in an expression. So x>y should be written as x > y.
4. For better readability, there should be no space between a unary negation operator (–) and the term it negates. So - x should be written as -x.
5. Also for readability, in a function call, there should be no space between a function name and the list of parameters. So abs (y) should be written as abs(y).
6. The same goes for definitions of functions. There should be no space between the function name and the list of arguments.
7. There should be no blank lines between lines of simple statements if they are intended to be in the same code block.
8. For better readability, there should be a blank line between simple statement(s) and compound statements if they are in the same code block, as shown in the following sample code:

Code sample

```
1  i, s = 1, 1   # first statement must be started at the
     very beginning
2
3  while I <= 100:   # this compound statement is in the same
     code block
4    s *= i   # statement must be indented
5    i += 1   # second statement in the code block/suite
6
7  print("the product of 1x2x3 … 100 is ", s)
8
```

Please note the blank line between line 1 and line 3, as well as between lines 5 and 7.

IF STATEMENT

An ***if*** statement is used to run a block of statements under a condition. The header of an ***if*** statement begins with the keyword ***if***, followed by a logical expression of a condition, and then a colon, as shown below:

```
if <condition>:
    <suite or code block>
```

https://doi.org/10.15215/remix/9781998944088.01

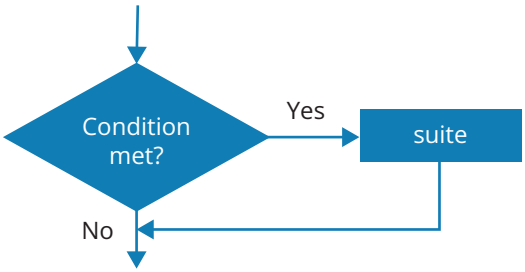


Figure 2-6: Flowchart of an ***if*** statement

Here is an example:

Code sample

```
1  mark = float(input("Please input your mark:"))
2
3  if mark >= 0:
4    print(f"The mark is {mark}.")
5
```

Note that although Python allows the suite to be on the same line as the header, as shown in the following sample, for readability, that is not preferable.

```
if mark >= 0: print(f"The mark is {mark}.") # allowed but not preferred
```

IF-ELSE STATEMENT

In the example above, the ***if*** statement can only make one selection. To do something in particular if the condition is not met, the ***else*** clause can be added. The syntax of ***if-else*** statement is shown below, and the corresponding flowchart is shown in Figure 2-7.

```
if <condition>:
  <code block 1>
else:
  <code block 2>
```

https://doi.org/10.15215/remix/9781998944088.01

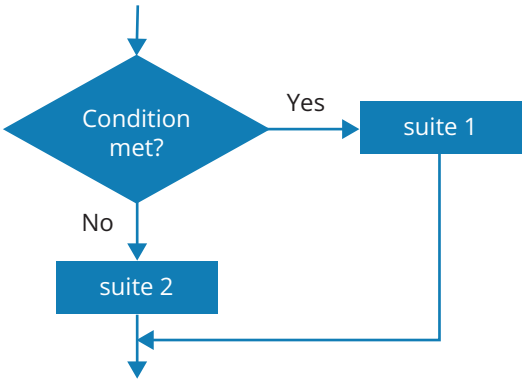


Figure 2-7: Flowchart of the ***if-else*** statement

The code sample shown above can be rewritten as follows by adding an ***else*** clause:

Code sample

```
1  mark = float(input("please input your mark:"))
2
3  if mark >= 0:>
4     print(f"The mark is {mark}.")
5  else:
6     print("Incorrect input! A mark cannot be a negative
   number.")
7
```

IF-ELIF STATEMENT

The ***if-else*** statement can only handle double selections. How can we handle multiple selections in Python? For example, in addition to telling whether a mark is legitimate or not, we may also want to convert the percentage mark to a letter grade. In Python, that can be done with an ***if-elif*** or ***if-elif-else*** statement. The syntax of the ***if-elif*** statement is shown below:

```
if <condition 1>:
  < suite 1 >
elif <condition 2>:
  < suite 2 >
elif <condition 3>:
  < suite 3 >
…
```

https://doi.org/10.15215/remix/9781998944088.01

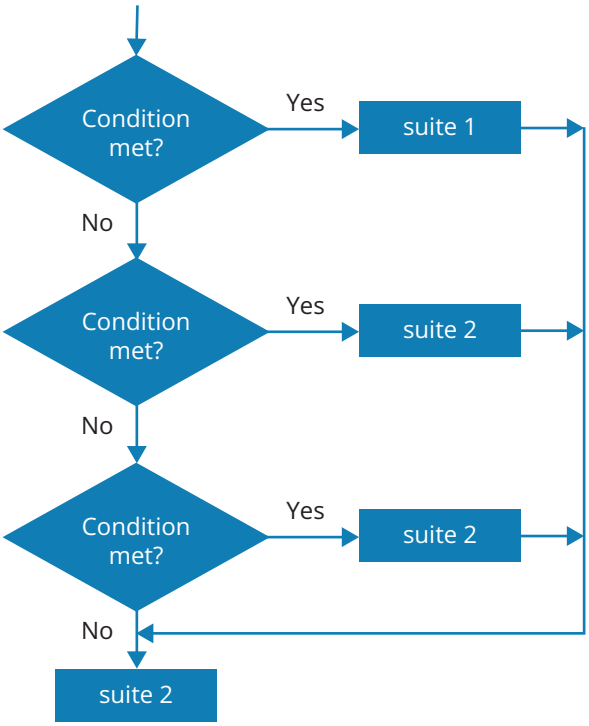


Figure 2-8: Flowchart of the ***if-elif-elif-…***

The code that can tell the letter grade from a percentage grade is shown below:

Code sample

```
 number_grade = round(float(input("please tell me a
  numeric grade between 0 and 100:")))
if number_grade >= 90:
  print(f"alpha/letter grade for {number_grade}% is A+")
elif number_grade >= 85:
  print(f"alpha/letter grade for {number_grade}% is A")
elif number_grade >= 80:
  print(f"alpha/letter grade for {number_grade}% is A-")
elif number_grade >= 76:
  print(f"alpha/letter grade for {number_grade}% is B+")
elif number_grade >= 73:
  print(f"alpha/letter grade for {number_grade}% is B")
elif number_grade >= 70:
  print(f"alpha/letter grade for {number_grade}% is B-")
elif number_grade >= 67:
  print(f"alpha/letter grade for {number_grade}% is C+")
elif number_grade >= 64:
  print(f"alpha/letter grade for {number_grade}% is C")
```

https://doi.org/10.15215/remix/9781998944088.01

```
elif number_grade >= 60:
  print(f"alpha/letter grade for {number_grade}% is C-")
elif number_grade >= 55:
  print(f"alpha/letter grade for {number_grade}% is D+")
elif number_grade >= 50:
  print(f"alpha/letter grade for {number_grade}% is D")
elif number_grade >= 0:
  print(f"alpha/letter grade for {number_grade}% is F")
else:
  print("Numeric grade must be a positive integer!")
```

IF-ELIF-ELSE STATEMENT

An ***else*** clause can be added to the end of an ***if-elif*** statement in case something special needs to be done if all the conditions are not met.

WHILE STATEMENT

The ***while*** statement is used to run a block of code repeatedly as long as a given condition is met. The syntax of the statement is as follows:

```
while <condition>:
  < a suite >
```

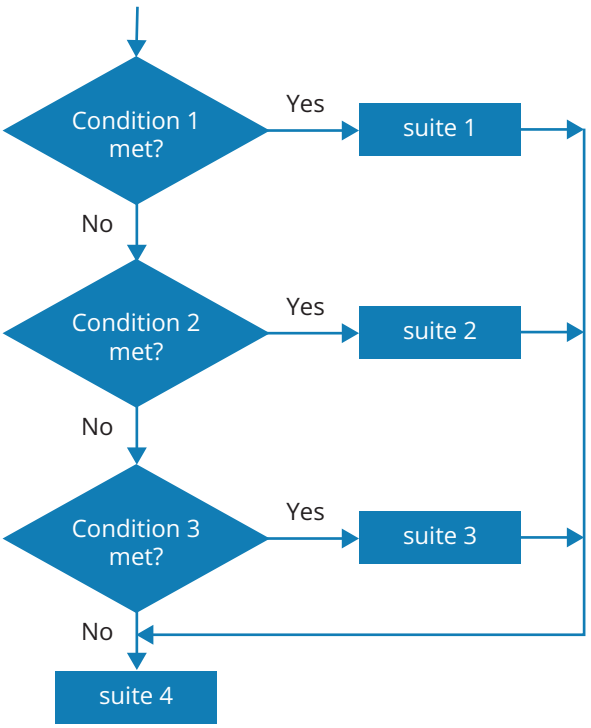


Figure 2-9: Flowchart of an ***if-elif-…elif-else*** statement

https://doi.org/10.15215/remix/9781998944088.01

The following is an example:

Code sample

```
1  i = 1
2
3  while i <= 10:
4     print(f"I am counting to {i}")
5     i += 1
```

The loop is ended when not (I <= 10). The ***while*** statement is more advantageous when used to form a loop if we only know when the loop should end, as shown in the following word-guessing program:

Code sample

```
1  cnt, your_guess = 0, ""
2
3  while (your_guess.lower()) != "python":
4     your_guess = input("Guess which programming language
    is my favourite: ")
5
6     cnt += 1
7  print(f"Congratulations! You got it in just {cnt} guesses")
```

FOR STATEMENT

A ***for*** statement provides another way to form a loop and is best for when the loop runs through an iterable, such as a list, a tuple, a string, a generator, a set, or even a dictionary. The syntax of the ***for*** statement is as follows:

```
for <iteration variable(s)> in <iterable>:
  < a suite >
```

Note that there can be more than one iteration variable if needed, but it is more common to have only one iteration variable.

The following is an example:

Code sample: *for* statement with a string

```
1  cnt = 0
2  my_string = "this is a secret"
3  for c in my_string:
4     print(c)
5     cnt += 1
6
7
```

https://doi.org/10.15215/remix/9781998944088.01

```
print(f"there are {cnt} characters in ({my_string})")
```

Code sample: *for* statement with a set

```
week_set = set(('Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat',
 'Sun'))
for w in week_set:
   print(w)
```

Code sample: *for* statement with a dictionary

```
week = {1: 'Mon', 2: 'Tue', 3: 'Wed', 4: 'Thu', 5: 'Fri',
 6: 'Sat', 7: 'Sun'}
for w in week.keys():
   print(week[w], " in key and value pair in dictionary")
```

DEF STATEMENT

The ***def*** statement is used to define new functions or methods if defined within a class definition. The syntax of ***def*** statement is as follows:

```
def <function_name>(<list of arguments>):
  < code block >
```

where function_name should be a unique identifier within the current scope, and the list of arguments can be empty. The details of defining and using functions will be presented in Chapter 6. For now, you need only be concerned with the definition of a simple function so that you know how the ***def*** statement is used. The function is to calculate a given number x to the power of 10, and return x*x*x*x*x*x*x*x*x*x:

```
In [ ]:  def power10(x):
           s = x
           for i in range(9):
             s *= x
             return s

         print(f'power10({2}) = {power10(2)}')
```

```
Out [ ]: 2 to the power of 10 = 1024
```

https://doi.org/10.15215/remix/9781998944088.01

CLASS STATEMENT

The ***class*** statement is used to define new classes. The syntax of defining a new class that only inherits from the base class (object) is as follows:

```
Class class_name:
  < suite >
```

or

```
Class class_name(object):
  < suite >
```

To define a new class that inherits from classes other than object, the syntax is as follows:

```
Class Class_name(<list of base classes>):
  < suite >
```

TRY-EXCEPT STATEMENT

The ***try-except*** statement is used to handle errors and exceptions, especially when certain errors are expected. The following are some common types of errors that you may encounter in your programs:

ArithmeticError	FileExistsError	LookupError
FloatingPointError	FileNotFoundError	IndexError
OverflowError	InterruptedError	KeyError
ZeroDivisionError	IsADirectoryError	MemoryError
AssertionError	NotADirectoryError	NameError
AttributeError	PermissionError	UnboundLocalError
BufferError	ProcessLookupError	BrokenPipeError
EOFError	TimeoutError	ConnectionAbortedError
ImportError	ReferenceError	ConnectionRefusedError
ModuleNotFoundError	RuntimeError	ConnectionResetError

Refer to https://docs.python.org/3/library/exceptions.html for a detailed discussion about the exceptions and error types defined in Python.

https://doi.org/10.15215/remix/9781998944088.01

The following is an example showing how to handle errors from user input that use 0 for the denominator.

Code sample: *for* statement with dictionary

```
try:
   a = int(input("give me a number:"))
   b = int(input("give me another number:"))
   print(f"{a} / {b} = {a / b}")
except ZeroDivisionError:
   print(f"incorrect second input {b}!")

```

The details of error and exception handling in programs will be discussed in Chapter 4.

WITH STATEMENT

The ***with*** statement is used to provide a context for the execution of a code block. The mechanism is a bit complex, but the following may provide some help. Remember that the ***with*** statement works only on objects that have special methods __enter__() and __exit__() implemented in accordance with Python context management protocol (PEP 343). For the mechanisms behind the ***with*** statement, read https://effbot.org/zone/python-with-statement.htm or search the internet for more details.

The general syntax of the ***with*** statement is as follows:

```
With <expression of object> as < variable referring to the object>:
  <suite>
```

where the value of <expression of the object> will be an object on which the context management protocol has been implemented and the <variable referring to the resulted object> will often be used in the suite. A good and common example of using the ***with*** statement is dealing files, which, when opened, are objects with context management protocol implemented. The following is an example:

Code sample: *with* statement

```
"""
This program will get an input of a big integer from
 user, and
find all the prime numbers not greater than the integer
 input from the user, and
```

https://doi.org/10.15215/remix/9781998944088.01

```
 write all the prime numbers into a text file.
"""
 m = int(input("""
    Please tell me a big integer number and
    then I can find all the primes less than
    the prime number:"""))
 import math as mt
 with open("c:\\workbench\\myprimes.txt", 'w') as p:
    i = 2
    while i <= m:
      flag = True
      j = 2
      while j <= int(mt.pow(i,1/2)) + 1 and flag:
        if i % j == 0:
          flag = False
          j += 1
        if flag:
          p.write(str(i)+ "\n")
          i += 1
```

Chapter Summary

- Vocabulary is important for any language, and even more important for computer languages, because computers will not understand your programs at all if you have used the wrong vocabulary.
- For programming languages, including Python, vocabulary includes various types of data, operators, built-in functions, reserved words (including keywords), and variables identified by user-defined names (also called identifiers).
- Identifiers must begin with a letter or underscore in the ASCII table, then be followed by letters, digits, and/or an underscore.
- Identifiers in Python are case-sensitive, which means that A1 and a1 are two different identifiers.
- Within the Python community, there are conventions for how identifiers should be made and used for identifying different things in Python programs.
- Simple data types include integer numbers, float numbers, Boolean numbers, and complex numbers.
- Complex numbers are represented as a + bj or a − bj, where a and b are integers or float numbers.
- Compound data are made of other data that can be of two types: simple or compound.

https://doi.org/10.15215/remix/9781998944088.01

- A string is a sequence of characters within a pair of single or double quotation marks.
- Some special characters in a string must be represented using an escape sequence, such as \n for newline, \t for tab, and \\ for a backslash, and so on.
- In a string, all characters are indexed starting from 0, so that each individual character can be accessed using its index.
- There are many functions available to manipulate strings.
- There are three ways of formatting strings: using placeholders, using the format method, and using the f prefix before a string. The last one is preferred.
- A list is a sequence of data within a pair of square brackets.
- Members of a list are also indexed, and each individual member can be accessed through its index.
- A tuple is a sequence of data within a pair of parentheses.
- Members of a tuple are also indexed, and each individual member can also be accessed through its index.
- While individual members of a list can be deleted or changed, individual members in a tuple cannot be deleted or changed.
- A set is a collection of data within a pair of curly braces.
- Members in a set are not indexed, so individual members in a set cannot be accessed through the index.
- A dictionary is a collection of key-value pairs within a pair of curly braces.
- Keys are used to access the values of a dictionary.
- In Python, everything can be treated as an object.

Exercises

1. Indicate which of the following are not legitimate Python identifiers to name variables, functions/methods, and classes, and explain why.

This	3da	My_name	for	i9	vote
$s	_sum_	cLearance	method	lists	t5#

2. Write a single statement to complete each of the following tasks:
   a. Read an integer from user into variable k.
   b. Print a multiple-line mailing label with your name and home address, including the postal code.
   c. Print the area of a circle with a radius of 13.

https://doi.org/10.15215/remix/9781998944088.01

   d. Assign the cube of 23 to variable x.
   e. Print the square of 12, 25, and 56, respectively, on one line.
3. Evaluate the following expressions
   a. 23 + 16 / 2
   b. round(78.3)
   c. pow(2,3) + 5
   d. sum([1,3,5,7,9])/13
   e. bin(9)
   f. divmod(13, 5)
   g. int(38.6)//7
   h. (3.5 + 6.7j) + (5.3 + 12.9j)
   i. 'Well' * 3 + '!'
4. Mentally run the following code blocks and state what each code block will display.
   a.
```
x = 5
y = 6
print(x + y)
```
   b.
```
m = 5
k = 3
print(f'{m}**{k} = {m**k}')
```
   c.
```
m, k = 35, 12
m //= k
print(m)
```
   d.
```
m, k = 35, 12
m %= k
print(m)
```

Projects

1. Write a program to read a float number from the user into variable s, then calculate and print the area of a square with s as the length of its side.
2. Write a program to read two numbers and calculate the product of the two numbers.
3. A parking lot charges $2.50 per hour. Write a program to read the number of hours a vehicle has parked, then calculate and print the total to be paid for parking.
4. The arithmetic mean of several numbers is the sum of all these numbers divided by the number of these numbers. For this project,

https://doi.org/10.15215/remix/9781998944088.01

write a program that will generate three numbers from users and calculate and display the arithmetic mean of these three numbers.

5. A cube has 6 faces and 12 edges, all of the same length. Write a program that takes a number from the user as the length of an edge and calculate and display the total surface area of the cube.

https://doi.org/10.15215/remix/9781998944088.01

This page intentionally left blank

Chapter 3

Flow Control of Statements

If you praise computers for their diligence when they iterate operations tirelessly trillions of trillions of times, you must also appreciate their intelligence when they do certain things only if certain conditions are met, because decision making is important for all intelligent beings, including modern computers. All computer programming languages provide constructs for decision making—to run a statement or a block of statements only under certain conditions. Python does the same.

In Chapter 3, you will learn how to use the ***if***, ***if-else***, ***if-elif***, and ***if-elif-else*** statements to instruct computers to do certain things only under certain conditions.

Learning Objectives

After completing this chapter, you should be able to

- use an ***if*** statement to run a code block only under a set condition.
- use ***if-else*** to run two code blocks under two different conditions.
- use ***if-elif*** to make multiple selections.
- use ***if-elif-else*** to make multiple selections.
- use ***for*** statements to make loops to run code blocks repeatedly.
- use ***while*** statements correctly and efficiently to put a code block in a loop.
- use ***break*** and ***continue*** statements correctly to change the flow of program within the code block.

3.1 Selective with the *if* Statement

In Python, all selections are done with the ***if*** statement, which can take multiple forms. The following is a code sample showing how ***if*** is used to make a single selection.

https://doi.org/10.15215/remix/9781998944088.01

Code sample in Python interactive mode

```
"""
Code sample showing how to use an if statement to have a
  code block run under a certain condition
this piece of code is used to calculate the square root of
  number n only if it is a positive number.
"""
n = input('Tell me a number and I will tell you the square
  root:')
n = float(n) # convert n from string to float
if n >= 0:
   print(f"The square root of {n} is {n ** (1 / 2)}")


```

Note that a code block for ***if*** statements must begin on the next line after a colon : and be properly indented, as shown below:

```
n = int(input("n = ?"))
if n >= 0:
  Code block
```

Note: Each code block must be properly indented to indicate what the code block belongs to.

The conditions for an ***if*** statement can be any Boolean expression, as discussed in 2.1.

3.2 Single-branch selective with *if* Statement

In the above example, the program specifies only what to do when n >= 0 but does not say what to do otherwise. With Python, you can further specify what can be done if n >= 0 is not true. The following is a revised code sample—it simply tells the user to input a positive number.

Code sample in Python interactive mode

```
"""
Code sample showing how to use an if statement have a code
  block run under a certain condition
this piece of code is used to calculate the square root of
  number n only if it is a positive number
"""
```

https://doi.org/10.15215/remix/9781998944088.01

```
5
6  n = float(input('Tell me a number and I will calculate the
     square root for you:'))
7  if n >= 0:
8    print(f"The square root of {n} is {n ** (1 / 2)}")
9  else:
10   print ("Please give me a positive number!")
```

Note that no condition needs to be specified for an ***else*** clause because it implies that the condition is the negation of the condition for that ***if*** clause—that is, not n >= 0.

With the ***if*** and ***if-elif*** statements studied above, you can make single- or two-branch selections, which are depicted in Figures 3-1 and 3-2.

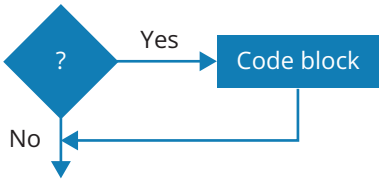


Figure 3-1: Flowchart of an ***if*** statement

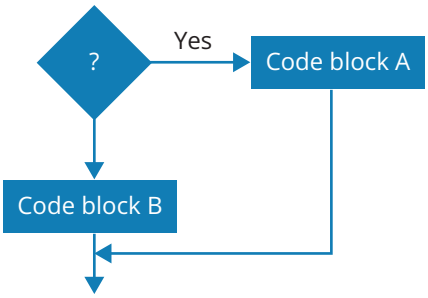


Figure 3-2: Flowchart of an ***if-else*** statement

3.3 Multiple-branch selective with *if-elif-…* and *if-elif-…-else* Statements

In decision making, there can be multiple options, each of which requires different actions to be taken. In Python, multiple selections can be made with ***if-elif*** and ***if-elif-else*** statements. The logic flow of these two statements is depicted in the following diagrams.

The flowchart in Figure 3-3 shows the logic flow of an ***if-elif-elif…elif*** statement without ***else***, which can be used to make multiple selections. The flowchart in Figure 3-4 illustrates the logic flow of an ***if-elif-elif…else*** statement for multiple selections.

https://doi.org/10.15215/remix/9781998944088.01

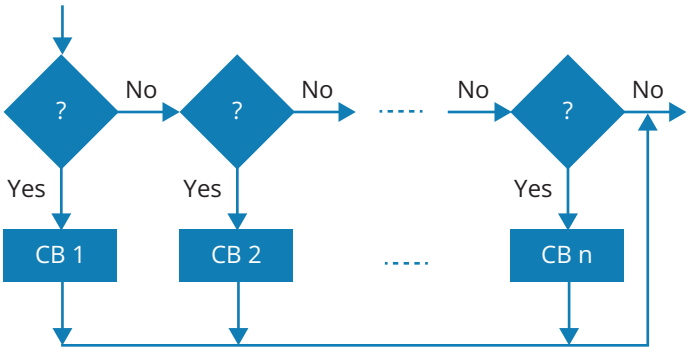


Figure 3-3: Flowchart of an ***if-elif-elif-elif…*** statement

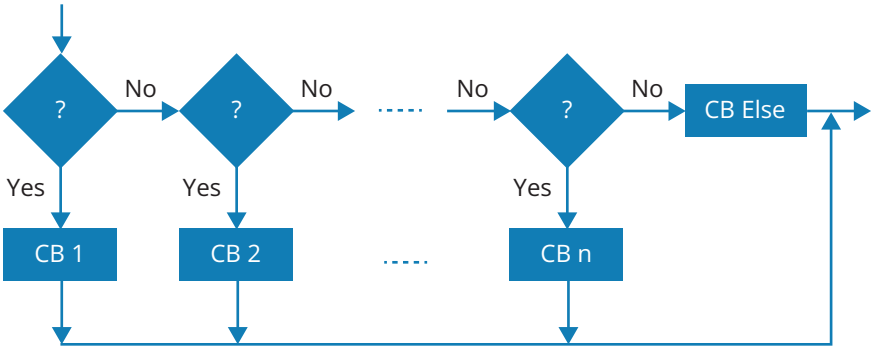


Figure 3-4: Flowchart of an ***if-elif-elif-else*** statement

Now, you are ready to tackle a real problem: to decide what letter grade should be assigned for a given numeric mark of a student in a course, according to Table 3-1.

Table 3-1: Assignment of letter grade to numeric grade

Number of selections needed	Numeric grade (%)	Alpha/letter grade
1	90–100	A+
2	85–89	A
3	80–84	A–
4	76–79	B+
5	73–75	B
6	70–72	B–
7	67–69	C+
8	64–66	C
9	60–63	C–
10	55–59	D+
11	50–54	D
12	0–49	F

https://doi.org/10.15215/remix/9781998944088.01

The table shows only integer numeric grades, but decimal inputs, automatically rounded to the nearest integer, are also allowed. The case study is shown in Table 3-2.

Table 3-2: Case study: How to use *if-elif-else*

The problem	In this case study, design a program using an ***if-elif-else*** statement to convert numeric grades to alpha/letter grades.
The analysis and design	As the above grade-conversion table shows, the program needs to make 12 selections. For each selection, a letter grade will be printed out when the numeric grade falls within the corresponding interval.

The code	`"""`
	`You are required to design a program using an if-elif-else statement`
	`to convert numeric grades to alpha letter/grades.`
	`"""`
	
	`number_grade = round(float(input("Please tell me a numeric grade between 0 and 100:")))`
	`if number_grade >= 90:`
	`  print(f"alpha/letter grade for {number_grade}% is A+")`
	`elif number_grade >= 85:`
	`  print(f"alpha/letter grade for {number_grade}% is A")`
	`elif number_grade >= 80:`
	`  print(f"alpha/letter grade for {number_grade}% is A-")`
	`elif number_grade >= 76:`
	`  print(f"alpha/letter grade for {number_grade}% is B+")`
	`elif number_grade >= 73:`
	`  print(f"alpha/letter grade for {number_grade}% is B")`
	`elif number_grade >= 70:`
	`  print(f"alpha/letter grade for {number_grade}% is B-")`
	`elif number_grade >= 67:`
	`  print(f"alpha/letter grade for {number_grade}% is C+")`
	`elif number_grade >= 64:`
	`  print(f"alpha/letter grade for {number_grade}% is C")`
	`elif number_grade >= 60:`
	`  print(f"alpha/letter grade for {number_grade}% is C-")`
	`elif number_grade >= 55:`
	`  print(f"alpha/letter grade for {number_grade}% is D+")`
	`elif number_grade >= 50:`
	`  print(f"alpha/letter grade for {number_grade}% is D")`
	`elif number_grade >= 0:`
	`  print(f"alpha/letter grade for {number_grade}% is F")`
	`else:`
	`  print("Numeric grade must be a positive integer!")`

The result	Please tell me a numeric grade between 0 and 100: 88 alpha/letter grade for 88 is A

https://doi.org/10.15215/remix/9781998944088.01

The code above didn't explicitly specify the upper bounds of the intervals shown in the grade conversion table because it takes advantage of the ***if-elif-elif*** statement—that is, the upper bound of the current ***if*** condition has been implicitly satisfied when the previous ***if*** condition is not satisfied, and the program flow gets into the current ***elif*** selection. Taking the first ***elif*** statement as an example, since the previous ***if*** condition is number_grade >= 90, when the condition is not satisfied and the program flow goes to the ***elif***, the number_grade must be less than 90, which is equal to number_grade <= 89, the upper bound of the first ***elif*** condition.

After you run the code on your computer, you may notice that for each conversion, you have to rerun the program to get another chance to input a numeric grade. How can you do as many conversions as you want until you tell the program to stop? You'll be able to do that after learning how to put a code block in a loop in the next chapter.

CODING TRICK

How would you specify the conditions for the ***elif*** if you began from the lowest numeric grade to make number_grade <= 49 for the ***if***?

3.4 Iterate with *for* Statement

Computers can do many amazing things. Many of these amazing things can only be done in thousands or even trillions of trillions of steps. Luckily, programmers don't need to write trillions of trillions of statements in a computer program, because computers can be instructed to run a block of statements in a loop as many times as needed without complaint. As such, a programmer must be able to correctly put code blocks in loops when programming. In this and the next section of this chapter, you will learn how to use ***for*** statements and ***while*** statements correctly and effectively to put code blocks in loops.

In Python, the ***for*** statement is one of only two statements that can be used to make loops or iterations. In previous sections, you already saw some examples using the ***for*** statement. Formally, a ***for*** loop takes the following form:

```
for <iteration variable> in <sequence to be looped through>:
  <Code Block>
```

in which ***for*** and ***in*** are keywords, and the iteration variable is used to take items one by one from the sequence, which can be a list, a tuple, a string, or an iterable object or generator, as you will see. The code block is a block of Python code to

https://doi.org/10.15215/remix/9781998944088.01

be executed in each iteration of the loop. The following example loops through a list of integers and calculates the cube of each.

```
In [ ]:   for i in range(1, 11):
            print(f'The cube of {i} is {i * i * i}')
```

```
Out [ ]: The cube of 1 is 1
         The cube of 2 is 8
         The cube of 3 is 27
         The cube of 4 is 64
         The cube of 5 is 125
         The cube of 6 is 216
         The cube of 7 is 343
         The cube of 8 is 512
         The cube of 9 is 729
         The cube of 10 is 1000
```

A flowchart describing the ***for*** loop is shown in Figure 3-5.

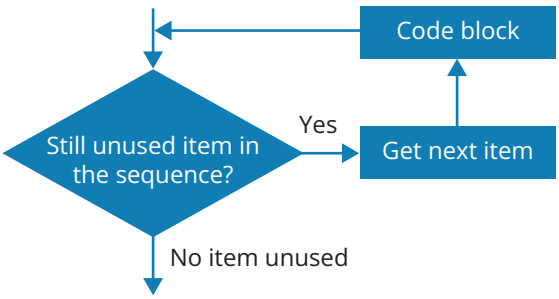


Figure 3-5: Flowchart of the ***for*** loop

Now we are ready to solve a more complex problem, shown in Table 3-3.

Table 3-3: Case study: How to display a multiplication table

The task	Print a multiplication table from 1 * 1 to 9 * 9, and the result table should be a right triangle.
Analysis and design	The resulting multiplication table should be a right triangle like this: 1 x 1 = 1 1 x 2 = 2 2 x 2 = 4 1 x 4 = 3 2 x 3 = 6 3 x 3 = 9 ... so we will need two loops: the outer one is used to loop through the row, whereas the inner one loops through the column of each row. The algorithm is as follows: Step 1: Start a row I, I = 1, 2,..., 9 Step 2: Start a column j of row I, j = 1,...i Step 3: Print j × I = j * i on the same line until I * i Step 4: Go back to step 1 and finish all rows

(continued on next page)

https://doi.org/10.15215/remix/9781998944088.01

Table 3-3: Case study: How to display a multiplication table *(continued)*

```
"""
This program is used to print a multiplication table
  from 1 x 1 to 9 x 9.
The table will be displayed nicely as a right
  triangle. It uses two loops,
an outer loop for the rows and an inner loop for the
  columns.
"""

for i in range(1, 10):
  for j in range(1, i + 1):
    print(f"{i}x{j} = {i * j}", end = "  ")
    if j == i:
      print("\n")

```

Output in terminal

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

Our next problem, in Table 3-4, is to find all the Pythagorean triples of integers less than an integer given by the user.

Table 3-4: Case study: How to find Pythagorean triples

The task

Three integers—A, B, and C—are called a Pythagorean triple when C * C = A * A + B * B.

This program will take an integer input by the user and find all the Pythagorean triples of integers less than that integer.

Analysis and design

Step 1: Get input from user
Step 2: Find all the Pythagorean triples
Step 3: Print all the triples
Step 2.0: Make an empty list, say, plist =[]
Step 2.1: One loop for A = 1…user input
Step 2.2: One loop for B = 1…user input
Step 2.3: One loop for C = 1…user input
Step 2.4: If C * C = A * A + B * B, add the triple to the list plist

```
"""
Three integers - A, B, and C - are called a
  Pythagorean triple when C * C = A * A + B * B.
```

https://doi.org/10.15215/remix/9781998944088.01

Table 3-4: Case study: How to find Pythagorean triples *(continued)*

```
  3  This program will take an integer input by the user
       and find all the Pythagorean triples of integers
       less than that integer.
  4  """
  5
  6  upper_bound = int(input("Give me a big integer:"))
  7  plist = []
  8  for i in range(1, upper_bound + 1):
  9    for j in range(i, upper_bound + 1):
 10      for k in range(j, upper_bound + 1):
 11        if k * k == j * j + i * i:
 12          plist.append((i, j, k))
 13  for i in plist:
 14    print(i)
 15
 16
 17
```

Output in terminal

```
Give me a big integer:39
(3, 4, 5)
(5, 12, 13)
(6, 8, 10)
(7, 24, 25)
(8, 15, 17)
(9, 12, 15)
(10, 24, 26)
(12, 16, 20)
(12, 35, 37)
(15, 20, 25)
(15, 36, 39)
```

A ***for*** statement can have multiple iteration variables if needed. In order to make it work, however, each item of the iteration sequence needs to have multiple values for the multiple iteration variables, as shown in the following example:

```
for i, j in [(1, 1), (2, 2), (3, 3), (4, 4), (5, 5)]:
    print(f"{i} * {j} = {i * j}")
```

or

```
for i, j in zip(range(5), range(5)):  # zip is a special
function
    print(f"{i + 1} * {j + 1} = {(i + 1) * (j + 1)}")
```

In the above, ***zip*** is a built-in function that takes an element from each iterable and forms a tuple, as shown below:

```
>>> list(zip(range(5), range(5), range(6)))
[(0, 0, 0), (1, 1, 1), (2, 2, 2), (3, 3, 3), (4, 4, 4)]
```

https://doi.org/10.15215/remix/9781998944088.01

The function ***zip*** will stop making tuples when the shortest iterable has reached the end.

Note that the following two statements are totally different:

```
for i in range(5):
  for j in range(5):
    print(f"{j + 1} * {i + 1} = {(i + 1)*(j + 1)}")z
```

versus

```
for i, j in zip(range(5), range(5)):
  print(f"{i + 1} * {j + 1} = {(i + 1)*(j + 1)}")
```

The first statement is two loops nested, whereas the second is a single loop.

You may copy and paste the code into Jupyter Notebook to find out why and how.

Using *break* and *continue* Statements and an *else* Clause Within Loops

Chapter 2 discussed what the ***break*** statement and ***continue*** statement do. Within a ***for*** loop, if you want to get out of the iteration immediately when something has occurred, you can use a ***break*** statement; if you want to go to the next item in the iteration sequence right away, you can use a ***continue*** statement.

A ***for*** statement can also have an ***else*** clause whose code block will be executed when the iteration sequence is used up. The following code example taken from the Python documentation explains how the ***break*** statement and ***else*** clause can be used on a ***for*** loop.

In [ ]:
```
for n in range(2, 10):
  for x in range(2, n):
    if n % x == 0:
      print(n, 'equals', x, '*', n // x)
      break   # if break is ever executed, the else
 code block will never be reached
    else:   # the else code block is executed when
      range(2, n) is used up without finding a factor
      print(n, 'is a prime number')
```

Out [ ]:
```
2 is a prime number
3 is a prime number
4 equals 2 * 2
5 is a prime number
6 equals 2 * 3
7 is a prime number
8 equals 2 * 4
9 equals 3 * 3
```

https://doi.org/10.15215/remix/9781998944088.01

In the example above, pay particular attention to the indentation of ***else***. The else block is treated as a clause of the inner ***for*** statement because it has the same indentation as the inner ***for***. If it were indented the same as the ***if*** statement, the ***else*** block would become part of the ***if*** statement.

Common Coding Mistakes with the *for* Loop

Because Python made the ***for*** loop to run an iteration variable or variables through a sequence with a finite number of items, it has essentially avoided some mistakes common in other languages such as C, C++, and Java. You should, however, remember to not change the value of any iteration variable within the code block of a ***for*** loop because it needs to be changed automatically by Python interpreter to the next item in the sequence. The iteration might be unexpected if the value of the iteration variable is changed in the code block.

3.5 Iterate with the *while* Statement

The ***while*** statement is another statement you can use to make loops. As discussed in Chapter 2, ***while*** is best used if you know when the loop should stop but do not know how many times the code will iterate. For the problem solved in the previous section, even though you also know when each loop should stop, it can still be coded with the ***while*** statement, as shown below:

The code

```
"""
This program is used to print a multiplication table from
 1 x 1 to 9 x 9.
The table will be displayed nicely as a right triangle.
 The two loops are coded with the while statement
instead of the for statement.
"""

 i = 1
 while i < 10:
   j = 1
   while j <= i:
     print(f"{j} x {i} = {i * j}", end ="")
     if j == i:   # if statement is used to decide when
to start a new line
       print("\n")
     j += 1
   i += 1


```

https://doi.org/10.15215/remix/9781998944088.01

While a ***for*** loop can always be replaced with a ***while*** loop, a ***while*** loop cannot be replaced with a ***for*** loop in cases where the number of iterations is unknown.

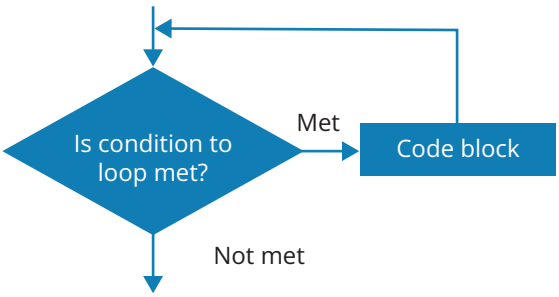


Figure 3-6: Flowchart illustrating the ***while*** loop

Consider the problem of calculating the average of the student marks taken from user input. Either the total number of marks is unknown or you do not expect the user to count how many marks they input, so it is not possible to use a ***for*** loop to solve the problem. However, you do know there will be no negative marks, which can then be used to indicate the end of the input. Hence, you can use a ***while*** loop to take input from the user until you get a negative number from the user. The program is shown in Table 3-5.

Table 3-5: Case study: How to stop a *while* loop

	Code sample in VS Code IDE
The task	Get student marks from user input and calculate the average mark, using a negative number input by the user to indicate the end of marks.
Analysis and design	You don't know how many marks will be input from the user, but you do know there will be no negative marks. You can then use a negative number, input by the user, to end the application and use the ***while*** loop to iterate. To calculate the average, you need to know the sum of all the marks as well as the total number of marks. For the former, you need to add each mark to the total, whereas for the latter, you need to use a counter to count every mark. Use the variable total to keep the sum and use the variable count to keep the number of marks. Both need to be initialized to 0. The algorithm is below: Step 1: Initialize both count and total to 0 Step 2: Take an input from user and convert it to a float number Step 3: If the number is a mark (>= 0), then do the following: Step 3.1: Increase count by 1 Step 3.2: Add mark to total Step 3.3: Get another input from user and convert to float Step 3.4: Go back to Step 3 Step 4: Calculate the average using total/count and print out the result. Step 5: End the program

https://doi.org/10.15215/remix/9781998944088.01

Table 3-5: Case study: How to stop a *while* loop *(continued)*

Code sample in VS Code IDE

```
"""
Program file name: averagemark.py
Application: this application takes marks from user
  input and calculates the average mark.
A negative number from user input is used to
  indicate the end of the program.
"""

count, total = 0, 0   # two variables are
  initialized in one assignment statement
mark = float(input("Please input a mark or a
  negative number to end:"))
while mark >= 0:
   total += mark
   count += 1
   mark = float(input("Please input next mark or a
  negative number to end:"))
print(f"The average of the {count} marks is {total/count}")

```

Output in terminal

```
Please input a mark or a negative number to end: 78
Please input next mark or a negative number to end: 89
Please input next mark or a negative number to end: 98
Please input next mark or a negative number to end: 88
Please input next mark or a negative number to end: 85
Please input next mark or a negative number to end: -1
The average of the 5 marks is 87.6
```

Similarly, we can also write a program that takes a list of student marks and identify the lowest marks, the highest marks, and the mean, as shown in Table 3-6.

Table 3-6: Case study: How to use a *while* loop statement

The problem	In this problem, you will get a list of student marks and find out the lowest, the highest, and the mean.
The analysis and design	The steps involved are Step 1: Start with an empty list Step 2: Get marks in a loop and build a list Step 3: Sort the list Step 4: Get the lowest mark, the mean, and the highest mark from the sorted list

(continued on next page)

https://doi.org/10.15215/remix/9781998944088.01

The code	"""
	Get a number of marks into a list, sort the list and find the mean, max, and min.
	"""
	
	mark = int(input("Please input a mark or -1 to complete:"))
	marks = []
	while mark >= 0:
	  marks.append(mark)
	  mark = int(input("Please input next mark or -1 to complete:"))
	marks = sorted(marks)
	ln = len(marks)
	print(f"The lowest is {marks[0]}, mean is {marks[ln // 2]}, highest is {marks[ln - 1]}")
	print(marks)

The result	The lowest is 25, mean is 65, highest is 85 [25, 32, 36, 54, 55, 58, 65, 66, 74, 77, 85, 85]

The next programming task for using a ***while*** loop is to write a program that can take an integer from a user and decide whether the integer is a prime number or not (see Table 3-7).

Table 3-7: Code sample: Prime number test

	Code sample in VS Code IDE
The task	Take an integer from a user and decide whether the integer is a prime number or not.
Analysis and design	A prime number is an integer greater than 1 that cannot be divided by any of the integers between 1 and itself. The way to test this can be very simple: try to divide that number by all the numbers between 1 and that number. If it can be divided, and the answer is a whole number, then that number is prime. However, since i*j = j*i, we only need to test integers <= square root of m to speed up the test. We can assume the integer is a prime at the start of the test with a flag initialized to True; if a number in the range discussed above is found to be able to divide into the integer, the flag is changed to False and the test is complete. If no such number is found until the end of the range, the flag will remain True, and the number is a prime. The algorithm is as follows: Step 1: Get an integer from the user into variable n Step 2: Initialize flag to True Step 3: Initialize variable i to 2 Step 4: If i <= sqrt(n), do the following:

https://doi.org/10.15215/remix/9781998944088.01

Table 3-7: Code sample: Prime number test *(continued)*

	Code sample in VS Code IDE
	Step 4.1: If n % m != 0, then m = m + 1; go to Step 4, else Step 4.2: If flag = False, break out and stop testing Step 5: If flag = True, then number n is a prime; print it out Step 6: End program

```
"""
Program file name: primetest.py
Application: this program takes an integer from
  user input and determines
whether it is a prime or not.
"""

m = int(input("Give me an integer that is greater
  than 1, and I will tell you if it is a prime: "))
flag = True
if m < 2:
  print("The number must be greater than 1")
  flag = False
else:
  i = 2
  while i< = m ** (1 / 2):   # because i * j = j *
  i, we only need to check integers < = sqrt(m)
    if m % i == 0:
      print(f"{m} is divisible by {i}, so that ",
  end=" ")
      flag = False
      break
    else:
      i += 1
if flag:
  print(f"{m} is a prime")
else:
  print(f"{m} is not a prime")
```

Output in terminal	Give me an integer that is greater than 1, and I will tell you if it is a prime: 911 911 is a prime

Please note the ***break*** statement in the example above. It is used to get out of the loop immediately by ignoring all the code before it without going back to test the looping condition. This is a way to get out of a loop in the middle of an iteration and is applicable to both the ***while*** loop and ***for*** loop.

Somewhat related to the ***break*** statement, the ***continue*** statement is used within a loop to go back directly to the beginning of the iteration—testing the

https://doi.org/10.15215/remix/9781998944088.01

looping condition in a ***while*** loop or taking the next item of the sequence in a ***for*** loop.

Common Coding Mistakes with a *while* Loop

As mentioned, the ***while*** loop is good for iterations in which the number of iterations is unknown but the condition of looping is known. To ensure that the loop will end as expected, the looping condition must be changed within the code block of the ***while*** statement. Otherwise, the loop will go on forever. To ensure that, there must be at least one variable within the code block of the loop whose value needs to be changed during iterations. Such a variable is called an iteration variable.

There are two common mistakes when using the ***while*** statement. The first one is not correctly writing the looping condition, which could be coded so that it is always true (hence, the iteration will keep going forever) or incorrectly coded to become false at the wrong time. For example, if x < 0 is written in place of x > 0 as the looping condition, the iteration would not finish as desired or wouldn't run at all.

The other common mistake people often make when using the ***while*** statement is not coding the code block correctly to ensure that the following conditions hold:

1. There will be at least one iteration variable within the code block of the ***while*** loop.
2. The value(s) of iteration variable(s) must change within the code block.
3. The logical expression of the looping condition is not correctly written. This mistake may occur when inequal operators are involved in the logical expression of the looping condition. For example, using > in place of >=, or using < in place <=, will cause the program to miss one iteration of the loop.

In the example we just mentioned above, if x is never changed within the code block of the while loop, the value of the looping condition will remain the same, and the iteration will keep going forever as well.

3.6 Iterate with *for* Versus *while*

The ***for*** loop is controlled by a variable going through a sequence with a finite number of items. So essentially, the ***for*** loop is good for cases when the number of iterations is known. Let's take a second look at the example about finding

https://doi.org/10.15215/remix/9781998944088.01

the average mark for a class. Assume now that we know the class has 30 students and we want the program to take the final marks of all students in the class and calculate the average mark. Because the number of students in the class is known, the ***for*** loop can be used to iterate, as shown below:

```
In [ ]:  total = 0
         for i in range(30):
           mark = int(input("Please input a mark:"))
           total += mark
         print(f'The average course mark is {total/30}.')
```

When the number of iterations is unknown, the ***while*** loop must be used, in which case the condition for exiting from the loop must be known. In this particular application, because no mark will be a negative number, we can use negative numbers to signify the end of input, to end the iteration, as shown in the following example:

```
In [ ]:  total, mark, count = 0, 0, 0
         while mark >= 0:
           mark = int(input("Please input a mark:"))
           if (mark >= 0):
             total += mark
             count += 1
         print(f'The average course mark of {count} students is
          {total/count}.')
```

```
Out [ ]: Please input a mark: 89
         Please input a mark: 96
         Please input a mark: 78
         Please input a mark: 97
         Please input a mark: 88
         Please input a mark: -7
         The average course mark of 5 students is 89.6.
```

In the example above, we use *mark* as an iteration variable to control the loop. Initializing it with 0 = ensures that the looping condition (logical expression mark >= 0) is satisfied to start the iteration.

Since we know the logical expression (mark >= 0) is true when 0 is assigned to mark, we can also simply use constant True in place of mark >= 0 and then use an ***if*** statement with the ***break*** statement to get out of the loop when a certain condition (condition to exit the loop) is met. The revised version of the program is shown below:

https://doi.org/10.15215/remix/9781998944088.01

```
In [ ]:  total, mark, count = 0, 0, 0
         while True:
           mark = int(input("Please input a mark:"))
           if (mark >= 0):
             total += mark
             count += 1
           else:
             break

         print(f'The average course mark of {count} students is
          {total/count}.')
```

```
Out [ ]: Please input a mark: 88
         Please input a mark: 98
         Please input a mark: 97
         Please input a mark: 96
         Please input a mark: 78
         Please input a mark: -3
         The average course mark of 5 students is 91.4.
```

In the previous section we mentioned that the ***while*** loop is an entry-controlled iteration, which means that the code block of the loop statement may not be executed at all if the entry condition is not met at the beginning. In the example above, when we use True in place of the looping condition, it has guaranteed that the code block will always be run at least once, and the iteration could go on forever if a ***break*** statement is not used and executed when a certain condition (condition to exit) is met. This has made ***while*** statement an exit-controlled iteration. The flowchart of such iteration is shown in Figure 3-7.

Compared to the ***for*** statement, the ***while*** statement is more powerful and its uses are more versatile. In fact, all code written with a ***for*** statement can be rewritten with a ***while*** statement, though when looping through sequences, coding with ***for*** statements is more elegant and more readable.

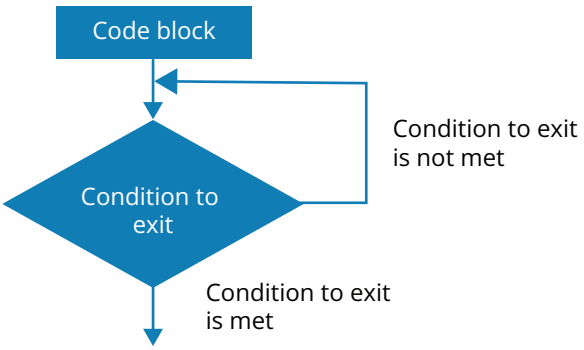


Figure 3-7: Flowchart for a ***while*** loop

https://doi.org/10.15215/remix/9781998944088.01

Since we know how to put a code block in a ***while*** loop, we can improve the grade conversion program written in the previous chapter so that we do not have to rerun the program for each conversion (see Tables 3-8 and 3-9).

Table 3-8: Conversion between numeric grade and letter grade in Alberta

Letter grade	Percentage
A+	90–100%
A	95–89%
A–	80–84%
B+	77–79%
B	73–76%
B–	70–72%
C+	67–69%
C	63–66%
C–	60–62%
D	55–59%
D	50–54%
F	0–49%

Table 3-9: Case study: How to make a grade converter

The problem	In this case study, you are required to design a program using an ***if-elif-else*** statement to convert as many numeric grades to alpha/letter grades as needed until the user inputs -1 to stop.
The analysis and design	In Canada, different provinces may use different conversion tables for numeric grade to letter grade conversion. In this case study, we take the one used in Alberta, as shown above. Based on the conversion table, our program needs to make 12 selections; for each selection, a letter grade will be printed out when the numeric grade falls within the corresponding interval. Since we allow a user to convert as many numeric grades as needed until the user explicitly tells the program to stop by inputting -1, we will put the above ***if-elif-elif*** multiple selection statements inside a ***while*** loop. Note that for this problem, the ***for*** loop will not work because we do not know how many times the loop will need to run.

(continued on next page)
https://doi.org/10.15215/remix/9781998944088.01

Table 3-9: Case study: How to make a grade converter *(continued)*

The code

```
"""
Design a program using an if-elif-else statement
to convert numeric grades to alpha/ letter grades.
"""
number_grade = 0
while number_grade >= 0:
  number_grade = input("Please input a numeric grade
  between 0 and 100 to convert; input -1 to exit:")
  number_grade = round(float(number_grade))
  if number_grade >= 90:
    print(f"alpha/letter grade for {number_grade}
  is A+")
  elif number_grade >= 85:
    print(f"alpha/letter grade for {number_grade}
  is A")
  elif number_grade >= 80:
    print(f"alpha/letter grade for {number_grade}
  is A-")
  elif number_grade >= 76:
    print(f"alpha/letter grade for {number_grade}
  is B+")
  elif number_grade >= 73:
    print(f"alpha/letter grade for {number_grade}
  is B")
  elif number_grade >= 70:
    print(f"alpha/letter grade for {number_grade}
  is B-")
  elif number_grade >= 67:
    print(f"alpha/letter grade for {number_grade}
  is C+")
  elif number_grade >= 64:
    print(f"alpha/letter grade for {number_grade}
  is C")
  elif number_grade >= 60:
    print(f"alpha/letter grade for {number_grade}
  is C-")
  elif number_grade >= 55:
    print(f"alpha/letter grade for {number_grade}
  is D+")
  elif number_grade >= 50:
    print(f"alpha/letter grade for {number_grade}
  is D")
  elif number_grade >= 0:
    print(f"alpha/letter grade for {number_grade}
  is F")
  else:
    print("Thank you for using this grade
  converter!")
```

The result

```
Please input a numeric grade between 0 and 100 to convert; input -1
  to exit:96
alpha/letter grade for 96% is A+
Please input a numeric grade between 0 and 100 to convert; input -1
  to exit:79
alpha/letter grade for 79% is B+
Please input a numeric grade between 0 and 100 to convert; input -1
  to exit:67
alpha/letter grade for 67% is C+
Please input a numeric grade between 0 and 100 to convert; input -1
  to exit:-1
Thank you for using this grade converter!
```

https://doi.org/10.15215/remix/9781998944088.01

Chapter Summary

- Knowing what to do at a given time and under certain conditions is important for any intelligent being.
- Conditional statements are necessary and important constructs in all programming languages.
- ***if***, ***if-else***, ***if-elif***, ***if-elif-else*** are the constructs for conditional statements.
- Each ***if***, ***if-else***, ***if-elif***, ***if-elif-else*** statement represents a flow of program execution.
- The ***if*** statement is for one selection. It will execute a code block if the condition is met.
- The ***if-else*** statement is good for two selections.
- The ***if-elif*** and ***if-elif-else*** statements are good for multiple selections.
- The conditions behind ***if*** and ***elif*** are logical or Boolean expressions.
- Python has two constructs for iteration, the ***for*** statement and the ***while*** statement.
- The ***for*** statement can be used to repeat a code block through each member of an iterable, such as a list, tuple, or string, or an iterable object such as range(...).
- When the iteration includes a ***for*** statement, the number of iterations can be determined most of the time, unless the ***break*** statement is used within the code block to break out from the loop.
- The ***while*** statement is used to iterate under certain conditions.
- The number of repetitions needed when using a ***while*** statement is often unknown. One can be used with or without a ***break*** statement within the code block.
- The ***continue*** statement can be used within the code block of a ***for*** or ***while*** statement to directly go to the next iteration.
- Any ***for*** statement can be rewritten as a ***while*** statement.

Exercises

1. Mentally run the following code blocks and write down the output of each code block.

   a.
   ```
   m, n = 10, 20
   if m * n < 1000:
      print('This is not enough!')
   ```

https://doi.org/10.15215/remix/9781998944088.01

b.
```
m, n = 10, 3
if m // n == m / n:
   print(f'{m} // {n} = {m//n}')
else:
   print(f'{m} / {n} = {m/n}')
```

c.
```
m, n = 13, 5
if m * 2 > n**2:
   print(f'{m} * {2} = {m*2}')
else:
   print(f'{n} ** {2} = {n**2}')
```

2. Mentally run each of the code blocks below and write down the output of each code block:

a.
```
for i in range(1, 6):
 print(f'The cube of {i} is {i*i*i}')
```

b.
```
i, s = 1, 1
while i<=10:
 s *= i
 i += 1
print("the product of 1x2x3…10 is ", s)
```

c.
```
total = 0
for i in range(10):
 total += i*2 +1
print(f'1+3+5+ … +19 = {total}')
```

d.
```
number = 32
factors = []
for d in range(1, number):
 if number % d == 0:
 factors += [d]
print(f'factors of {number} are {factors}')
```

Projects

1. Write a program that gets three numbers from the user and displays the biggest number among the three.
2. Write a program that gets a number from the user then says whether the number is an even number or odd number.
3. Write a program that takes three numbers from the user as the lengths of three lines, then determines if the three lines can make a triangle.
4. Write a program that takes three numbers from the user as the lengths of three lines, then determines if the three lines can make a triangle. If the three lines can make a triangle, the program should further determine if the triangle is an equilateral triangle or an isosceles triangle.

https://doi.org/10.15215/remix/9781998944088.01

5. Write a program that takes three numbers from the user as the lengths of three lines, then determines if the three lines can make a triangle. If the three lines can make a triangle, the program should further determine if the triangle will be a right triangle.
6. Compound interest is a common practice in finance and banking, allowing you to earn interest on interest as well as on the principal. Assume that your bank offers a savings account with which you can earn compound interest. The amount you deposit into the account is p, and annual compound interest is r. By the end of n years after your initial deposit, the account balance will be $a = p(1 + r)^n$. For this project, write a program that takes three numbers from the user as the initial deposit, the annual interest, and the number of years that the user wants the money to stay in the account. Calculate and display how much money the user will have by the end of the nth year.
7. In some countries like Canada, tax on taxable personal income for a year is calculated progressively according to a calculation table set for the year, such as the one shown below:

Income tax	15% on the first $48,534 or less	20.5% on the next $48,534	26% on the next $53,404	29% on the next $63,895	33% on taxable income over $214,368
Income	$0–$48,535	$48,536–$97,069	$96,070–$150,473	$150,474–$214,368	over $214,368

   Write a program that takes taxable income from a user and calculates the total income tax payable according to the table above.
8. A mortgage is the money borrowed from a lender for the purchase of a property. Mortgages and mortgage payments are a big thing for almost everyone. For a mortgage with principal of P at a fixed monthly interest rate of r that needs to be paid off in Y years or 12 * Y = N months, the monthly payment would be:

$$M = \frac{rP(1+r)^N}{(1+r)^N - 1}, \text{ if r is not equal to 0; otherwise, } M = \frac{P}{N}$$

   Write a program that takes the principal, fixed annual interest rate, and years of amortization then calculates and displays the monthly payment amount. *Hint:* You will need to work out the number of months from number of years, and the monthly interest rate from the annual interest rate.

https://doi.org/10.15215/remix/9781998944088.01

9. In the world of science and mathematics, the existence of some constants has attracted the attention of many scientists and mathematicians. Among those constants, pi π is the most well-known. Many great efforts have been made to get a more precise value for π. The following is a formula developed by Gottfried Leibniz for the calculation of π.

$$\pi/4 = 1 - 1/3 + 1/5 - 1/7 + 1/9 - \ldots = \sum_{i=0}^{n} \frac{(-1)^i}{2i+1}$$

   Write a program that takes an integer from the user to specify the number of terms used to calculate π. Calculate and display the approximate value.
10. Three integers—A, B and C—are called a Pythagorean when C * C = A * A + B * B. Write a program to take an input of integer from the user and find all the Pythagorean triples of integers, none of which are bigger than the integer taken from the user. For example, if the number from the user is 6, then 5, 4, and 3 are a Pythagorean triple because $5^2 = 4^2 + 3^2 = 25$.
11. Compound interest is a common practice in finance and banking to earn interest on interest as well as on the principal. Assume that your bank offers you a deposit account through which you can earn compound interest, the amount you deposit into the account is p, and annual compound interest is r. By the end of n years after your initial deposit, the account balance will be $a = p(1 + r)^n$. For this project, get p, r, and n from the user, then calculate and display a table showing the balance, the interest earned each year, and the total interest earned so far.
12. An integer is called a perfect number* if the sum of its factors, excluding itself, is equal to the integer itself. For example, 6 is a perfect number because 6 = 1 + 2 + 3, and 1, 2, and 3 are all its factors. Write a program to get a number from a user, then determine if the number is a perfect number. If yes, display all its factors.
13. For a given integer n, if another integer m can divide n, then m is called a factor of n. In mathematics, finding all factors of a given integer is an important operation, especially for cryptography. Write a program that takes an integer from the user and determine and display all the factors of the number given by the user.

* By definition, a perfect number is a positive integer that is equal to the sum of all its divisors, excluding itself but including 1. The smallest perfect number is 6, which is equal to the sum of 1, 2, and 3.

https://doi.org/10.15215/remix/9781998944088.01

14. Read a series of float numbers from the user and calculate and display the average of all the numbers given by the user. Assume that the number of float numbers is unknown but that a negative number is used to indicate the end of the input.
15. For a mortgage with a principal of P at a fixed monthly interest rate of r that needs to be paid off in Y years or 12 * Y = N months, the monthly payment would be:

    $$M = \frac{rP(1+r)^N}{(1+r)^N - 1}, \text{ if r is not equal to 0; otherwise, } M = \frac{P}{N}$$

    Continuing the project you did above for project 8, calculate a table showing the interest and principal paid each month and the principal balance at each month's end.

https://doi.org/10.15215/remix/9781998944088.01

This page intentionally left blank

Chapter 4

Handle Errors and Exceptions in Programs

Errors in programs are inevitable but must be handled elegantly when they occur. In this chapter, you will learn how to raise and handle exceptions when errors occur in programs.

Learning Objectives

After completing this chapter, you should be able to

- explain different types of errors and exceptions that may occur in a Python program.
- write down the specific codes for some common types of exceptions.
- use the ***try*** statement properly to handle possible exceptions raised by potential errors in a block of program code.
- understand the cause of the following messages:
  - TypeError
  - NameError
  - RuntimeError
  - OSError
  - ValueError
  - ZeroDivisionError
  - AssertionError
  - FileNotFoundError
- understand how to purposely throw an exception using a ***raise*** statement in a program.
- use an ***assert*** statement to prevent future exceptions in a program.
- list user-defined exceptions.
- define a class of exceptions.

https://doi.org/10.15215/remix/9781998944088.01

4.1 Errors in Your Programs

It is not unusual to have errors in your programs, especially for beginners. There are three types of errors in computer programs: syntax errors, runtime errors, and logic errors. If you program in a modern IDE such as VS Code, syntax errors can be easily avoided because whenever there is incorrect syntax, such as a misspelled keyword, you will be alerted by the IDE. Modern IDEs can even detect whether an identifier is used properly, such as when an undefined function is called or the value of a variable is used but no value has been previously assigned to the variable.

On the other hand, runtime errors only happen during the runtime of programs. Runtime errors are the most irritating to users and should be caught and handled gracefully in programs.

Logic errors are those caused by incorrect logic or operation in a program for a given problem or task. The following are some examples of logic errors:

1. An incorrect operator is used in an expression or statement, such as use + instead of −, > instead <, or vice versa.
2. The boundary of a sequence is miscounted.

Compared to syntax errors and runtime errors, the consequences of logic errors can be more costly because they often cause the program to produce incorrect results. It is even more concerning because logic errors often remain undetected until someone realizes that the result from the program is not what was expected. For example, syntax and runtime errors will produce no result, which will be immediately noticed by the user, whereas an incorrect result often goes unnoticed till it causes unexpected consequences, such as a missile being sent to the wrong target.

No programming techniques can help eliminate logic errors. It is up to the programmers to make the logic correct. That is why computer programmers should also take certain math courses.

Because Python programs are normally interpreted without compilation, syntax errors such as misspelled keywords are often found at runtime as well (unless the programs were developed in a smart IDE that can help identify the syntax errors while programming). However, the exception-handling mechanism provided by Python or other programming is not intended to catch syntax errors. You, as a programmer, must ensure that the programs you write use correct syntax, with the help of an IDE whenever is available.

You may be wondering what errors can be handled by the exception-handling mechanism provided by Python. The following is a long list of exception classes, all of which can be caused by errors in programs. The ***try-except*** statement provided by Python to handle exceptions and errors uses the names of exception classes to

https://doi.org/10.15215/remix/9781998944088.01

identify particular exceptions caused by specific errors. Remember at least the names of these commonly occurring exceptions or know where to find them.

Exception

This is the superclass of all exception classes to be detailed below. In the code sample below, any error will be caught and treated the same because it has "Exception" in the list of exceptions behind the ***except*** clause.

```
In [ ]:  try:
           n = int(input('Give me an integer:'))
           m = int(input('Give me another integer:'))
           n /= m   # divide n by m
         except Exception:
           print('Wrong: It is not an integer or m is 0')
Out [ ]: Give me an integer: 12
         Give me another integer: 0
         Wrong: It is not an integer or m is 0
```

In this particular case, m was given 0 so that the exception was caused by dividing n by 0, but the ***except*** clause could not tell because "Exception" is not a specific exception class.

When programming, you should put a code block in a ***try-except*** statement if you know that an error or errors might occur and you may even know what kind of error it may be. You want to have the errors handled elegantly. In the sample code above, because we ask for input from users and there is no guarantee that the user will not input a 0 for m, which will be used as the denominator or divisor, we put the code in a ***try-except*** to handle the possible exception raised by the error. Otherwise, the program would stop with a report of the error, as shown below:

```
In [ ]:  n = int(input('Give me an integer:'))
         m = int(input('Give me another integer:'))
         n /= m# divide n by m
Out [ ]: Give me an integer: 12
         Give me another integer: 0
         ---------------------------------
         ZeroDivisionError  Traceback (most recent call last)
         <ipython-input-2-d0d8abede315> in <module>
         1 n = int(input('give me an integer:'))
         2 m = int(input('give me another integer:'))
         -> 3 n /= m   # divide n by m

         ZeroDivisionError: division by zero
```

https://doi.org/10.15215/remix/9781998944088.01

The use of the ***try-except*** statement will be explained in the next section. For now, just remember the code lines between ***try*** and ***except*** are part of the program to be executed for the application, while the code under the ***except*** header tells what to do when a given exception has occurred.

ArithmeticError

The base class of all arithmetic errors, including OverflowError, ZeroDivisionError, and FloatingPointError, which means ***except ArithmeticError*** would be the same as ***except (OverflowError, ZeroDivisionError, FloatingPointError)***. Note that when there are multiple error names behind the ***except*** keyword, a pair of parentheses is used to enclose them.

OverflowError

This subclass of ArithmeticError is raised when the result of an arithmetic operation is too large to be represented.

A coding example of catching arithmetic errors like this is shown below:

```
In [ ]:  try:
           x = pow(123567,999999)
           print(f'Big x has {len(str(x))} digits')
         except OverflowError:
           print('It has overflowed because the number is too
          big')
```

```
Out [ ]: Big x has 5091898 digits
```

The code in the ***try*** clause calculates the power of 123567 to 999999 or 123567**999999. The code is put in a ***try-except*** statement because the result is expected to be very big and may overflow. Though the result has over five million digits, no exception is raised because the design and implementation of Python can handle very big numbers.

ZeroDivisionError

ZeroDivisionError is raised when the divisor of a division or module operation is 0. With this very specific exception/error name, the earlier example of this section can be rewritten as shown below:

```
In [ ]:  try:
           n = int(input('Give me an integer:'))
           m = int(input('Give me another integer:'))
           n /= m   # divide n by m
         except ZeroDivisionError:
           print('Wrong: 0 cannot be used as a divisor!')
```

https://doi.org/10.15215/remix/9781998944088.01

```
Out [ ]:  Give me an integer: 23
          Give me another integer: 0
          Wrong: 0 cannot be used as a divisor!
```

FloatingPointError

FloatingPointError is raised when a floating-point operation fails. However, Python does not raise such errors by default in its standard distribution. You will need a Python built with the -- with-fpectl flag, and import a module called fpectl when you want to turn the floating-point error control on or off.

AssertionError

AssertionError is raised when the ***assert*** statement fails. The ***assert*** statement is used to make an assertion on an assumed fact, such as whether a variable is defined, whether a variable is holding a specific value, or whether a value is a member of a sequence or set. If the assumed fact is not True, an AssertionError will be raised so that we know the assumed fact is untrue and we may need to deal with it, such as doing something else in the absence of the assumed fact.

```
In [ ]:  vs = list(range(19))   # create a list with 19 members
           indexed from 0 to 18
         assert(20 in vs)   # 20 is not in the list
```

```
Out [ ]: ---------------------------------
         AssertionError  Traceback (most recent call last)
         <ipython-input-14-5a912881af6c> in <module>
         1 vs = list(range(19))   # create a list with 19 members indexed from 0 to 18
         ----> 2 assert(20 in vs)   # 19 is out of the range > 18

         AssertionError:
```

AttributeError

AttributeError is raised when an attribute assignment or reference fails. Such an error will occur if you use an attribute of an object but the attribute itself does not exist.

```
In [ ]:  class Student:
           pass
         s0 = Student()
         s0.firstname = 'John'
         s0.lastname = 'Doe'
         print(s0.fullname)
```

https://doi.org/10.15215/remix/9781998944088.01

```
Out [ ]: ---------------------------------
         AttributeError  Traceback (most recent call last)
         <ipython-input-15-5dc5b1212e9f> in <module>
         6 s0.lastname = 'Doe'
         7
         ----> 8 print(s0.fullname)

         AttributeError: 'Student' object has no attribute 'fullname'
```

BufferError

BufferError is raised when a buffer-related operation cannot be performed. This often happens when working directly with computer memory and making restricted changes to a given memory area (buffer). The following is an example:

```
In [ ]: import io
        data = b'Hello, Python!'   # this creates a bytearray
        darray = io.BytesIO(data)   # this creates a read-write
          copy of the bytearray
        dbuff = darray.getbuffer()   # the memory of the
          bytearray is exported
        darray.write(b'Hello World!')   # raise error because
          the buffer is not changeable

Out [ ]: ---------------------------------
         BufferError  Traceback (most recent call last)
         <ipython-input-23-dc2c4a5f6bbd> in <module>
         3 darray = io.BytesIO(data)   # this creates a read-write copy of the
           bytearray.
         4 dbuff = darray.getbuffer()   # the memory of the bytearray is exported
         --> 5 darray.write(b'Hello World!')   # raise error because the buffer is not
           changeable

         BufferError: Existing exports of data: object cannot be re-sized
```

EOFError

EOFError is raised when the input() function hits the end-of-file condition.

GeneratorExit

GeneratorExit is raised when a generator's close() method is called.

ImportError

ImportError is raised when the imported module is not found.

https://doi.org/10.15215/remix/9781998944088.01

IndexError

IndexError is raised when the index of a sequence is out of range.

```
In [ ]:  vs = list(range(19))   # create a list with 19 members
           indexed from 0 to 18
         vs[19] *= 3   # 19 is out of the range > 18
```

```
Out [ ]: ---------------------------------
         IndexError  Traceback (most recent call last)
         <ipython-input-3-47e31ad8b75b> in <module>
         1 vs = list(range(19))
         ----> 2 vs[19] *= 3

         IndexError: list index out of range
```

KeyError

KeyError is raised when a key is not found in a dictionary.

```
In [ ]:  vdict = {1:'One', 2:'Two', 3:'Three'}   # create a
           dictionary
         vdict[5]   # the dictionary doesn't have key 5
```

```
Out [ ]: ---------------------------------
         KeyError  Traceback (most recent call last)
         <ipython-input-4-3011ee6a346e> in <module>
         1 vdict = {1:'One', 2:'Two', 3:'Three'}
         ----> 2 vdict[5]
         KeyError: 5
```

KeyboardInterrupt

KeyboardInterrupt is raised when the user hits the interrupt key (Ctrl+C or Delete).

MemoryError

MemoryError is raised when an operation runs out of memory.

ModuleNotFoundError

ModuleNotFoundError is raised by ***import*** when a module could not be located, or None is found in sys.modules.

```
In [ ]:  import fpectl   # import module fpectl for floating-
           points control
         round(14.5/0, 3)
```

https://doi.org/10.15215/remix/9781998944088.01

```
Out [ ]: ---------------------------------
         ModuleNotFoundError  Traceback (most recent call last)
         <ipython-input-8-3808b892163e> in <module>
         ----> 1 import fpectl
         2 round(14.5/0, 3)

         ModuleNotFoundError: No module named 'fpectl'
```

NameError

NameError is raised when a variable is not found in the local or global scope.

```
In [ ]: print(what)    # print the value of variable named what,
          undefined

Out [ ]: ---------------------------------
         NameError  Traceback (most recent call last)
         <ipython-input-1-8b57ddde6300> in <module>
         ----> 1 print(what)

         NameError: name 'what' is not defined
```

NotImplementedError

NotImplementedError is raised by abstract methods such as when an abstract method is called.

OSError

OSError is raised when a system operation causes a system-related error.

BlockingIOError

BlockingIOError is a subclass of OSError, raised when an operation would block on an object (e.g., a socket) set for a nonblocking operation.

ChildProcessError

ChildProcessError is a subclass of OSError, raised when an operation on a child process fails.

ConnectionError

ConnectionError is a subclass of OSError and a base class for connection-related issues.

https://doi.org/10.15215/remix/9781998944088.01

BrokenPipeError

BrokenPipeError is a subclass of ConnectionError, raised when trying to write on a pipe while the other end has been closed or when trying to write on a socket that has been shut down for writing.

ConnectionAbortedError

ConnectionAbortedError is a subclass of ConnectionError, raised when a connection attempt is aborted by the peer.

ConnectionRefusedError

ConnectionRefusedError is a subclass of ConnectionError, raised when a connection attempt is refused by the peer.

ConnectionResetError

ConnectionResetError is a subclass of ConnectionError, raised when a connection is reset by the peer.

FileExistsError

FileExistsError is a subclass of OSError, raised when trying to create a file or directory that already exists.

FileNotFoundError

FileNotFoundError is a subclass of OSError, raised when a file or directory is requested but does not exist.

IsADirectoryError

IsADirectoryError is a subclass of OSError, raised when a file operation is requested on a directory.

NotADirectoryError

NotADirectoryError is a subclass of OSError, raised when a directory operation is requested on something that is not a directory.

PermissionError

PermissionError is a subclass of OSError, raised when trying to run an operation without adequate access rights such as filesystem permissions.

https://doi.org/10.15215/remix/9781998944088.01

ProcessLookupError

ProcessLookupError is a subclass of OSError, raised when a given process doesn't exist.

TimeoutError

TimeoutError is a subclass of OSError, raised when a system function has timed out at the system level.

RecursionError

RecursionError is a subclass of Exception, raised when the maximum recursion depth set by the system is exceeded. The set recursion depth can be found by calling sys.getrecursionlimit().

ReferenceError

ReferenceError is a subclass of Exception, raised when a weak reference proxy is used to access a garbage collection referent.

RuntimeError

RuntimeError is raised when an error does not fall under any other category.

StopIteration

StopIteration is raised by the next() function to indicate that there is no further item to be returned by the iterator.

StopAsyncIteration

StopAsyncIteration is raised by the __anext__() method of an asynchronous iterator object to stop the iteration.

SyntaxError

SyntaxError is raised by the parser when a syntax error is encountered.

```
In [ ]:  s = 0
         for i in range(10)   # a colon is missing which will
           cause a syntax error
            s += i
```

```
Out [ ]: File '<ipython-input-16-6e54ba8cdb35>', line 2
         for i in range(10)
                          ^
         SyntaxError: invalid syntax
```

https://doi.org/10.15215/remix/9781998944088.01

IndentationError

IndentationError is raised when there is an incorrect indentation. Such errors may occur quite often at the beginning of your study of Python programming. You must pay great attention to it because indentation matters a lot in Python programs/scripts.

In [ ]:
```
s = 0
while i < 10:
  s += i
  i += 1# not indented the same
```

Out [ ]:
```
File '<tokenize>', line 4
  i += 1   # not indented the same
  ^
IndentationError: unindent does not match any outer indentation level
```

TabError

TabError is raised when the indentation consists of inconsistent tabs and spaces. Indentations can be made of spaces and tabs, but they need to be consistent to avoid such errors.

SystemError

SystemError is raised when the interpreter detects an internal error.

SystemExit

SystemExit is raised by the sys.exit() function.

TypeError

TypeError is raised when a function or operation is applied to an object of an incorrect type.

In [ ]:
```
sm = 10 + 'twenty'   # computer doesn't know twenty is
  20
```

Out [ ]:
```
---------------------------
TypeError  Traceback (most recent call last)
<ipython-input-19-a90f29de94a2> in <module>
----> 1 sm = 10 + 'twenty'
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

https://doi.org/10.15215/remix/9781998944088.01

UnboundLocalError

UnboundLocalError is 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

UnicodeError is raised when a Unicode-related encoding or decoding error occurs.

UnicodeEncodeError

UnicodeEncodeError is raised when a Unicode-related error occurs during encoding.

UnicodeDecodeError

UnicodeDecodeError is raised when a Unicode-related error occurs during decoding.

UnicodeTranslateError

UnicodeTranslateError is raised when a Unicode-related error occurs during translation.

ValueError

ValueError is raised when a function gets the correct type of argument but an improper value.

```
In [ ]:  i = int('ten')    # int doesn't convert ten into 10

Out [ ]: ---------------------------------
         ValueError  Traceback (most recent call last)
         <ipython-input-20-6bbb9f319a0e> in <module>
         ----> 1 i = int('ten')

         ValueError: invalid literal for int() with base-10: 'ten'
```

The following is a sample program to show how errors should be handled in a Python program:

```
# a python program to show how errors and exceptions are
handled

# ask the user to enter two numbers
num1 = input("Enter the first integer number: ")
num2 = input("Enter the second integer number: ")
```

https://doi.org/10.15215/remix/9781998944088.01

```
# try to convert the inputs to floats and divide them
try:
  result = int(num1) / int(num2)
  print(f"The result of dividing {num1} by {num2} is
{result}.")
# handle the possible errors and exceptions
except ValueError:
    print("Invalid input. Please enter numbers only.")
except ZeroDivisionError:
    print("Cannot divide by zero. Please enter a nonzero
number.")
except Exception as e:
    print(f"An unexpected error occurred: {e}")
```

The following exception will be raised when you run the code but input a literal instead of an integer:

```
Invalid input. Please enter numbers only.
```

If you type a 0 for the second input, the following exception will be raised:

```
Cannot divide by zero. Please enter a nonzero number.
```

Note that in real applications, you may not want a user to restart the program when an error has occurred. Instead, you may want to let the program continue until the user has given a valid input.

```
# a python program to show how errors and exceptions are
handled

# initialize a flag to indicate if the division is
successful
success = False

# use a while loop to keep asking for inputs until
success is True
while not success:
    # ask the user to enter two numbers
    num1 = input("Enter the first number: ")
    num2 = input("Enter the second number: ")
```

https://doi.org/10.15215/remix/9781998944088.01

```
    # try to convert the inputs to floats and divide them
    try:
        result = int(num1) / int(num2)
        print(f"The result of dividing {num1} by {num2}
is {result}.")
        # set success to True if no error occurs
        success = True
    # handle the possible errors and exceptions
    Zexcept ValueError:
        print("Invalid input. Please enter numbers
only.")
    except ZeroDivisionError:
        print("Cannot divide by zero. Please enter a
nonzero number.")
    except Exception as e:
        print(f"An unexpected error occurred: {e}")
```

This will ensure the program will continue until two integer numbers are received and the division is successful.

Now let us solve a real problem: write a Python program to find all perfect numbers in a given range set by a user. A perfect number is an integer number that is equal to the sum of all factors, including 1 but excluding itself. For example, 6 is a perfect number because 6 = 1 + 2 + 3, and 1, 2, and 3 are all its factors.

So basically, the steps to take, or the algorithm, will be as follows:

1. Get two integers from user to set the range, assigned to a and b, respectively. If a is greater than b, then we swap their values.
2. Loop through all the numbers between a and b, including a and b, and test each number to see if it is a perfect number; if yes, we add it to a list holding perfect numbers found so far. The list should be set as empty at the beginning.
3. Print out all the perfect numbers in the list, and stop.

To test a number to see if it is perfect, according to the definition above, we need to first find out its factors and check whether the sum of all the factors is equal to the number itself. So there will be two additional steps:

2.1. Find out all the factors, and keep them all in a list.
2.2. Check whether the number is equal to the sum of all its factors in the list built up in step 2.1.

https://doi.org/10.15215/remix/9781998944088.01

The following is one implementation of the algorithm in Python, taken from a Jupyter Notebook cell:

```
# a python program to find all the perfect numbers in a
range
# set by the user

# first get two integers from user
# initialize a flag to indicate if the division is
successful
success = False

# use a while loop to keep asking for inputs until
success is True
while not success:
    # ask the user to enter two numbers
    num1 = input("Enter the first number: ")
    num2 = input("Enter the second number: ")

    # try to convert the inputs to floats and divide them
    try:

        a, b = int(num1), int(num2)

        # set success to True if no error occurs
        success = True

        if a>b: # then we need to swap a and b
            c = a
            a = b
            b = c
        perfect_list = [] # make an empty list ready to
hold all perfect numbers
        for n in range(a, b+1): # we said b is included
            # start finding all factors of n
            factor_list = [1] # make a list with 1 as a
single element
            for f in range(2,n): # start from 2, with n
as excluded from factors
                if n%f == 0: # f is a factor of n
```

https://doi.org/10.15215/remix/9781998944088.01

```
                if n%f == 0: # f is a factor of n
                    if not f in factor_list:
                        factor_list.append(f)
            # now we have a list of factors for n
            if n == sum(factor_list): # n is a perfect
number
                perfect_list.append([n, factor_list]) #
keep factors too for checking
        # now we have found all the perfect numbers in
the range
        print(f"Perfect numbers found between {a} and
{b}:")
        for n in perfect_list:
            print(n, end=" ")

    # handle the possible errors and exceptions
    except ValueError:
        print("Invalid input. Please enter numbers
only.")
    except ZeroDivisionError:
        print("Cannot divide by zero. Please enter a
nonzero number.")
    except Exception as e:
        print(f"An unexpected error occurred: {e}")
```

When running the program with Jupyter Notebook in VS Code, by pressing Ctrl+Enter and inputting 3 and 1000 for the two numbers requested, the following output will be produced:

```
Perfect numbers found between 3 and 1000:
[6, [1, 2, 3]] [28, [1, 2, 4, 7, 14]] [496, [1, 2, 4, 8, 16, 31, 62, 124, 248]]
```

The output shows both the perfect numbers and a list of their factors.

4.2 Handling Runtime Errors and Exceptions

The runtime errors and exceptions discussed can be fatal or nonfatal. By a nonfatal error, we mean that when the error occurs, the program can still continue if it is properly handled. In contrast, when a fatal error occurs, the program stops and has to be fixed before you can restart the program.

https://doi.org/10.15215/remix/9781998944088.01

When programming, since runtime errors (or "exceptions," to use a more techie term) are unavoidable, the best a programmer can do is to have all exceptions caught and handled properly when they occur. Otherwise, what renders to the users when an error has occurred could be overwhelming and irritating. Consider the following code example, which simply asks for a numeric grade.

```
In [ ]:  grade = int(input('Please input your grade: '))
```

When a letter grade is entered instead, a runtime error occurred, and Python interpreter or Python Virtual Machine (PVM) renders a bunch of information about the error, as shown below:

```
Out [ ]: Please input your grade: A
         ---------------------------------
         ValueError  Traceback (most recent call last)
         <ipython-input-13-d0126eb94acb> in <module>
         ----> 1 grade = int(input('Please input your grade: '))

         ValueError: invalid literal for int() with base-10: 'A'
```

Although the information above may be helpful for programmers to debug the code and correct the error, it is completely useless for ordinary users. For that reason, all modern programming languages provide some way to handle runtime errors and exceptions so that only sensible information can be seen by users and, in most cases, programs can continue and recover from errors peacefully, without crashing.

Similar to those in other modern programming languages, Python exceptions are handled with a ***try*** statement. The general coding structure of ***try*** statements is as follows:

```
# the try clause is for enclosing a code block of the
program
# in which errors may occur
try:
  <code block in the normal code flow>   # things to do
normally

except <exception/error 1>:   # the except clause is for
handling errors
```

https://doi.org/10.15215/remix/9781998944088.01

```
  <code block for exception 1>   # things to do if
exception 1 happens

except <exception/error 2>:   # except clause may have a
list of errors
  <code block for exception 2>   # things to do if
exception 2 happens

except <exception/error 3>:   # there can be more
exceptions
  <code block for exception 3>   # things to do if
exception 3 happens

else:   # do something when no error has occurred
  <code block for else>   # things to do when no
exceptions are raised

finally:   # finally clause allows to do something
regardless of the above
  <code block for finally>   # things to do regardless of
the above
```

As shown, a ***try*** statement starts with a ***try*** clause, which encloses the code that you want to run for the application but may cause errors and raise exceptions.

Right after the ***try*** clause is one or more exception clauses, each of which starts with the keyword "except," followed by a system-defined error/exception name. The code of each exception clause specifies what you want to do if this error happens.

In [ ]:
```
try:
  grade = int(input('Please input your grade: '))
except ValueError as e:
  print('Exception has been raised! ', e)
```

Out [ ]:
```
Please input your grade: A
Exception has been raised! invalid literal for int() with base-10: 'A's
```

Note that in the code above, ValueError is the name of a built-in exception class. We put this code in a ***try*** statement because you do not know whether the user will type an integer as expected. If a nonnumber is typed, an error will occur for function int.

https://doi.org/10.15215/remix/9781998944088.01

The ***else*** clause encloses the code specifying what to do if no errors are caught by all the exception blocks.

The ***finally*** clause encloses the code that will run regardless of whether any exception has been raised within the ***try*** clause.

Now comes the question of when you need to put a code block in a ***try*** statement. You cannot put every piece of code in a ***try*** statement. The rule is that if a statement or block of code may cause an error during runtime, especially when the possibility is out of your control, the statement or code should be enclosed in a ***try*** statement and each of the possible errors should be handled properly. Examples of such a possibility may include the following situations:

1. Statements involve user input, because you never know if a user will do what you expected.
2. Statements involve the use of files, because there is no guarantee as to the status of the file. It may have been deleted when you want to read it; it may already have existed when you want to create a new one with same name.
3. Statements involve the use of resources on the web or on the internet in general because the resources may no longer be available.
4. Statements involve the use of numbers from user input or involve calculation, where 0 may end up as a denominator.
5. Statements involve extensive use of computer memory, which may lead to the depletion of RAM.

Chapter Summary

- Programs can have syntax errors and logic errors.
- Errors that are found when running the program are runtime errors.
- A good IDE can often help you avoid many syntax errors, including undefined names, incorrect indentation, incorrect formation of statements, and more. If there are syntax errors in your program, look for marks that your IDE may have added to indicate them in your code.
- Logic errors are caused by incorrect problem solving or programming logic. Examples include values that are out of the intended range and the incorrect use of operators or functions.
- Some logic errors can raise exceptions at runtime. For example, if an index variable for a sequence (string, list, or tuple) is out of range, an exception will be raised.
- Some logic errors can only be discovered by programmers, system testers, or users who notice an unexpected result or behaviour in the system.

https://doi.org/10.15215/remix/9781998944088.01

- A Python interpreter or virtual machine will raise exceptions when a runtime error has occurred. If exceptions are not caught and properly handled, the application or system will crash and become very ugly.
- Python and many other modern programming languages use ***try-except*** statements to handle exceptions.
- In Python, a ***try*** statement may take one of five forms: ***try-except***, ***try-except-else***, ***try-except-finally***, ***try-except-else-finally***, and ***try-finally***.
- Between ***try*** and ***except***, there should be a code block that may have runtime errors (such as incorrect input from users, files, or network sockets). The ***try*** keyword and the code block together form a ***try*** clause; right behind the ***except*** keyword, there should be the name or names of errors, followed by a code block to be run when the named error or errors occurred, which together form an ***except*** clause.
- A ***try*** clause can be followed by multiple ***except*** clauses.
- An ***except*** clause can be followed by an ***else*** clause, which consists of the ***else*** keyword and a code block to be run when an error has occurred but didn't match any of the named errors in the except clauses.
- The ***finally*** clause can be used as a final clause of a ***try*** statement. It consists of the ***finally*** keyword and a code block that is to be run in all circumstances, whether an error has occurred or not.

Exercises

1. Suppose that you want to get an integer from a user, but you are concerned that the user may type something else other than an integer. Write a piece of code, as concise as possible, that asks the user for input until an integer is entered by the user.
2. What error will occur when running the following code?
   a. `s = 8 + 'a'`
   b. `students = ['John', 'May', 'Jim']`
   c. `total = sum(12, 90, 32, 'one hundred')`
3. What's wrong with the following code?

```
idx = 1
product = 0
while idx>10:
 product *= idx
 idx++
print(product)
```

https://doi.org/10.15215/remix/9781998944088.01

Chapter 5

Use Sequences, Sets, Dictionaries, and Text Files

Chapter 5 details how compound data types and files can be used in programming to solve problems. Data need to be structured and organized to represent certain kinds of information and to make problem solving, information processing, and computing possible and more efficient. In addition to integer, float, and complex numbers, Python provides compound data types to represent more complicated information. These compound data types include strings, lists, tuples, sets, and dictionaries, as well as files that can be used to store a large volume of data in the long term (after the computer is shut off).

Learning Objectives

After completing this chapter, you should be able to

- explain sequences.
- explain strings and the methods and functions that can be applied to them.
- construct and format strings with the ***f*** prefix and the format method.
- discuss lists and tuples and the differences between the two.
- properly use the methods and functions of lists and tuples.
- explain sets and dictionaries and discuss the methods and functions that can be used on them.
- explain files and discuss the differences between text files and binary files and the methods and functions available for manipulating files.
- use strings, lists, tuples, sets, dictionaries, and files in problem solving and system design and development with Python.

https://doi.org/10.15215/remix/9781998944088.01

5.1 Strings

The string is one of the most important data types for information representation and processing. Strings are the base of information and data, and they were used to structure the sequences of characters for the ASCII table in the early days of modern computers. They are still used the same way now in UTF-8 (8-bit Unicode Transformation Format Unicode), which includes characters for all human languages in the world.

Because strings are sequences of characters, the characters are ordered and indexed. We can access and manipulate individual characters through these indexes, starting from 0, as shown in the following example:

```
>>> name = "John Doe"
>>> name[0]
"J"
>>> name[3]
"n"
```

To construct a string from another data type, you use built-in function str(), as shown in the following example:

```
>>> tax_rate = 0.16
>>> tax_string = str(tax_rate)
>>> tax_string
'0.16'
>>> type(tax_string)
<class 'str'>
```

Methods of Built-In Class str

As is the case with some other object-oriented programming languages, string is a built-in class but is named str in Python. The str class has many powerful methods built into it, as detailed below with coding samples.

S.CAPITALIZE()

This converts the first character of the first word of string s to upper case and returns the converted string. Please note that characters in string s remain unchanged. This is the same for all string methods: no string method will alter the content of the original string variable. Rather, the method will make a copy of the content, manipulate the copy, and return it.

https://doi.org/10.15215/remix/9781998944088.01

```
>>> s = "intro to programming with python"
>>> s_capitalized = s.capitalize()
>>> s_capitalized
'Intro to programming with python'
>>> s
'intro to programming with python'
```

S.CASEFOLD()

Converts all characters of string s into lower case and returns the converted characters.

```
>>> s_capitalized
'Intro to programming with python'
>>> s_capitalized.casefold()
'intro to programming with python'
```

S.CENTER(SPACE)

Returns a string centred within the given space. Note how the empty whitespace is divided when the number is not even.

```
>>> s="hello"
>>> s.center(10)
' hello '
```

S.COUNT(SUB)

Returns the number of times a specified value occurs in a string.

```
>>> s = "intro to programming with python"
>>> s.count('i')
3
>>> s.count('in')
2
```

S.ENCODE()

Returns an encoded version of characters if they are not in the standard ASCII table. In the example below, there are Chinese characters in the string assigned to variable es.

```
>>> es = "Python is not a big snake (蟒蛇)"
>>> print(cs.encode())
b'Python is not a big snake \xe8\x9f\x92\xe8\x9b\x87'
```

https://doi.org/10.15215/remix/9781998944088.01

Please note that the b in b'Python is not a big snake \xe8\x9f\x92\xe8\x9b\x87' indicates that all non-ASCII characters in the string are in byte.

S.ENDSWITH(SUB)

Returns true if the string ends with the specified value, such as a question mark.

```
>>> cs = "Is Python an animal?"
>>> print(cs.endswith('?'))
True
```

S.EXPANDTABS(TS)

Sets the size of tabs in the string to ts, which is an integer.

```
>>> cs = "Is\t Python\t an\t animal?"
>>> cs
'Is\t Python\t an\t animal?'
>>> print(cs)
Is Python an animal?
>>> print(cs.expandtabs(10))
Is  Python  an  animal?
```

S.FIND(SUB)

Searches the string for a substring and returns the position of where it was found.

```
>>> s= 'intro to programming with python'
>>> s.find("ro")
3
```

S.FORMAT(*ARGS, **KWARGS)

Formats specified values given in the list of position arguments *args, and/or the list of keyword arguments **kwargs into string s, according to the formatting specs given in s.

This is very useful in constructing complex strings.

```
>>> "Hello {0}, you are {1:5.2f} years
old.".format("Python", 23.5)
'Hello Python, you are 23.50 years old.'
```

https://doi.org/10.15215/remix/9781998944088.01

Please note that when mapping a dictionary, s.format(**mapping) can be used to format a string by mapping values of the Python dictionary to its keys.

```
>>> point = {'x':9,'y':-10} # point is a dictionary
>>> print('{x} {y}'.format(**point))
9 -10
```

Please note that ** has converted the dictionary point into a list of keyword arguments. This formatting can also be done by directly using keyword arguments:

```
>>> print('{x} {y}'.format(x=9,y=-10))
9 -10
```

S.FORMAT_MAP(MAPPING)

Similar to format(**mapping) above. The only difference is that this one takes a dictionary without operator **.

```
>>> point = {'x':9,'y':-10}
>>> print('{x} {y}'.format_map(point))
9 -10
```

S.INDEX(SUB)

Searches the string for a substring and returns the position of the substring. Generates a return error if there is no such substring.

Note that this may not be a good method to test if one string is a substring of another.

```
>>> s= 'intro to programming with python'
'intro to programming with python'
>>> s.index("ing")
17
>>> s.index('w')
21
>>> s.index('z')
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
ValueError: substring not found
```

https://doi.org/10.15215/remix/9781998944088.01

S.ISALNUM()

Returns True if all characters in the string are alphanumeric.

```
>>> "98765".isalnum()
True
>>> "98765abcde".isalnum()
True
>>> "98765<>abcde".isalnum()
False
```

S.ISALPHA()

Returns True if all characters in the string are in the alphabet, including Unicode characters.

```
>>> "abcde".isalpha()
True
>>> "abcTde".isalpha()
True
>>> "abc35Tde".isalpha()
False
>>> "abc他Tde".isalpha()
True
```

S.ISDECIMAL()

Returns True if all characters in the string are decimals.

```
>>> "1235".isdecimal()
True
>>> "1235.65".isdecimal()
False
>>> "1235.65e".isdecimal()
False
```

S.ISDIGIT()

Returns True if all characters in the string are digits.

```
>>> "123565".isdigit()
True
>>> "1235.65".isdigit()
False
```

https://doi.org/10.15215/remix/9781998944088.01

```
>>> "1235y65".isdigit()
False
```

S.ISIDENTIFIER()

Returns True if the string is an identifier by Python's definition.

```
>>> "w1235y65".isidentifier()
True
>>> "9t1235y65".isidentifier()
False
>>> "w1235_y65".isidentifier()
True
```

S.ISLOWER()

Returns True if all characters in the string are lower case.

```
>>> "w1235y65".isidentifier()
True
>>> "9t1235y65".isidentifier()
False
>>> "w1235_y65".isidentifier()
True
```

S.ISNUMERIC()

Returns True if all characters in the string are numeric.

```
>>> "123565".isnumeric()
True
>>> "1235.65".isnumeric()
False
>>> "123565nine".isnumeric()
False
```

S.ISPRINTABLE()

Returns True if all characters in the string are printable.

```
>>> "123565nine".isprintable()
True
>>> "123565 all printable".isprintable()
True
```

https://doi.org/10.15215/remix/9781998944088.01

```
>>> "123565 all printable<>!@#$%^&**".isprintable()
True
```

S.ISSPACE()

Returns True if all characters in the string are whitespace.

```
>>> "  ".isspace()
True
>>> " \t  ".isspace()
True
>>> " \t  \n".isspace()
True
>>> " \t  m\n".isspace())
False
```

S.ISTITLE()

Returns True if the string follows the rules of a title—that is, the first letter of each word is upper case, while the rest are not.

```
>>> "Python Is a Great Language".istitle()
False
>>> "Python Is A Great Language".istitle()
True
```

S.ISUPPER()

Returns True if all characters in the string are upper case.

```
>>> "THIS IS ALL UPPER".isupper()
True
>>> "THIS IS ALL UPPER with some lower".isupper()
False
```

SEP.JOIN(ITERABLE)

Joins the elements of an iterable with the separator. The iterable can be a list, tuple, string, dictionary, or set. Note that each element of the iterable must be a string. An integer or other number will raise an error.

```
>>> "-".join([" for", " programming!"])
'for- programming!'
>>> "&".join([" for", " programming!"])
```

https://doi.org/10.15215/remix/9781998944088.01

```
'for& programming!'
>>> "%".join([" for", " programming!"])
'for% programming!'
>>> "%".join(" for programming!")
'%f%o%r% %p%r%o%g%r%a%m%m%i%n%g%!'
>>> "%".join(('a', '2', '3'))
'a%2%3'
>>> "%".join({'a', '2', '3'})
'3%a%2'
>>> "%".join({'a':'mnmn', '2':'987', '3':'43322'})
'a%2%3'
```

S.LJUST(SL)

Returns a left-justified version of the string within the given size of space.

```
>>> "Python Is A Great Language".ljust(30)
'Python Is A Great Language  '
```

S.LOWER()

Converts a string into lower case.

```
>>> "Python Is A Great Language".lower()
'python is a great language'
```

S.LSTRIP()

Returns a left trim version of the string.

```
>>> "  Python Is A Great Language  ".lstrip()
'Python Is A Great Language  '
```

S.MAKETRANS(DICT)
S.MAKETRANS(S1, S2)

Return a translation table to be used in translations.
In s.maketrans(dict), key-value pairs of dict provide mapping for translation; in the case of s.maketrans(s1, s2), chars in s1 are mapped to chars in s2 one by one.

```
>>> "Python Is A Great Language".maketrans({'a':'b',
'c':'d'})
{97: 'b', 99: 'd'}
>>> "Python Is A Great Language".maketrans('ab', 'cd')
{97: 99, 98: 100}
```

https://doi.org/10.15215/remix/9781998944088.01

S.PARTITION(SUB)

Returns a tuple where the string is divided into three parts with sub in the middle.

```
>>> "Python Is A Great Language".partition('A')
('Python Is ', 'A', ' Great Language')
```

S.REPLACE(S1, S2)

Returns a string where a specified value is replaced with a specified value.

```
>>> "Python Is A Great Language".replace('Great',
'Powerful')
'Python Is A Powerful Language'
```

S.RFIND(SUB)

Searches the string from the right for a substring and returns the position of where it was first found.

```
>>> "Python Is A Great Language".rfind('g')
24
```

S.RINDEX(SUB)

Searches the string from the right for a substring and returns the index of the substring where it was first found.

```
>>> "Python Is A Great Language".rindex('g')
24
```

S.RJUST(SUB)

Returns a right-justified version of the string within the given size of space.

```
>>> "Python Is A Great Language".rjust(35)
'   Python Is A Great Language'
```

S.RPARTITION(SUB)

Returns a tuple where the string is divided into three parts at the substring found from the right.

```
>>> "Python Is A Great Language".rpartition('g')
('Python Is A Great Langua', 'g', 'e')
```

https://doi.org/10.15215/remix/9781998944088.01

S.RSPLIT(SEP)

Splits the string at the specified separator and returns a list.

```
>>> "Python Is A Great Language".rsplit('g')
['Python Is A Great Lan', 'ua', 'e']
```

S.RSTRIP()

Returns a right-trimmed version of the string.

```
>>> "Python Is A Great Language ".rstrip()
'Python Is A Great Language'
```

S.SPLIT(SEP)

Splits the string at the specified separator and returns a list.

```
>>> "Python Is A Great Language".split('g')
['Python Is A Great Lan', 'ua', 'e']
```

S.SPLITLINES()

Splits the string at line breaks and returns a list.

```
>>> "Python Is A Great Language.\n I love
it.".splitlines()
['Python Is A Great Language.', ' I love it.']
```

S.STARTSWITH(SS)

Returns true if the string starts with the specified value.

```
>>> "Python Is A Great Language".startswith('g')
False
>>> "Python Is A Great Language".startswith('P')
True
```

S.STRIP()

Returns a trimmed version of the string.

```
>>> " Python Is A Great  Language ".strip()
'Python Is A Great  Language'
```

https://doi.org/10.15215/remix/9781998944088.01

S.SWAPCASE()

Swaps cases, so lower case becomes upper case, and vice versa.

```
>>> "Python Is A Great Language".swapcase()
'pYTHON iS a gREAT lANGUAGE'
```

S.TITLE()

Converts the first character of each word to upper case.

```
>>> 'pYTHON iS a gREAT lANGUAGE'.title()
'Python Is A Great Language'
```

S.TRANSLATE()

Returns a string translated from s using a translation table created with the maketrans() method.

```
>>> table = "".maketrans("ab", 'cd')
>>> print("Python Is A Great Language".translate(table))
Python Is A Grect Lcngucge
```

S.UPPER()

Converts a string into upper case.

```
>>> "Python Is A Great Language".upper()
'PYTHON IS A GREAT LANGUAGE'
```

S.ZFILL(SL)

Fills the string to a specific length with a specified number of 0s at the beginning.

```
>>> "Python Is A Great Language".zfill(39)
'0000000000000Python Is A Great Language'
```

Built-In Functions and Operators for Strings

In addition to the string methods that you can use to manipulate strings, there are built-in functions and operators. The following are some examples.

USE OPERATOR + TO JOIN STRINGS TOGETHER

```
>>> "Python is a good language " + "for first-time
programming learners."
'Python is a good language for first-time programming
learners.'
```

https://doi.org/10.15215/remix/9781998944088.01

USE OPERATOR * TO DUPLICATE A STRING

```
>>> "Python! "*3
'Python! Python! Python! '
```

USE BUILT-IN FUNCTION LEN(S) TO FIND OUT THE LENGTH IF A STRING

```
>>> p_string = "Python is a good language " + "for first-
time programming learners."
>>> len(p_string)
62
```

USE OPERATOR [I:J] TO SLICE A STRING

```
>>> p_string[0:5]   # slice begins at index 0 till index
5 but excluding 5
'Pytho'
>>> p_string[5:25]   # slice begins at index 5 till index
25 but excluding 25
'n is a good language'
>>> p_string[:16]   # when the starting index point is
missing, 0 is assumed
'Python is a good'
>>> p_string[6:]   # when the ending index point is
missing, the string is copied from the start to the end
'is a good language for first-time programming
learners.'
>>> p_string[:]   # when both indexes are missing, the
entire string is copied
'Python is a good language for first-time programming
learners.'
>>> p_string[:-1] #the result is the same as using [:]
'Python is a good language for first-time programming
learners'
```

Table 5-1 summarizes the operators and built-in functions you can use to manipulate strings.

https://doi.org/10.15215/remix/9781998944088.01

Table 5-1: String-related operators and built-in functions

Operators and built-in functions on string	Operation	Code samples in Python interactive mode

s[n]	Get the character at nth position (n is an integer)	`>>> name = "John Doe"`
		`>>> name[2]`
		`'h'`

s[start:end]	Get a slice of the string. Negative indexes can also be used to count from the end.	`>>> name = "John Doe"`
		`>>> name[2:6]`
		`'hn D'`
		`>>> name = "John Doe"`
		`>>> name[:-3]`
		`'John '`

s1 + s2	Concatenate two strings together	`>>> first_name = "John"`
		`>>> last_name = "Doe"`
		`>>> first_name +' '+`
		`  last_name`
		`'John Doe'`

s * n	Duplicate s n times	`>>> (name +' ')*3` `'John Doe John Doe John Doe '`
s1 in s	Test if s1 is a substring of s	`>>> first_name in name` `True`
len(s)	Get the length of string s	`>>> len(name)` `8`

print(s)	Print string s	`>>> name = "John Doe"`
		`>>> print(name)`
		`John Doe`

In addition to the ways discussed above to construct and manipulate strings, Python also provides some methods for constructing and formatting strings nicely.

Constructing and Formatting Strings

Because text is made of strings, and text is very important in representing data, information, and knowledge, there is a need to convert various data objects into well-formatted strings. For this purpose, Python has provided programmers with a very powerful means of formatting strings that may consist of various types of data such as literals, integer numbers, float numbers with different precisions, compound data, and even user-defined objects.

https://doi.org/10.15215/remix/9781998944088.01

In 2.1, we saw how we could use f/F, r/R, u/U, and b/B to provide some direction on string formation and representation. We also saw that prefixing f or F to a string allows us to conveniently embed expressions into the string with { } and have the expressions be automatically evaluated. In the following, you will see two more ways of formatting strings.

FORMATTING WITH %-LED PLACEHOLDERS

Let's begin with an example to explain how %-led placeholders are used to format and construct strings:

```
In [ ]:  d, n = 5.689, 8   # assigning values to variable d and
          n
         s0 = "n has a value of %3d, and d has a value of
          %9.5f"%(n, d)   # with %-led placeholders
         print(s0)   # s is evaluated

Out [ ]: n has a value of 8, and d has a value of 5.68900
```

In the example above, the string before the last percentage sign % is called a formatting string. %3d is a %-led placeholder for an integer that will take a 3-digit spot, whereas %9.5 is a %-led placeholder for a float number, where 9 specifies the total number of digits the float number will take and 5 specifies the total number of decimal digits. The values in the tuple behind the last percentage sign are to be converted and placed into their corresponding placeholders. In the example, the value of n will be converted to an integer and placed into the first placeholder, whereas the value of d will be converted into a float number and placed into the second placeholder.

You can also use named placeholders, as shown in the next example, where the course and language (in the parentheses) are the names of the placeholders.

```
In [ ]:  course_number = 'comp218'
         language = 'Python'
         s1 = '%(course)7s - introduction to programming in
          %(language)s '%{'course':course_number,
         'language':language}
         print(s1)

Out [ ]: comp218 - introduction to programming in Python
```

Note that when named placeholders are used, you will need to use dictionary instead of a tuple behind the last percentage sign.

The general format of a %-led placeholder is as follows:

https://doi.org/10.15215/remix/9781998944088.01

```
%[flags][width] [.precision] type
```

or the following, if you like to use named placeholders:

```
%[(name)][flags][width] [.precision] type
```

The flags may use one or a combination of the characters in Table 5-2.

Table 5-2: Flags used in placeholders in formatting strings

Flag	Meaning	Code sample

#	Used with b, o, x, or X, this specifies that the formatted value is preceded with 0b, 0o, 0x, or 0X, respectively.	`>>> data = ("COMP 218", 10)`
		`>>> '%s has %#X units.' % data`
		`'COMP 218 has 0XA units.'`

0	The conversion result will be zero-padded for numeric values.	`>>> data = {"course": "COMP 218", "number": 10}`
		`>>> '%(course)s has %(number)016d units.' % data`
		`'COMP 218 has 0000000000000010 units.'`

-	The converted value is left-adjusted.	`>>> data = {"course": "COMP 218", "number": 10}`
		`>>> '%(course)s has %(number)-6d units.' % data`
		`'COMP 218 has 10  units.'`

	If no sign (e.g., a minus sign) is going to be written, a blank space is inserted before the value.	`>>> data = {"course": "COMP 218", "number": -10}`
		`>>> '%(course)s has %(number)16d units.' % data`
		`'COMP 218 has  10 units.'`

+	The converted value is right-adjusted, and a sign character (+ or -, depending on whether the converted value is positive or negative) will precede it.	`>>> data = {"course": "COMP 218", "number": -10}`
		`>>> '%(course)s has %(number)+6d units.' % data`
		`'COMP 218 has  -10 units.'`

The width is the total width of space held for the corresponding value, and precision is the number of digits that the decimal portion will take if the value is a float number. The type can be one of the types shown in Table 5-3.

https://doi.org/10.15215/remix/9781998944088.01

Table 5-3: Types used in placeholders for formatting strings

Conversion	Meaning	Code sample

d, i, or u	Signed integer decimal.
	Note that in the last three coding samples, the plus sign has been automatically removed in the printout.
		`>>> print("%+d"% (88))`
		`+88`
		`>>> print("%+i"% (88))`
		`+88`
		`>>> print("%+u"% (88))`
		`+88`
		`>>> print("%+u"% (-88))`
		`-88`
		`>>> print("%+i"% (-88))`
		`-88`
		`>>> print("%+d"% (-88 ))`
		`-88`

o	Unsigned octal.	`>>> print("%10o"% (25))`
		`31`
		`>>> print("%10.3o"%`
		`(25))`
		`031`
		`>>> print("%10.5o"%`
		`(25))`
		`00031`

X or x	Unsigned hexadecimal.	`>>> print("%6.5X"% (88))`
		`00058`
		`>>> print("%6.5x"% (88))`
		`00058`
		`>>> print("%#5X"% (88))`
		`0X58`
		`>>> print("%5X"% (88))`
		`58`

E or e	Floating-point exponential format (lower case or upper case).	`>>> print("%10.3e"%`
		`(123456.789))`
		`1.235e+05`
		`>>> print("%10.3E"%`
		`(123456.789))`
		`1.235E+05`

F or f	Floating-point decimal format.	`>>> print("%13.5f"%`
		`(123456.789))`
		`123456.78900`
		`>>> print("%13.5F"%`
		`(123456.789))`
		`123456.78900`

G or g	Same as E or e if exponent is greater than −4 or less than precision; F otherwise.	`>>> print("%13.5g"%`
		`(123456.789))`
		`1.2346e+05`
		`>>> print("%13.5G"%`
		`(123456.789))`
		`1.2346E+05`

(continued on next page)
https://doi.org/10.15215/remix/9781998944088.01

Table 5-3: Types used in placeholders for formatting strings *(continued)*

Conversion	Meaning	Code sample

c	Single character (accepts integer or single character string).	`>>> s = 'Python is great!'`
		`>>> 'The first character of string %s is %c'%(s, s[0])`
		`'The first character of string Python is great! is P'`

r	String (converts any python object using repr() or __repr__(), instead of str() or __str__()). In class definition, you need to implement the dunder method __repr__ in order for repr() or __repr__() to work on the objects of the defined class.	`>>> 'The complex number will be displayed as %r'%(cn)` `'The complex number will be displayed as (12+35j)'`
s	String (converts any python object using str()). We will see the difference between %r and %s when we defined __repr__ method for a user-defined class.	`>>> 'The complex number will be displayed as %s'%(cn)` `'The complex number will be displayed as (12+35j)'`
%%	No argument is converted (results in a "%" character in the result). It works only if the formatting is complete.	`>>> '%% will be displayed as a single percentage sign, and the complex number is %s'%(cn)` `'% will be displayed as a single percentage sign, and the complex number is (12+35j)'`

FORMATTING STRINGS WITH THE *FORMAT* METHOD

Compared to the two methods we have seen so far, a more formal way of string formatting in Python is using the ***format*** method, as shown in the following example:

```
>>> s = "{0} is the first integer; {1} is the second integer".format(88, 99)
>>> s
'88 is the first integer; 99 is the second integer'
```

https://doi.org/10.15215/remix/9781998944088.01

The {} in the above example is also called a placeholder or replacement field. You can index the placeholders with integer numbers starting from 0, corresponding to the positions of values. You can also name the placeholders, in which case dictionary or keywords arguments need to be used within the ***format*** method call. In the example above, if we switch the indices (0 and 1), 99 will be placed as the first integer and 88 will be placed as the second integer, as shown below:

```
>>> s = "{1} is the first integer; {0} is the second
integer".format(88, 99)
>>> print(s)
99 is the first integer; 88 is the second integer
```

The general form of the replacement field is as follows:

{[field_name] [! conversion] [: format_spec]}

As mentioned before, having the item inside [] is optional; a placeholder can be as simple as an empty {}, as shown in the following example:

```
>>> 'X: {}; Y: {}'.format(3, 5)
'X: 3; Y: 5'
```

In the general form of the replacement field above, *field name* is something that can be used to identify the object within the arguments of the ***format*** method. It can be an integer to identify the position of the object, a name if keyword arguments are used, or a dot notation referring to any attribute of the object, as shown in the following example:

```
>>> c = 23 - 35j
>>> ('The complex number {0} has a real part {0.real} and
an imaginary part {0.imag}.').format(c)
'The complex number (23 - 35j) has a real part 23.0 and
an imaginary part -35.0.'
```

In this string formatting example, the first placeholder is {0}, in which integer 0 indicates that the value of the first argument of the ***format*** method call will be placed here; the second placeholder is {0.real}, which indicates that the value of the attribute real of the first object pf the ***format*** method call will be converted and inserted in that location; and the third placeholder is

https://doi.org/10.15215/remix/9781998944088.01

{0.imag}, which indicates that the value of the attribute imag of the first object of the ***format*** method call will be converted and inserted in that location. It is up to the programmer to use the right attribute names or to compose the right reference to a valid object or value within the arguments of the ***format*** method call.

Please note that *conversion* in the general form of the replacement field above is led by an exclamation mark !, which is followed by a letter: r, s, or a. The combination !r is used to convert the value into a raw string, !s is used to convert the value into a normal string, and !a is used to convert the value into standard ASCII, as shown in the following examples:

```
>>> print('{!r} is displayed as a raw string'.format('\t
is not tab, \n is not newline'))
'\t is not tab, \n is not newline' is displayed as a raw
string.

>>> print('{!s} is not displayed as a raw string'.
format('\t is a tab, \n is a new line'))
is a tab,
is a new line is not displayed as a raw string.

>>> print('{!s} is displayed in Chinese'.format('Python
is not 大蟒蛇.'))
Python is not 大蟒蛇. is displayed in Chinese.

>>> print('{!a} is displayed as an ASCII string'.
format('Python is not 大蟒蛇.'))
'Python is not \u5927\u87d2\u86c7.' is displayed as an
ASCII string.
```

Please note the difference between the two outputs using !s and !a in particular.

It may also have been noted that with !r, the quotation marks surrounding the argument remain in the output, whereas with !s, the quotation marks have disappeared from the output. This is true when the argument for the !r is a string.

When the argument for !r is not a string, especially when it is a complicated object, o, the !r will cause the placeholder to be replaced with the result of o.repr(), which in turn calls the dunder method __repr__() defined for the

https://doi.org/10.15215/remix/9781998944088.01

object's class. You will learn how to define and use Python dunder methods later in Chapter 7.

In string formatting with format method, formatting specification is led by a colon :, which is followed by formatting instructions, including the following:

1. justification or alignment: > for right justification, < for left justification, ^ for centre justification
2. with/without sign for numbers: + for always showing the sign, − for only show the minus sign, and '' for showing a whitespace when the number is positive
3. the total number of letter spaces allocated to the data, such as in {:6d}, where 6 specifies that 6 letter spaces are taken by the integer number
4. the number of decimal digits for float numbers, such as in {:6.2f}, in which the 2 specifies the decimal, so the float number will be rounded to take 2 spaces
5. data type conversion indicates what data will be converted and inserted into the placeholder; the types of data include
   a. s for string
   b. d for integer
   c. f for float number
   d. x or X for hex number
   e. o for octal number
   f. b for binary number
   g. #x, #X, #o, and #b to prefix the numbers 0x, 0X, 0o, and 0b, respectively

The following example shows how the data type conversions work:

```
>>> '{:+12.8f}, {:+f}, {:#b}, {:#X}'.format(2.71828182,
-3.14, 78, 127)
' +2.71828182, -3.140000, 0b1001110, 0X7F'
```

If you wish the output of a placeholder to be left, right, or centre justified within the given space, <, >, or ^ can be used to lead the format spec, as shown in the following example:

```
>>> '{:<+22.8f}, {:+f}, {:#b}, {:#X}'.format(2.71828182,
-3.14, 78, 127)
'+2.71828182   , -3.140000, 0b1001110, 0X7F'
```

https://doi.org/10.15215/remix/9781998944088.01

If you want the extra space to be filled with a special character, such as #, you can put the character between the colon and <, >, or ^, as shown below:

```
>>> '{:#^+22.8f}, {:+f}, {:#b}, {:#X}'.format(2.71828182,
-3.14, 78, 127)
'#####+2.71828182######, -3.140000, 0b1001110, 0X7F'
```

By this point, we have learned three ways of constructing and formatting strings: the first one is to use the f/F prefix, the second is to use a %-led placeholder, and the last is to use the format method.

Among the three, the first one is the most compact and good for simple string construction without any fancy formatting. The expression within each {} will be evaluated, and the value will be converted into a string with which the placeholder is replaced as is.

Both the second and the third way can be used to construct and format more complex strings from various objects. The difference between the two is that the second, using a %-led placeholder, is more casual, whereas the third is more formal and the code more readable.

Regular Expressions

Information processing and text manipulation are important uses for modern computers, and regular expressions, called "REs" or "regexes" for short, were developed as a powerful way to manipulate text. Many modern programming languages have special libraries for searching and manipulating text using regular expressions. In Python, a standard module called re was developed for that purpose.

To correctly use the re module, we first must understand what regular expressions are and how to construct a regular expression that genuinely defines the strings we want to find and/or manipulate within a text because almost all functions/methods of the re module are based on such defined regular expressions.

What is a regular expression? A regular expression is a pattern that describes certain text or literal strings. Examples of some useful patterns include telephone numbers, email addresses, URLs, and many others.

To be able to correctly define a regular expression precisely describing the strings we want to find and manipulate, we must first understand and remember the rules of regular expressions, as well as special characters and sequences that have special meanings in a re module. Since regular expressions are strings themselves, they should be quoted with single or double quotation marks. For

https://doi.org/10.15215/remix/9781998944088.01

simplicity, however, we may omit some quotation marks in our discussion when we know what we are talking about in the context.

Plain literals such as a, b, c,...z, A, B, C,...Z, and numeric digits such as 0, 1, 2,...9 can be directly used in a regular expression to construct a pattern, such as Python, Foo, Canada. Some symbols in the ASCII table have been given special meanings in re. These symbols, called metacharacters, are shown in Table 5-4.

Table 5-4: Metacharacters and basic rules for constructing regular expressions

Symbols	Meaning	Example
.	Match any character except \n, a new line, in a string.	t..t will match test, text,...
^	Affixed to a pattern to match the preceding regex if it is at the beginning of the string being searched.	^Hello will only match Hello when it is at the start of an email
$	Affixed to a pattern to match the preceding regex if it is at the end of a string.	mpeg$ will only match mpeg when it is at the end of a text
|	Match either the regex on the left or the regex on the right.	Wang|Wong will match either Wang or Wong
\	Form an escape sequence such as \d, \s,... with special meaning. Table 5-5 lists all the escape sequences defined in the re module. Also used to escape the metacharacters in this table back to their original meanings.	\d will match any single decimal digit \D is the negation of \d, meaning it will not match any single decimal digit
[...]	Define a set/class of characters.	[xyz] will match either x, y, or z. W[ao]ng is the same as Wang|Wong
[^...]	Define a set of characters excluded from the pattern. Inside and at the beginning of [], ^ is used as negation	[^A-Z\s] will match everything else except upper case letters and whitespace
[...x-y...]	Within [], define a range of characters from x to y	[0-9], [a-zA-Z]

(continued on next page)
https://doi.org/10.15215/remix/9781998944088.01

Table 5-4: Metacharacters and basic rules for constructing regular expressions *(continued)*

Symbols	Meaning	Example
(…)	Match enclosed regex and save as subgroup for later use.	(B|blah\s)+ will only match the second blah in "blah, blah and blah" and save it
?	This and the rest in this table are called quantifiers. When ? is affixed to a preceding regex (character or group) it becomes a nongreedy qualifier, meaning it will match only 0 or 1 occurrence of the preceding regex. ? can also be affixed to + as +?, or * as *?, to make + or * nongreedy.	mpe?g will match mpg or mpeg
*	Affixed to pattern meaning to match 0 or more (greedy) occurrences of preceding regular expression. Greedy means that it will match as many as possible.	=* will match 0 or more consecutive =s
+	Affixed to a pattern to match 1 or more occurrences of the preceding regular expression.	=+ will match 1 or more consecutive =s
{n}	Affixed to a pattern to match exactly n occurrences of the preceding regex.	[0-9]{3} will match the first 3 occurrences of digits, like an area code, for example
{m, n}	Affixed to a pattern to match from m to n occurrences of the preceding regex.	[0-9]{5, 11} will match all sequences of decimal digits that are 5 to 11 digits in length

Table 5-5: Escape sequences with special meanings in re

Escape sequence	Special meaning in re	Example
\d	Match any decimal digit 0-9.	Img\d+.jpg
\D	Opposite of \d, meaning do not match any decimal digit.	[\D] will match everything but decimal digits
\w	Match any alphanumeric character, A-Z, a-z, 0-9.	[_a-zA-Z]\w* will match all legitimate identifiers in Python

https://doi.org/10.15215/remix/9781998944088.01

Table 5-5: Escape sequences with special meanings in re *(continued)*

Escape sequence	Special meaning in re	Example
\W	Opposite of \w, meaning do not match any alphanumeric character.	[\W] will match everything but alphanumeric characters
\n	Match a new line whitespace.	\.\n will match all periods that end a paragraph
\t	Match a tab whitespace.	re.findall(r'\t', py_scripts) will find all the tabs in the py_scripts
\r	Match a return/enter whitespace.	re.findall(r'\r', article) will find all the return/enter whitespaces in the article.
\v	Match a vertical feed whitespace.	re.findall(r'\v', article) will find all the vertical feed whitespaces in the article
\f	Match a feed whitespace.	re.findall(r'\f', article) will find all the feed whitespaces in the article
\s	Match any of the whitespaces above.	re.findall(r'\s', article) will find all the whitespaces in the article
\S	Opposite of \s, \S matches any character which is not a whitespace character.	re.findall(r'\S', article) will find everything except whitespaces in the article
\N	N is an integer > 0. \1 refers to the first subgroup saved with (…).	In r'\b\w*(\w)\w*\1', \1 refers to the first found alphanumeric characters that appear more than once in a word
\b	Match any word boundary: the left boundary if \b is at the left of the pattern, the right boundary if \b is at the right side of the pattern	\bthe\b will match *the* if it is not part of other words
\B	Opposite of \b.	\bthe\B will match *the* if it is at the beginning of other words
\. \\ \+ *	Match a special symbol ., \, +, * respectively.	\d+*\d+ will match multiplications of two integers in a text
\A	Match at the start of a string, same as ^.	\AHello will match Hello if Hello is at the beginning of the string
\Z	Match at the end of a string, same as $.	\.com\Z will match .com if it is at the end of the string

https://doi.org/10.15215/remix/9781998944088.01

The above are the basic rules for constructing regular expressions or regex patterns. Using these rules, we can write regular expressions to define most string patterns we are interested in.

The following are some examples of regex patterns:

```
780-\d{7}, pattern for telephone numbers in Edmonton, Alberta
\$\d+\.\d{2}, pattern for currency representations in accounting
[A-Z]{3}-\d{3}, pattern for licence plate numbers
```

The re module is also empowered with the following extension rules, which all begin with a question mark ? within a pair of parentheses. Although surrounded by a pair of parentheses, an extension rule, except (?P<name>...), does not create a new group.

(?AILMSUX)

Here, ? is followed by one or more letters from set a, i, L, m, s, u, and x, setting the corresponding flags for the re engine. (?a) sets re.A, meaning ASCII-only matching; (?i) sets re.I, meaning ignore case when matching; (?L) sets re.L, meaning local dependent; (?m) sets re.M, meaning multiple lines; (?s) sets re.S, meaning dot matches all characters including newline; (?u) sets re.U, meaning Unicode matching; (?x) sets re.X, meaning verbose matching. These flags are defined in the re module. The details can be found by running help(re).

The flags can be used at the beginning of a regular expression in place of passing the optional flag arguments to re functions or methods of pattern object.

(?AILMSUX-IMSX:...)

Sets or removes the corresponding flags. (?a-u...) will remove Unicode matching.

(?:...)

Is a noncapturing version of regular parentheses, meaning the match cannot be retrieved or referenced later.

(?P<NAME>...)

Makes the substring matched by the group accessible by name.

(?P=NAME)

Matches the text matched earlier by given name.

https://doi.org/10.15215/remix/9781998944088.01

(?#...)

Is a comment; ignored.

(?=...)

Matches if... matches next but does not consume the string being searched, which means that the current position in string remains unchanged. This is called a lookahead assertion.

John (?=Doe) will match John only if it is followed by Doe.

(?!...)

Matches if... does not match next.

Jon (?!Doe) will match Jon only if it is not followed by Doe.

(?<=...)

Matches if preceded by... (must be fixed length).

(?<=John) Doe will find a match in John Doe because there is John before Doe.

(?<!...)

Matches if not preceded by... (must be fixed length).

(?<!John) Doe will find a match in Joe Doe because there is not Joe before Doe.

(?(ID)YES PATTERN | NO PATTERN)
(?(NAME)YES PATTERN | NO PATTERN)

Match yes pattern if the group with id or name is matched; match no pattern otherwise.

To do text manipulation and information processing using regular expressions in Python, we will need to use a module in the standard Python library called Re. Similarly, we will need to import the module before using it, as shown below:

```
>>> import re
```

Using the dir(re) statement, you can find out what names are defined in the module, as shown below, but you will need to use help(re) to find out the core functions and methods you can use from the re module.

The following are functions defined in the re module:

```
re.compile(pattern, flags=0)
```

Compile a pattern into a pattern object and return the compiled pattern object for more effective uses later.

https://doi.org/10.15215/remix/9781998944088.01

```
>>> import re
>>> pobj=re.compile('780-?\d{3}-?\d{4}')
>>> pobj.findall('780-9381396, 7804311508,
18663016227')   # findall method of pattern object
['780-9381396', '7804311508']
>>> b = re.compile(r'\d+\.\d*')
>>> b.match('32.23')   # match method of pattern object
<re.Match object; span=(0, 5), match='32.23'>
```

RE.MATCH(PATTERN, STRING, FLAGS=0)

Match a regular expression pattern to the beginning of a string. Return None if no match is found.

```
>>> r = re.match(r'\d+\.\d*', '123.89float')
>>> r
<re.Match object; span=(0, 6), match='123.89'>
```

RE.FULLMATCH(PATTERN, STRING, FLAGS=0)

Match a regular expression pattern to all of a string. Return None if no match is found.

```
>>> r = re.fullmatch(r'\d+\.\d*', '123.89')
# this will match
>>> r = re.fullmatch(r'\d+\.\d*', '123.89float')
# this will not match
```

RE.SEARCH(PATTERN, STRING, FLAGS=0)

Search a string for the presence of a pattern; return the first match object. Return None if no match is found.

```
>>> r = re.search(r'\d+\.\d+', 'real 123.89')
>>> r
<re.Match object; span=(5, 11), match='123.89'>
```

RE.SUB(PATTERN, REPLACING, STRING, COUNT=0, FLAGS=0)

Substitute occurrences of a pattern found in a string by replacing and return the resulted string.

```
>>> re.sub('t', 'T', 'Python is great.')
'PyThon is greaT.'
```

https://doi.org/10.15215/remix/9781998944088.01

RE.SUBN(PATTERN, REPLACING, STRING, COUNT=0, FLAGS=0)

Same as sub, but also return the number of substitutions made.

```
>>> re.subn('t', 'T', 'Python is great.')
('PyThon is greaT.', 2)
```

RE.SPLIT(PATTERN, STRING, MAXSPLIT=0, FLAGS=0)

Split a string by the occurrences of a pattern and return a list of substrings cut by the pattern.

```
>>> re.split(r'\W+', 'Python is great.')   # \W is
nonalphanumeric so it will get a list of words
['Python', 'is', 'great', '']
```

RE.FINDALL(PATTERN, STRING, FLAGS=0)

Find all occurrences of a pattern in a string and return a list of matches.

```
>>> re.findall('t', 'Python is great.')
['t', 't']
```

RE.FINDITER(PATTERN, STRING, FLAGS=0)

Return an iterator yielding a match object for each match.

```
>>> re.finditer('t', 'Python is great.')
<callable_iterator object at 0x00000198FE0F5FC8>
```

RE.PURGE()

Clear the regular expression cache.

```
>>> re.purge()
>>>
```

RE.ESCAPE(PATTERN)

Backslash all nonalphanumerics in a string.

```
>>> print(re.escape('1800.941.7896'))
1800\.941\.7896
```

Suppose we want to write a program to check if a name given by a user is a legitimate Python identifier. We can define a regex pattern for a Python identifier as shown below:

https://doi.org/10.15215/remix/9781998944088.01

```
idPatt = '\b_{0,2}[A-Za-z](_?[A-Za-z0-9])*_{0,2}\b\'
```

Before using the re module, we need to import it, as shown below:

```
import re
```

The next step will be to get an input from the user and test it:

```
name = input('Give me a name and I will tell you if it is
a Python identifier: ')
```

Trim the whitespace at the beginning and the end of the name just in case:

```
name = name.strip()  # this will strip the whitespaces
```

Then, we do the real test:

```
if re.match(idPatt, name) is not None:
print('Congratulations! It is!') else:
print('Sorry, it is not.')
```

The complete code of the program is shown in the code section of Table 5-6.

Table 5-6: Case study: How to check Python identifiers

The problem	In this case study, we will write a program to check if a name given by a user is legitimate Python identifier.
The analysis and design	Steps: Step 1: Import re module before using it Step 2: Define a regex pattern for Python identifiers Step 3: Get an input from the user, and Step 4: Test it with an ***if-else*** statement

The code	`import re`
	`idPatt = '(^[A-Za-z]\w+)$|(^_[A-Za-z]\`
	`  w+_$)|(^__[A-Za-z]\w+__$)'`
	`name = input('Give me a name and I will tell you`
	`  if it is a Python identifier:')`
	`name = name.strip()`
	`if re.match(idPatt, name) is not None:`
	`  print('Congratulations! It is!')`
	`else:`
	`  print('Sorry, it is not.')`

The result	Give me a name and I will tell you if it is a Python identifier:A2 Congratulations! It is!

https://doi.org/10.15215/remix/9781998944088.01

5.2 Lists

The list is an important compound data type in Python and in almost all programming languages, though not many programming languages have list as a built-in data type.

In previous sections, you saw a few program examples with a list involved. In the following, we explain the operators and functions that can be used on lists.

LIST(ITERABLE)

To construct a list from an iterable such as a sequence or call to range().

```
>>> l1 = list("test")
>>> l1
['t', 'e', 's', 't']
>>> l2 = list((1,2,3,4))
>>> l2
[1, 2, 3, 4]
>>> l5 = list(range(13, 26))
>>> l5
[13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25]
```

In addition, you can also create a list by directly putting items in a pair of square brackets, as shown below:

```
>>> l6 = ['Jon', 'John', 'Jonathan', 'Jim', 'James']
>>> l6
['Jon', 'John', 'Jonathan', 'Jim', 'James']
```

L[NTH]

To get the nth element of list l.

```
>>> students = ['John', 'Mary', 'Terry', 'Smith', 'Chris']
>>> students[3]
'Smith'
```

L[START:END]

To get a slice/sublist of l, including the members from a start position till right before the end position.

https://doi.org/10.15215/remix/9781998944088.01

```
>>> students = ['John', 'Mary', 'Terry', 'Smith',
'Chris']
>>> students[1:3]
['Mary', 'Terry']
```

L[START:END:STEP]

To get a slice/sublist of l, including members from a start position to right before an end position with set step.

```
>>> l6 = ['Jon', 'John', 'Jonathan', 'Jim', 'James']
>>> l6[:5:3]   # start from 0 till 5 with step set as 3.
['Jon', 'Jim']
```

L[N] = E

To replace element at n with e.

```
>>> print(students)
['John', 'Mary', 'Terry', 'Smith', 'Chris']
>>> students[2] = 'Cindy'
>>> print(students)
['John', 'Mary', 'Cindy', 'Smith', 'Chris']
```

L1 + L2

To concatenate list l2 to l1, but without changing l1. As such, if you want to keep the result of concatenation, you will need to assign the result to a new variable.

```
>>> teachers = ['Jeffery', 'Clover', 'David']
>>> students + teachers
['John', 'Mary', 'Terry', 'Smith', 'Chris', 'Jeffery',
'Clover', 'David']
>>> teachers
['Jeffery', 'Clover', 'David']
>>> class_members = students + teachers
>>> class_members
['John', 'Mary', 'Terry', 'Smith', 'Chris', 'Jeffery',
'Clover', 'David']
```

L * N
N * L

To duplicate list l n times but without changing l.

https://doi.org/10.15215/remix/9781998944088.01

```
>>> students[1:3]
['Mary', 'Terry']
>>> students[1:3] * 2
['Mary', 'Terry', 'Mary', 'Terry']
>>> 2*students[1:3]
['Mary', 'Terry', 'Mary', 'Terry']
```

E IN L

To test if e is in list l. If l has compound data such as lists, tuples, or instances of a class, e is only part of a compound data or object and is not considered in the list.

```
>>> teachers
['David', 'Jeffery', 'Clover']
>>> 'Clover' in teachers
True
>>> l0 = [1, 2, 3, [4, 5], 6]   # 4 and 5 are members of
a sublist of l0
>>> 5 in l0   # so that 5 is not considered as part of
list l0
False
```

LEN(L)

To get the number of elements in the list l.

```
>>> students
['John', 'Mary', 'Terry', 'Smith', 'Chris']
>>> len(students)
5
```

PRINT(L)

To print list l. Note that the list will be recursively printed, but complex objects such as instances of a user-defined class may not be printed the way you expected unless you have defined the __str__() method for the class.

```
>>> print(teachers)
['Jeffery', 'Clover', 'David']
```

In addition, there are also built-in methods for list objects, as detailed below.

https://doi.org/10.15215/remix/9781998944088.01

L.APPEND(E)

To append element e to list l.

```
>>> l = ['T', 'h']
>>> l.append('e')
>>> l
['T', 'h', 'e']
```

L.CLEAR()

To remove all items from list l.

```
>>> l1 = list("test")
>>> l1
['t', 'e', 's', 't']
>>> l1.clear()
>>> l1   # l1 became an empty list
[]
```

L.COPY()

To return a shallow copy of list l—that is, it only copies simple objects of the list such as numbers and strings; for compound data, it does not copy the actual objects but only makes references to the objects.

```
>>> l7 = l6.copy()   # from above we know that items in
l6 are all simple strings
>>> l7
['Jon', 'John', 'Jonathan', 'Jim', 'James']
>>> l7[3] = 'Joe'   # change the value of l7[3]
>>> l7   # it shows l7 has been changed
['Jon', 'John', 'Jonathan', 'Joe', 'James']
>>> l6   # it shows l6 remains the same
['Jon', 'John', 'Jonathan', 'Jim', 'James']
```

Now suppose we have

```
>>> l8 = [[13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23,
24, 25], ['Jon', 'John', 'Jonathan', 'Joe', 'James'],
100]
>>> l9 = l8.copy()
>>> l9   # l9 has the same items as l8
```

https://doi.org/10.15215/remix/9781998944088.01

```
[[13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25],
['Jon', 'John', 'Jonathan', 'Joe', 'James'], 100]
>>> l9[0][0] = 1000   # make change to the internal value
of list l9[0], that is, l9[0][0] to 1000
>>> l9   # l9 has been changed
[[1000, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25],
['Jon', 'John', 'Jonathan', 'Joe', 'James'], 100]
>>> l8   # l8 has been changed as well
[[1000, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25],
['Jon', 'John', 'Jonathan', 'Joe', 'James'], 100]
```

As can be seen, if you make changes to a piece of compound data (a list as the first item of l9 copied from l8), the change also occurs in the original list, and vice versa.

L.INDEX(E, START = 0, STOP = 9223372036854775807)

To return the first index of element e, from a start position till a stop position. The default range is from 0 to 9223372036854775807.

```
>>> l6.index('Jim')
3
```

L.POP()

To remove and return an item from the end of list l.

```
>>> l.pop()
'e'
```

L.POP(2)

To remove and return an item from the middle of list l. When there are an even number of elements in the list, there will be two elements in the middle, but only the first one pops out.

```
>>> l = [1, 3, 2, 6, 5, 7]
>>> l.pop(2)
2
>>> l
[1, 3, 6, 5, 7]
>>> l.pop(2)
6
```

https://doi.org/10.15215/remix/9781998944088.01

L.REVERSE()

To reverse the list.

```
>>> l.reverse()
>>> l
[7, 5, 3, 1]
```

L.SORT()

To sort the list in ascending order by default. To sort in descending order, use l.sort(reverse = True).

```
>>> l.sort()
>>> l
[1, 3, 5, 7]
```

L.EXTEND(L0)

To extend list l by appending list l0 to the end of list l. It is different from l + l0 but it is same as l += l0.

```
>>> l = list(range(5))
>>> l
[0, 1, 2, 3, 4]
>>> l0 = list(range(6, 11))
>>> l0
[5, 6, 7, 8, 9, 10]
>>> l.extend(l0)
>>> l
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

L.INSERT(I, E)

To insert e before index i of existing list l.

```
>>> l = list(range(10))
>>> l
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> l.insert(5, 13)
>>> l
[0, 1, 2, 3, 4, 13, 5, 6, 7, 8, 9]
```

https://doi.org/10.15215/remix/9781998944088.01

L.REMOVE(E)

To remove first occurrence of e in the list.

```
>>> l
[0, 1, 2, 3, 4, 13, 5, 6, 7, 8, 9]
>>> l.remove(5)
>>> l   # 5 has been removed from l
[0, 1, 2, 3, 4, 13, 6, 7, 8, 9]
```

L.COUNT(E)

To search the list and return the number of occurrences of e.

```
>>> l
[0, 1, 2, 3, 4, 13, 6, 7, 8, 9]
>>> l.count(6)
1
```

As you can see, elements in lists can be changed or mutated. You can insert, delete, replace, expand, and reorder all the elements in a list.

Lists are a very important data model in programming and problem solving. First, lists can be used as collections of data. Each member of the collection can be as simple as a number or a string and as complex as another list or any other compound data type, or even an object. Many functions and methods, as discussed above, have been made available for accessing and manipulating lists and their members.

Suppose we want to develop a management system for a company, for example. Within the system, we need to represent information on its employees. We can use a list containing the name, birthdate, department, start date at the company, and level of employment to represent information on each employee, then use another list to represent a collection of employees. This is illustrated as follows:

```
# this defines an embedded list or two-dimensional array
employees = [['Kevin Smith', 19560323, 'Sale', 20100621, 3],
['Paul Davina', 19860323, 'HR', 20120621, 5],
['Jim Carri', 1969323, 'Design', 20120625, 2],
['John Wong', 19580323, 'Customer Service', 20110323, 3],
['Keri Lam', 19760323, 'Sale', 20130522, 5]]
```

Moreover, lists can be used to represent trees, which is an important data structure in programming and problem solving.

https://doi.org/10.15215/remix/9781998944088.01

5.3 Tuples

Unlike a list, a tuple is an immutable object, which means that once created, the internal structure of a tuple cannot be changed. Hence, most methods you have seen for lists are not available for tuples, except for the following two.

T.COUNT(E)

To count and return the number of occurrences of a specified value in a tuple.

```
>>> t = (3, 6, 5, 7, 5, 9)
>>> t.count(5)
2
```

T.INDEX(E, START = 0, STOP = 9223372036854775807)

To search the tuple for a specified value e and return the index of the first occurrence of the value. Remember that just like a list, a tuple can have duplicate values as well.

```
>>> t.index(6)
1
>>> t.index(7)
3
>>> t0 = tuple("same as list, tuple")
>>> t0
('s', 'a', 'm', 'e', ' ', 'a', 's', ' ', 'l', 'i', 's',
't', ',', ' ', 't', 'u', 'p', 'l', 'e')
>>> t0.index('l')   # it only returns the index of the
first l
8
>>> t.index('l', 9)   # to get the index of the next
occurrence
17
```

As well, compared to list, fewer number of operators and built-in functions can be used on tuples, as shown below.

TUPLE(ITERABLE)

To construct a tuple from an iterable such as another sequence or a call to range(), a built-in function.

https://doi.org/10.15215/remix/9781998944088.01

```
>>> l1 = [1, 2, 3]
>>> t0 = tuple(l1)
>>> t0
(1, 2, 3)
```

This would be the same as the following:

```
>>> t0 = (1, 2, 3)

>>> t1 = tuple('tuple')
>>> t1
('t', 'u', 'p', 'l', 'e')

>>> tuple(range(7))
(0, 1, 2, 3, 4, 5, 6)
```

T[N]

To get nth element of a tuple.

```
>>> teachers = ('Jeffery', 'Clover', 'David')
>>> teachers[2]
'David'
```

Please note that because a tuple is an immutable sequence, making changes to its members will generate an error, as shown below:

```
>>> teachers[1] = 'Chris'
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item
assignment
```

T[I:J]

To get a slice of tuple t including elements from point i to the one right before point j.

```
>>> teachers[0:2]
('Jeffery', 'Clover')
>>> print(teachers)
('Jeffery', 'Clover', 'David')
```

https://doi.org/10.15215/remix/9781998944088.01

T1 + T2

To concatenate tuple t2 to t1.

```
>>> students = tuple(students)
>>> print(students)
('John', 'Mary', 'Terry', 'Smith', 'Chris')
>>> students + teachers
('John', 'Mary', 'Terry', 'Smith', 'Chris', 'Jeffery',
'Clover', 'David')
```

T * N

To duplicate tuple t n times.

```
>>> teachers * 2
('Jeffery', 'Clover', 'David', 'Jeffery', 'Clover',
'David')
```

E IN T

To test if e is an element of tuple t.

```
>>> teachers
('David', 'Jeffery', 'Clover')
>>> 'David' in teachers
True
```

LEN(T)

To get the number of elements in the tuple t.

```
>>> len(teachers * 2)
6
```

PRINT(T)

To print tuple t. Again, ***print*** may print the tuple recursively, but the expected result can only be achieved if __str__() has been defined for every object at all levels.

```
>>> print(students)
('John', 'Mary', 'Terry', 'Smith', 'Chris')
```

Again, although we can extend the tuple by concatenating and duplicating, we cannot make any change to the existing element of a tuple as we did to

https://doi.org/10.15215/remix/9781998944088.01

lists, because tuples are immutable. As a result, the tuple is not a suitable data structure for representing the group of employees in the example presented at the end of the previous section because employees may come and go.

5.4 Sets

As in mathematics, a set is a collection of unindexed and unordered elements. For sets, Python has very few operators and built-in functions that we can use.

SET(S)

To construct a set from s, which can be a list, tuple, or string.

```
>>> students = ['Cindy', 'Smith', 'John', 'Chris',
'Mary']
>>> students = set(students)
>>> students
{'Cindy', 'Smith', 'John', 'Chris', 'Mary'}
>>> numbers = set(range(10))
>>> numbers
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
```

E IN S

To test if e is a member of set s.

```
>>> students
{'Cindy', 'Smith', 'John', 'Chris', 'Mary'}
>>> 'Chris' in students
True
```

LEN(S)

To get the total number of elements in the set.

```
>>> numbers
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
>>> len(numbers)
10
```

However, there are good number of methods defined for sets.

https://doi.org/10.15215/remix/9781998944088.01

S.ADD(M)

To add an element to the set s.

```
>>> s = set([3])
>>> s
{3}
>>> s.add(5)
>>> s
{3, 5}
```

S.CLEAR()

To remove all the elements from the set s.

```
>>> s.clear()
>>> s
set()
```

S.COPY()

To make and return a copy of the set s.

```
>>> s
{3, 5}
>>> s1 = s.copy()
>>> s1
{3, 5}
```

S.DIFFERENCE(S1,…)

To make and return a set containing only members of s that other sets in the arguments don't have—that is, the difference between two or more sets.

```
>>> s1
{3, 5}
>>> s2 = {5, 7}
>>> s1.difference(s2)
{3}
>>> s3={3,7}
>>> s1.difference(s2,s3)   # returns an empty set
set()
```

https://doi.org/10.15215/remix/9781998944088.01

S.DIFFERENCE_UPDATE(*SX)

To remove the items in set s that are also included in another, specified set.

```
>>> s1 = {2 * i for i in range(15)}
>>> s2 = {3 * i for i in range(15)}
>>> s1
{0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28}
>>> s2
{0, 33, 3, 36, 6, 39, 9, 42, 12, 15, 18, 21, 24, 27, 30}
>>> s1.difference_update(s2)
>>> s1
{2, 4, 8, 10, 14, 16, 20, 22, 26, 28}
>>> s2
{0, 33, 3, 36, 6, 39, 9, 42, 12, 15, 18, 21, 24, 27, 30}
```

S.DISCARD(M)

To remove the specified item.

```
>>> s2
{0, 33, 3, 36, 6, 39, 9, 42, 12, 15, 18, 21, 24, 27, 30}
>>> s2.discard(18)
>>> s2
{0, 33, 3, 36, 6, 39, 9, 42, 12, 15, 21, 24, 27, 30}
```

S.INTERSECTION(*SX)

To return a set that is the intersection of two other sets.

```
>>> s1 = {2 * i for i in range(15)}
>>> s2 = {3 * i for i in range(15)}
>>> s1
{0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28}
>>> s2
{0, 33, 3, 36, 6, 39, 9, 42, 12, 15, 18, 21, 24, 27, 30}
>>> s1.intersection(s2)
{0, 6, 12, 18, 24}
```

S.INTERSECTION_UPDATE(*SX)

To remove the items in this set that are not present in another, specified set or sets.

https://doi.org/10.15215/remix/9781998944088.01

```
>>> s1
{0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28}
>>> s2
{0, 33, 3, 36, 6, 39, 9, 42, 12, 15, 18, 21, 24, 27, 30}
>>> s1.intersection_update(s2)
>>> s1
{0, 6, 12, 18, 24}
```

S.ISDISJOINT(SX)

To check and return whether two sets have an intersection (common member) or not.

```
>>> s1
{0, 6, 12, 18, 24}
>>> s2
{0, 33, 3, 36, 6, 39, 9, 42, 12, 15, 18, 21, 24, 27, 30}
>>> s1.isdisjoint(s2)
False
```

S.ISSUBSET(SX)

To check and return whether another set contains this set or not.

```
>>> s1 = {2 * i for i in range(15)}
>>> s2 = {3 * i for i in range(15)}
>>> s1
{0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28}
>>> s2
{0, 33, 3, 36, 6, 39, 9, 42, 12, 15, 18, 21, 24, 27, 30}
>>> s1.issubset(s2)
False
```

S.ISSUPERSET(SX)

To check and return whether this set contains another set or not.

```
>>> s1.issuperset(s2)
False
```

S.POP()

To remove an element from the set.

https://doi.org/10.15215/remix/9781998944088.01

```
>>> s2
{0, 33, 3, 36, 6, 39, 9, 42, 12, 15, 18, 21, 24, 27, 30}
>>> s2.pop()
0
>>> s2
{33, 3, 36, 6, 39, 9, 42, 12, 15, 18, 21, 24, 27, 30}
```

S.REMOVE(M)

To remove the specified element.

```
>>> s2
{33, 3, 36, 6, 39, 9, 42, 12, 15, 18, 21, 24, 27, 30}
>>> s2.remove(18)
>>> s2
{33, 3, 36, 6, 39, 9, 42, 12, 15, 21, 24, 27, 30}
```

S.SYMMETRIC_DIFFERENCE(SX)

To construct and return a set with elements in either set s or another set but not both. These are called set symmetric differences ("I have you do not; you have I do not").

```
>>> s1 = {2 * i for i in range(15)}
>>> s2 = {3 * i for i in range(15)}
>>> s1
{0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28}
>>> s2
{0, 33, 3, 36, 6, 39, 9, 42, 12, 15, 18, 21, 24, 27, 30}
>>> s1.symmetric_difference(s2)
{2, 3, 4, 8, 9, 10, 14, 15, 16, 20, 21, 22, 26, 27, 28,
30, 33, 36, 39, 42}
```

S.SYMMETRIC_DIFFERENCE_UPDATE(SX)

To insert the symmetric differences from this set and another.

```
>>> s1.symmetric_difference_update(s2)
>>> s1
{2, 3, 4, 8, 9, 10, 14, 15, 16, 20, 21, 22, 26, 27, 28,
30, 33, 36, 39, 42}
```

https://doi.org/10.15215/remix/9781998944088.01

S.UNION(SX)

To return a set containing the union of sets.

```
>>> s2 = {3 * i for i in range(5)}
>>> s1 = {2 * i for i in range(5)}
>>> s1
{0, 2, 4, 6, 8}
>>> s2
{0, 3, 6, 9, 12}
>>> s1.union(s2)
{0, 2, 3, 4, 6, 8, 9, 12}
```

S.UPDATE(SX)

To update the set by adding members from other sets.

```
>>> s1
{0, 2, 4, 6, 8}
>>> s2
{0, 3, 6, 9, 12}
>>> s1.update(s2)
>>> s1
{0, 2, 3, 4, 6, 8, 9, 12}
```

5.5 Dictionaries

A dictionary is a collection of key and value pairs enclosed in curly brackets. As with a set, the dictionary is also immutable. There are very few operators and built-in functions that can be used on dictionaries, as shown below.

DICT(**KWARG)

To construct a dictionary from a series of keyword arguments.

```
>>> dt = dict(one = 1, two = 2, three = 3)
>>> dt
{'one': 1, 'two': 2, 'three': 3}
```

DICT(MAPPING, **KWARG)

To construct a dictionary from mapping. If keyword arguments are present, they will be added to the dictionary constructed from the mapping.

https://doi.org/10.15215/remix/9781998944088.01

```
>>> d1 = dict(zip(['one', 'two', 'three'], [1, 2, 3]))
>>> d2 = dict(zip([1, 2, 3], ['one', 'two', 'three']))
>>> d1
{'one': 1, 'two': 2, 'three': 3}
>>> d2
{1:'one', 2:'two', 3:'three'}
```

DICT(ITERABLE, **KWARG)

To construct a dictionary from an iterable. If keyword arguments are present, they will be added to the dictionary constructed from the mapping.

```
>>> d3 = dict([('two', 2), ('one', 1), ('three', 3)])
>>> d3
{'two': 2, 'one': 1, 'three': 3}
```

LIST(DT)

To return a list of all the keys used in the dictionary dt.

```
>>> d3
{'two': 2, 'one': 1, 'three': 3}
>>> list(d3)
['two', 'one', 'three']
```

DT[K]

To get the value of key k from dictionary dt.

```
>>> dt = {1:'One', 2:'Two', 3:'Three'}
>>> dt[1]
'One'
```

DT[K] = V

To set d[key] to value V.

```
>>> d3
{'two': 2, 'one': 1, 'three': 3}
>>> d3['two']
2
>>> d3['two'] = bin(2)
>>> d3['two']
'0b10'
```

https://doi.org/10.15215/remix/9781998944088.01

DEL DT[K]

To remove dt[key] from dt.

```
>>> d3
{'two':'0b10', 'one': 1, 'three': 3}
>>> del d3['two']
>>> d3
{'one': 1, 'three': 3}
```

K IN DT

To test if dt has a key k.

```
>>> d3
{'one': 1, 'three': 3}
>>> 'two' in d3
False
```

K NOT IN DT

Same as *not k in dt*, or *k not in dt*.

```
>>> 'two' not in d3
True
>>> not 'two' in d3
True
```

ITER(DT)

To return an iterator over the keys of dt. Same as iter(dt.keys()).

```
>>> iter(d3)
<dict_keyiterator object at 0x00000198FE0EFEF8>
>>> list(iter(d3))
['one', 'three']
```

LEN(DT)

To get the total number of elements in the dictionary.

```
>>> dt
{1:'One', 2:'Two', 3:'Three'}
>>> len(dt)
3
```

https://doi.org/10.15215/remix/9781998944088.01

REVERSED(DT)

To return a reverse iterator over the keys of the dictionary. Same effect as reversed(dt.keys()). This is new in Python 3.8.

```
>>> dt = {1:'One', 2:'Two', 3:'Three'}   # keys: 1, 2, 3
>>> rk = reversed(dt)   # reversed iterator over the keys
in rk
>>> for k in rk:
print(k)
…
3
2
1
```

Note that in the output above, the keys in rk are 3, 2, 1.

The following built-in methods of the dictionary class can be used to manipulate dictionaries.

D.CLEAR()

To remove all the elements from the dictionary.

```
>>> dt = {1:'One', 2:'Two', 3:'Three'}
>>> dt
{1:'One', 2:'Two', 3:'Three'}
>>> dt.clear()
>>> dt
```

D.COPY()

To make and return a copy of the dictionary.

```
>>> dt = {1:'One', 2:'Two', 3:'Three'}
>>> dx = dt.copy()
>>> dx
{1:'One', 2:'Two', 3:'Three'}
```

DICT.FROMKEYS()

To make a dictionary from a list of keys.

https://doi.org/10.15215/remix/9781998944088.01

```
>>> keys = ['Edmonton','Calgary','Toronto']
>>> weather = dict.fromkeys(keys, 'Sunny')
>>> print(weather)
{'Edmonton': 'Sunny', 'Calgary': 'Sunny', 'Toronto': 'Sunny'}
```

D.GET(K)

To return the value of the specified key.

```
>>> d3 =dict([('two', 2), ('one', 1), ('three', 3)])
>>> d3.get('two')
2
```

D.ITEMS()

To return a list containing a tuple for each key-value pair.

```
>>> d3.items()
dict_items([('two', 2), ('one', 1), ('three', 3)])
```

D.KEYS()

To return a list containing the dictionary's keys.

```
>>> d3.keys()
dict_keys(['two', 'one', 'three'])
```

D.VALUES()

To return a list of all the values in the dictionary.

```
>>> d3.values()
dict_values([2, 1, 3])
```

D.POP(K)

To remove the element with the specified key. Note that the removed item will no longer exist in the dictionary.

```
>>> d3
{'two': 2, 'one': 1, 'three': 3}
>>> d3.pop('two')
2
>>> d3
{'one': 1, 'three': 3}
```

https://doi.org/10.15215/remix/9781998944088.01

D.POPITEM()

To remove an item from the end of the dictionary, as a key and value pair.

```
>>> d3
{'two': 2, 'one': 1, 'three': 3}
>>> d3.popitem()
('three', 3)
```

D.SETDEFAULT(KEY, VALUE)

To insert a key-value pair into the dictionary if the key is not in the dictionary; return the value of the key if the key exists in the dictionary.

```
>>> d3
{'two': 2, 'one': 1}
>>> d3.setdefault('three', 3)
3
>>> d3.setdefault('two', 'II')
2
>>> d3
{'two': 2, 'one': 1, 'three': 3}
```

D.UPDATE(DX)

To update the dictionary with the specified key-value pairs in dx.

```
>>> d3
{'two': 2, 'one': 1, 'three': 3}
>>> d2
{1:'one', 2:'two', 3:'three'}
>>> d3.update(d2)
>>> d3
{'two': 2, 'one': 1, 'three': 3, 1:'one', 2:'two',
3:'three'}
```

5.6 List, Set, and Dictionary Comprehension

Lists, sets, and dictionaries are important data models for programmers to structure and organize data with. Before using lists, tuples, sets, and dictionaries, it is important to create them in a nice way. List, set, and dictionary comprehension is provided by Python to construct lists, sets, and dictionaries

https://doi.org/10.15215/remix/9781998944088.01

in a concise but efficient language. The essential idea for list, set, and dictionary comprehension is the use of a ***for*** loop with or without conditions.

List Comprehension

The following is an example that constructs a list of odd numbers from 1 to 100:

```
In [ ]:  l0 = [i * 2 + 1 for i in range(50)]
         print(l0)
Out [ ]: [1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29, 31, 33, 35, 37, 39, 41, 43,
          45, 47, 49, 51, 53, 55, 57, 59, 61, 63, 65, 67, 69, 71, 73, 75, 77, 79, 81, 83,
          85, 87, 89, 91, 93, 95, 97, 99]
```

In the example, the expression before ***for*** represents the items of the list; the ***for*** loop will run through the item expression in each iteration. This list can also be generated using the ***for*** loop with an ***if*** clause, as shown below:

```
In [ ]:  l1 = [i for i in range(100) if i % 2 != 0]
         print(l1)
Out [ ]: [1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29, 31, 33, 35, 37, 39, 41, 43,
          45, 47, 49, 51, 53, 55, 57, 59, 61, 63, 65, 67, 69, 71, 73, 75, 77, 79, 81, 83,
          85, 87, 89, 91, 93, 95, 97, 99]
```

In the example above, the list item expression will be evaluated in every iteration of the ***for*** loop only if the condition of the ***if*** clause is met. In fact, we can put any condition on the iteration variable i. For example, assume we have a Boolean function isPrime(N) that can test if number N is prime or not; then the following statement will produce a list of prime numbers in the given range:

```
primes = [i for i in range(1000) if isPrime(i)]
```

Please note that the list item expression before ***for*** can be anything whose value is a legitimate list item, as shown in the example below:

```
In [ ]:  detailed = [f"{i} is odd" if i % 2 !=0 else f"{i} is
           even" for i in range(10)]
         print(detailed)
Out [ ]: ['0 is even', '1 is odd', '2 is even', '3 is odd', '4 is even', '5 is odd', '6 is even', '7
          is odd', '8 is even', '9 is odd']
```

For item expressions involving two variables or more, nested ***for*** statements can be used. For example, the following statement will generate a list of combinations of some years and months:

https://doi.org/10.15215/remix/9781998944088.01

In [ ]:
```
years = ['2015', '2016', '2017', ' 2018', '2019']
combo = [year + str(month).rjust(2, '0') for year in
  years for month in range(1, 13)]
print(combo)
```

Out [ ]:
```
['201501', '201502', '201503', '201504', '201505', '201506', '201507', '201508',
  '201509', '201510', '201511', '201512', '201601', '201602', '201603', '201604',
  '201605', '201606', '201607', '201608', '201609', '201610', '201611', '201612',
  '201701', '201702', '201703', '201704', '201705', '201706', '201707', '201708',
  '201709', '201710', '201711', '201712', ' 201801', ' 201802', ' 201803', '
  201804', ' 201805', ' 201806', ' 201807', ' 201808', ' 201809', ' 201810', '
  201811', ' 201812', '201901', '201902', '201903', '201904', '201905', '201906',
  '201907', '201908', '201909', '201910', '201911', '201912']
```

Set Comprehension

A set is a collection of unique unordered items. With that in mind, set comprehension is similar to list comprehension, except that the items are enclosed in curly brackets, as shown in the example below:

In [ ]:
```
asc = {chr(c) for c in range(ord('A'), ord('Z') + 1)}
print(asc)
```

Out [ ]:
```
{'K', 'M', 'G', 'T', 'C', 'O', 'L', 'D', 'S', 'I', 'B', 'N', 'A', 'F', 'W', 'H', 'P', 'X', 'J', 'Z', 'E', 'R', 'U',
  'Y', 'Q', 'V'}
```

What would happen if items generated from the iteration were duplicated? No worries! The implementation of set comprehension can take care of that. If we want to find out all the unique words contained in a web document, we can simply use set comprehension to get them, as shown below:

In [ ]:
```
import requests   # import requests module to handle
  requests for web resources
content = requests.get("https://scis.athabascau.ca/").
  text
separators = [',', '.', '"', "'", '>', '<', '--', '!',
  '|', ']', '[', '?', ';', '/']   # separators used to
  separate words
separators += [')', '(', '$', '&', ':', '}', '{']
operators = ['=', '+']
for sp in separators:
   content = content.replace(sp, ' ')   # replace each
  of the separators with a space
for op in operators:
   content = content.replace(op, f' {op} ')   # add a
  space before and after each operator
unique_words = {w for w in content.split()}
print(len(unique_words), unique_words)
```

Out [ ]:
```
969
```

https://doi.org/10.15215/remix/9781998944088.01

The example above took a web document at https://scis.athabascau.ca, pulled all the unique words used in the document into a set, and printed the number of unique words used, which is 969.

As you can see, we could get the unique words in a document very easily by using set comprehension. How would we find out the ratio between the number of unique words and the total number of words used?

Dictionary Comprehension

Dictionary comprehension is very similar to set comprehension, except that we need to add a key and colon before each item to make a dictionary item, as shown in the following:

```
In [ ]:  months = ['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun',
           'Jul', 'Aug', 'Sep', 'Oct', 'Nov', 'Dec']
         m_dict = {i + 1: months[i] for i in range(12)}
         print(m_dict)

Out [ ]: {1: 'Jan', 2: 'Feb', 3: 'Mar', 4: 'Apr', 5: 'May', 6: 'Jun', 7: 'Jul', 8: 'Aug', 9: 'Sep', 10:
           'Oct', 11: 'Nov', 12: 'Dec'}
```

This can also be written in nested ***for*** clauses, as shown below:

```
In [ ]:  months = ['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun',
           'Jul', 'Aug', 'Sep', 'Oct', 'Nov', 'Dec']
         m_dict = {i + 1: m for i in range(12) for m in
           months}
         print(m_dict)

Out [ ]: {1: 'Dec', 2: 'Dec', 3: 'Dec', 4: 'Dec', 5: 'Dec', 6: 'Dec', 7: 'Dec', 8: 'Dec', 9: 'Dec',
           10: 'Dec', 11: 'Dec', 12: 'Dec'}
```

5.7 Text Files

For people and smart beings in general, part of intelligence is being able to remember things. Memory is an important part of that.

Computers have two types of memory. The first is RAM, which is volatile, expensive, and of relatively lower capacity but provides high-speed access. The variables we have been talking about so far are using RAM to hold data and running programs that are also inside RAM. If the computer is turned off, both data and programs will disappear from RAM. RAM is also called internal memory.

https://doi.org/10.15215/remix/9781998944088.01

The second type of memory that modern computers have is persistent memory, such as a hard drive, flash memory card, or solid-state hard disk. This type of memory is also called external memory. For this type of memory to be useful, it must be part of a file system managed by an OS such as Windows, iOS, or Linux. Within a file system, files are used for saving data and programs to external memory, and the files are organized into hierarchical directories or folders. In a file system, a file can be located through a path from the root to the file within a tree-like structure.

Opening and Closing a File

To use a file, the first step is to open the file using the built-in function ***open***. The following will open a file for writing:

```
f = open("./mypoem.txt", 'w')   # open file mypoem.txt in
the current working directory for writing
```

The statement above opened the file mypoem.txt in the current working directory and returned a stream, which was then assigned to variable f, often called a file handle.

The general syntax of the ***open*** statement is as follows:

```
open(file, mode = 'r', buffering = -1, encoding = None,
errors = None, newline = None, closefd = True, opener =
None)
```

The statement opens a file and returns a stream or file object; it will raise OSError upon failure. In the statement, *file* is a text or byte string referring to the file to be opened. It may include the path to the actual file if the file is not in the current working directory.

Apart from the first argument for the name of the file to be opened, all other arguments have default values, which means that these arguments are optional. If no value is supplied, the default value will be used.

The second argument is called mode. It is optional with a default value r, which means "open a text file for reading." This argument is used to tell the program what to do with the file after opening. Because reading from a text file is not always what you want it to do with a file, the argument is not optional. All available values for the mode argument are shown in Table 5-7.

https://doi.org/10.15215/remix/9781998944088.01

Table 5-7: List of mode arguments for the open function

Mode argument	Access or accesses
r	Open for reading only; file must already exist.
r+	Open for both reading and writing; file must already exist.
w	Open for writing only; file may or may not exist. If not, a new file will be created and ready for writing; if file already exists, content will be overwritten.
w+	Open for writing and reading; file may or may not exist.
a	Open for appending, same as w, but will not overwrite the existing content.
a+	Open for both appending and reading.
x	Create a new file for writing. File Exists Error will occur if the file already exists. This would help prevent accidentally overwriting an existing file.

The file to be opened or created can be either a text file, referred to as t, or a binary file containing raw bytes, referred as b. To explicitly specify whether the file is a text or binary file, t or b can be used in combination with each of the values in Table 5-7. The default file type is text, so you do not have to use t if the file is a text file. So in the example above, the statement is equivalent to the following, in which t is used to explicitly indicate that it is a text file:

```
f = open("./mypoem.txt", 'wt')   # open text file mypoem.
txt in the current working directory
```

In both examples, we assigned the file stream returned from the ***open*** statement to a variable f, which is called the file handle. After the file is opened, all operations on the file—such as read, write, append, or even close—must be appended to the file handle. Every file must be closed using the f.close() method after open and use unless the file is opened within a ***with*** statement, in which case a context manager will take over access to the file and close it when the job is done, as in the sample code below:

```
In [ ]:  with open("./mypoem.txt", 'r') as f:   # f is still a
           file handle
            for ln in f:   # a file stream referred by f is an
           iterator
              print(ln)
Out [ ]: Yet it was plain she struggled, and that salt
```

https://doi.org/10.15215/remix/9781998944088.01

When not using the ***with*** statement, you will need to use the following instead:

```
In [ ]:  f = open("./mypoem.txt", 'r')   # f is still a file
          handle
         for ln in f:   # a file stream referred by f is an
          iterator
           print(ln)
         f.close()
```

```
Out [ ]: Yet it was plain she struggled, and that salt
```

The third argument is called buffering. It takes an optional integer used to specify the buffering policy of the file operation. Passing 0 turns off buffering, but it is only allowed for binary files. Passing 1 selects line buffering, which is only allowed for text files. Passing an integer greater than 1 specifies the actual size of a fixed-size chunk buffer. When no buffering argument is provided, the default value −1 is used, which means that if the file is a binary file, the file will be buffered in a fixed-size chunk; if the file is a text file, line buffering policy is used, which means that the data will be flush to the actual file (on a disk such as a hard drive) from the buffer (a portion of internal memory to buffer the data) after each line was written.

The fourth argument is encoding, which specifies how the data in the file are encoded. This argument only makes sense for text files. The default value is platform dependent. If you believe that the encoding of data on a file is not the default, you can specify whatever encoding in which the data are encoded. However, the encoding must be supported by Python. In most cases, UTF-8 is the default encoding.

The next optional argument is errors, which takes a string if provided. The string specifies how encoding errors should be handled. Again, this argument only makes sense if the file is a text file. The same is true for the optional newline argument, which controls how universal newlines work in a text file. The optional newline argument can take the following:

- None
- ,
- \n
- \r
- \rn

Once a file is opened, a list of methods can be used to operate on the file object, as detailed below.

https://doi.org/10.15215/remix/9781998944088.01

Write or Append to a File

New data can be added to a file in two different manners. The first one is to overwrite everything already in the file and place the new data at the beginning of the file, and the second one is to keep the data already in the file and append the new data to the end of the existing data. The write methods are the same for both, but the file must be opened with a different mode depending on the operation. That is, mode w or x must be used to open a file for writing the data from the beginning of the file, and mode a must be used to append new data, as you will see shortly in the examples.

There are two methods for writing to a file. The first one is f.write(string), which writes string to the file referred by f. The second method is f.writelines(sequence), in which the sequence is any iterable object, such as a list or tuple, but often a list of strings. The following is an example of opening or creating a file (if the file does not exist yet) for writing data from the beginning of the file using the write(string) method:

```
>>> f = open("./mypoem.txt", "w")   # open file in the
current working directory for writing
>>> f.write("\nYou may write me down in history")   # add
\n to write on a new line
>>> f.flush()   # to flush the data out to the actual file
```

The resulting file will read,

```
You may write me down in history
```

If you could write only this one line of your poem and had to close the file and shut down the computer, you would be more likely to continue from where you had stopped the next time you came back to the poem. So you need to append the new lines to the file. This is done by opening the file in a mode, as shown below:

```
>>> f = open("./mypoem.txt", "a")   # open file in the
current working directory for writing
>>> f.write("\nWith your bitter, twisted lies")   # add
\n to write on a new line
>>> f.flush()   # to flush the data out to the actual
file
```

The file is extended with one more line of poem:

https://doi.org/10.15215/remix/9781998944088.01

```
You may write me down in history
With your bitter, twisted lies
```

Note that the write method will only write a string to the file. As such, anything that is not a string must be converted into a string before being written to the file, as shown in the following example:

```
>>> f.write(f'\n{3.1415926}')
```

Also, because of buffering, the data you write to a file will not immediately show up in the actual file until you close the file or use the flush() method to flush the data in the buffer out to the file, as shown below:

```
>>> f.flush()
```

The write(string) method can only write one string to a file each time. To write multiple strings to a file, the writelines(sequence) method is used. However, keep in mind that writelines() does not automatically write one string on each line. You will still need to add \n at the beginning of the string if you want it to be on the next line or at the end of the string if you don't want anything behind the string on the same line.

Recall the example of printing a 9 × 9 multiplication table. Now we can write the table to a text file so that you can print it out whenever you want. This is shown in the two examples below:

```
"""This first code sample is using the write method."""

f = open('./my9x9table.txt', 'w')
for i in range(1, 10):
  for j in range(1, i + 1):
    f.write('{:1d} x {:1d} = {:2d} '.format(j, i, i * j))
  f.write('\n')
f.close()
```

The output of the program is in the file my9x9table.txt:

```
1 x 1 = 1
1 x 2 = 2     2 x 2 = 4
1 x 3 = 3     2 x 3 = 6     3 x 3 = 9
1 x 4 = 4     2 x 4 = 8     3 x 4 = 12     4 x 4 = 16
1 x 5 = 5     2 x 5 = 10    3 x 5 = 15     4 x 5 = 20     5 x 5 = 25
```

https://doi.org/10.15215/remix/9781998944088.01

To use the writelines(sequence) method, we need to store the results in a list first; each item of the list will be printed on one line. The code is shown as follows:

```
"""This code sample is using the writelines method."""

table = []
for i in range(1, 10):
  newline = ''
  for j in range(1, i + 1):
    newline += '{:1d} x {:1d} = {:2d} '.format(j, i, i * j)
  newline += '\n'
  table.append(newline)

f = open('./my9x9table0.txt', 'w')
f.writelines(table)
f.close()
```

The result is the same as in the my9x9table.txt text shown above.

Reading from a File

To read from a file, three methods can be used. These methods are read([size]), readline([size]), and readlines([sizehint]).

Use the read([size]) method to read the entire file and return the entire contents as a single string or, if the optional size argument is given, to read the specified number of bytes and return the contents as a single string. The following example shows how the 9 × 9 multiplication table is read using the read([size]) method:

```
In [ ]:  f = open('./my9x9table0.txt', 'r')
         ln = f.read()
         print(ln)
         f.close()

Out [ ]: 1 x 1 = 1
         1 x 2 = 2 2 x 2 = 4
         1 x 3 = 3 2 x 3 = 6 3 x 3 = 9
         1 x 4 = 4 2 x 4 = 8 3 x 4 = 12 4 x 4 = 16
         1 x 5 = 5 2 x 5 = 10 3 x 5 = 15 4 x 5 = 20 5 x 5 = 25
         1 x 6 = 6 2 x 6 = 12 3 x 6 = 18 4 x 6 = 24 5 x 6 = 30 6 x 6 = 36
         1 x 7 = 7 2 x 7 = 14 3 x 7 = 21 4 x 7 = 28 5 x 7 = 35 6 x 7 = 42 7 x 7 = 49
         1 x 8 = 8 2 x 8 = 16 3 x 8 = 24 4 x 8 = 32 5 x 8 = 40 6 x 8 = 48 7 x 8 = 56 8 x
           8 = 64
         1 x 9 = 9 2 x 9 = 18 3 x 9 = 27 4 x 9 = 36 5 x 9 = 45 6 x 9 = 54 7 x 9 = 63 8 x
           9 = 72 9 x 9 = 81
```

https://doi.org/10.15215/remix/9781998944088.01

If size is given, only that number of bytes will be read, as shown in the next example:

```
In [ ]:  f = open('./my9x9table0.txt', 'r')
         ln = f.read(135)   # only read 135 bytes
         print(ln)
         f.close()
Out [ ]: 1 x 1 = 1
         1 x 2 = 2 2 x 2 = 4
         1 x 3 = 3 2 x 3 = 6 3 x 3 = 9
         1 x 4 = 4 2 x 4 = 8 3 x 4 = 12 4 x 4 = 16
         1
```

Because the given size is so small, only a small portion of the multiplication table has been read from the file.

Our next method for reading data from a file is readline([size]). This method will read and return one entire line from the file if the optional size argument is not provided or if the integer value is equal to or greater than the size of the line. If the provided size is smaller than the actual size of the line being read, then only part of that line, equal to the size in bytes, will be read and returned, as shown in the following example:

```
In [ ]:  f = open('./my9x9table0.txt', 'r')
         ln = f.readline(3)
         print(ln, end='')
         f.close()
Out [ ]: 1 x
```

Using this method to read all the lines of the 9 × 9 multiplication table in the file shown in the previous examples, we will need to put it in a loop and read line by line until the end of the file. In Python, however, there is no effective way to test if it has reached the end of the file. For this particular file, since we know there is no blank line before the end of the file, we will use an empty string to signify the end of the file. The revised code is shown below:

```
In [ ]:  f = open('./my9x9table0.txt', 'r')
         while True:
           ln = f.readline()
           print(ln, end='')
           if ln == '':   # test if it has reached the end of
          the table
             break
         f.close()
```

https://doi.org/10.15215/remix/9781998944088.01

```
Out [ ]:  1 x 1 = 1
          1 x 2 = 2 2 x 2 = 4
          1 x 3 = 3 2 x 3 = 6 3 x 3 = 9
          1 x 4 = 4 2 x 4 = 8 3 x 4 = 12 4 x 4 = 16
          1 x 5 = 5 2 x 5 = 10 3 x 5 = 15 4 x 5 = 20 5 x 5 = 25
          1 x 6 = 6 2 x 6 = 12 3 x 6 = 18 4 x 6 = 24 5 x 6 = 30 6 x 6 = 36
          1 x 7 = 7 2 x 7 = 14 3 x 7 = 21 4 x 7 = 28 5 x 7 = 35 6 x 7 = 42 7 x 7 = 49
          1 x 8 = 8 2 x 8 = 16 3 x 8 = 24 4 x 8 = 32 5 x 8 = 40 6 x 8 = 48 7 x 8 = 56 8 x
            8 = 64
          1 x 9 = 9 2 x 9 = 18 3 x 9 = 27 4 x 9 = 36 5 x 9 = 45 6 x 9 = 54 7 x 9 = 63 8 x
            9 = 72 9 x 9 = 81
```

The code above does not look so neat. In fact, since the text file is treated as an iterator in Python, with one item for each line, the above code can be simply written as follows:

```
f = open('./my9x9table0.txt', 'r')
for ln in f:
  print(ln, end = '')
f.close()
```

The output is the same as above.

Using a context manager with the code can be further simplified as follows:

```
with open('./my9x9table0.txt', 'r') as f:
for ln in f:
  print(ln, end='')   # keyword argument end is set empty
because ln already has newline in it
```

Considering the fact that a text file is an iterator, the built-in function next(iterator) can be used to iterate the file line by line. However, it would raise a StopIteration error if it reached the end of the file. The following example shows how to use next(iterator) to read and print the entire multiplication table:

```
In [ ]:  f = open('./my9x9table0.txt', 'r')
         try:
           while True:
             line = next(f)   # treat f as an iterator
             print(line, end='')
         except (StopIteration):
           f.close()   # if it reached the end of the file,
          close the file
```

https://doi.org/10.15215/remix/9781998944088.01

```
Out [ ]:  1 x 1 = 1
          1 x 2 = 2 2 x 2 = 4
          1 x 3 = 3 2 x 3 = 6 3 x 3 = 9
          1 x 4 = 4 2 x 4 = 8 3 x 4 = 12 4 x 4 = 16
          1 x 5 = 5 2 x 5 = 10 3 x 5 = 15 4 x 5 = 20 5 x 5 = 25
          1 x 6 = 6 2 x 6 = 12 3 x 6 = 18 4 x 6 = 24 5 x 6 = 30 6 x 6 = 36
          1 x 7 = 7 2 x 7 = 14 3 x 7 = 21 4 x 7 = 28 5 x 7 = 35 6 x 7 = 42 7 x 7 = 49
          1 x 8 = 8 2 x 8 = 16 3 x 8 = 24 4 x 8 = 32 5 x 8 = 40 6 x 8 = 48 7 x 8 = 56 8 x
            8 = 64
          1 x 9 = 9 2 x 9 = 18 3 x 9 = 27 4 x 9 = 36 5 x 9 = 45 6 x 9 = 54 7 x 9 = 63 8 x
            9 = 72 9 x 9 = 81
```

The third method for reading data from a file is readlines([sizehint]), where optional sizehint, if provided, should be an integer hinting at the amount of data to be read. Again, it is only available for text files. As the name implies, it reads multiple lines into a Python list until the end of the file, or as much as defined by sizehint, if the argument is provided. For example, if the total amount of data of the first n lines is less than sizehint, but the first n + 1 lines is greater than sizehint, then the method will read (n + 1) lines. So it will read whole lines rather than partial, in contrast to the readline([size]) method.

Sometimes, we might like to read from a particular portion of a file, just like we want to start reading a book from a specific page. How can we do that in Python?

Imagine there is a pointer indicating where the reading will start in a file. In Python, several methods can be used to adjust the pointer.

The first method is f.tell(), which determines where the pointer is in terms of how many bytes ahead it is, as shown in the following example:

```
In [ ]:   f = open('./my9x9table0.txt', 'r')
          while True:
            pt = f.tell()
            line = f.readline()
            print('{:3d}: {:s}'.format(pt, line), end = '')
            if line == '':   # test if it has reached the end of
           the table
              break

          f.close()
```

```
Out [ ]:  0:  1 x 1 = 1
          15:  1 x 2 = 2 2 x 2 = 4
          43:  1 x 3 = 3 2 x 3 = 6 3 x 3 = 9
          84:  1 x 4 = 4 2 x 4 = 8 3 x 4 = 12 4 x 4 = 16
          138: 1 x 5 = 5 2 x 5 = 10 3 x 5 = 15 4 x 5 = 20 5 x 5 = 25
          205: 1 x 6 = 6 2 x 6 = 12 3 x 6 = 18 4 x 6 = 24 5 x 6 = 30 6 x 6 = 36
          285: 1 x 7 = 7 2 x 7 = 14 3 x 7 = 21 4 x 7 = 28 5 x 7 = 35 6 x 7 = 42 7 x 7 = 49
          378: 1 x 8 = 8 2 x 8 = 16 3 x 8 = 24 4 x 8 = 32 5 x 8 = 40 6 x 8 = 48 7 x 8 = 56
            8 x 8 = 64
          484: 1 x 9 = 9 2 x 9 = 18 3 x 9 = 27 4 x 9 = 36 5 x 9 = 45 6 x 9 = 54 7 x 9 = 63
            8 x 9 = 72 9 x 9 = 81
          603:
```

https://doi.org/10.15215/remix/9781998944088.01

The output above shows where each line starts. For example, the end of the file is at 603. So with this information, the code using the readline([size]) method to read the multiplication table previously given can be easily revised to the following:

```
f = open('./my9x9table0.txt', 'r')
while f.tell() != 603:   # we use the read location to
identify if it has reached the end of the file
  line = f.readline()
  print(line, end = '')
f.close()
```

What if we want to read from a specific point in the file? To do that, we need to move the pointer to that point. That brings us to the second method of adjusting the pointer, which is f.seek(offset, start), in which the offset is how much to move it from the start. The default value for start is the current position of the pointer in the file. It would have been 0 when the file opened.

So suppose we want to read the line of the multiplication table 138 bytes from the beginning of the file. We would have to move the pointer to 138 first. From 0 at the beginning, the offset would be 138 as well. The code is shown below:

```
In [ ]:   f = open('./my9x9table0.txt', 'r')
          f.seek(138)
          pt = f.tell()
          line = f.readline()
          print('{:3d}: {:s}'.format(pt, line), end = '')

          f.close()

Out [ ]:  138: 1 x 5 = 5 2 x 5 = 10 3 x 5 = 15 4 x 5 = 20 5 x 5 = 25
```

Update Existing Content of a Text File

In a text file, how do we replace an existing line with something else? To achieve this, we need to take the following steps:

1. Open the file in r+ mode
2. Find out the position of the line
3. Move the file pointer to that position using f.seek(offset, start)
4. Write whatever you want to the file, which will replace the original content on that line

https://doi.org/10.15215/remix/9781998944088.01

An example of such an update is shown below:

```
In [ ]:  f = open('./my9x9table0.txt', 'r+')
         f.seek(138)
         pt = f.tell()
         line = f.readline()
         f.seek(138, 0)
         f.write("update this line\n")
         print('{:3d}: {:s}'.format(pt, line), end = '')

         f.close()
```

```
Out [ ]: 138: 1 x 5 = 5 2 x 5 = 10 3 x 5 = 15 4 x 5 = 20 5 x 5 = 25
```

The updated content of the file is shown below:

```
1 x 1 = 1
1 x 2 = 2     2 x 2 = 4
1 x 3 = 3     2 x 3 = 6      3 x 3 = 9
1 x 4 = 4     2 x 4 = 8      3 x 4 = 12     4 x 4 = 16
1 x 5 = 5 2 x 5 = 1update this line
= 20     5 x 5 = 25
```

Note that if there is less new content than the original, only part of the original is replaced.

We can also replace a single original line with multiple lines, as shown below:

```
In [ ]:  f = open('./my9x9table0.txt', 'r+')
         f.seek(138)
         pt = f.tell()
         line = f.readline()
         f.seek(138, 0)
         f.write("we write a line at the current position\n")
         f.write("we write another line below\n")
         f.write("we add third line below the two lines already
          written\n")
         print('{:3d}: {:s}'.format(pt, line), end = '')

         f.close()
```

```
Out [ ]: 138: 1 x 5 = 5 2 x 5 = 10 3 x 5 = 15 4 x 5 = 20 5 x 5 = 25
```

The updated file is shown below:

https://doi.org/10.15215/remix/9781998944088.01

```
1 x 1 = 1
1 x 2 = 2     2 x 2 = 4
1 x 3 = 3     2 x 3 = 6      3 x 3 = 9
1 x 4 = 4     2 x 4 = 8      3 x 4 = 12     4 x 4 = 16
1 x 5 = 5 2 x 5 = 1we write a line at the current position
we write another line below
we add third line below the two lines already written
4 3 x 7 = 21 4 x 7 = 28 5 x 7 = 35 6 x 7 = 42 7 x 7 = 49
1 x 8 = 8 2 x 8 = 16 3 x 8 = 24 4 x 8 = 32 5 x 8 = 40 6 x 8 = 48 7 x 8 =
56 8 x 8 = 64
1 x 9 = 9 2 x 9 = 18 3 x 9 = 27 4 x 9 = 36 5 x 9 = 45 6 x 9 = 54 7 x 9 =
63 8 x 9 = 72 9 x 9 = 81
```

However, if the total size of the new written data is longer than the line being replaced, part of the line or lines under will be overwritten. Therefore, if you want to replace a specific line of the text file exactly, you will need to write just enough to cover the existing data—no more, no less.

Deleting Portion of a Text File

To delete a portion of existing data from a file, you will need to use the f.truncate([size]). If the optional size argument is given, the file will be truncated to that size or the size of the file. Otherwise, the file will be truncated to the current file position. So if a file is freshly opened in w, w+, a, or a+ mode, f.truncate() will remove all data from the beginning to the end of the file. Please note that if the optional size argument is given and greater than 0, the file must be opened in a or a+ mode in order to truncate the file to the expected size.

```
f = open('./my9x9table0.txt', 'a')
f.truncate(399)

f.close()
```

The resulting content of the file is shown below:

```
1 x 1 = 1
1 x 2 = 2 2 x 2 = 4
1 x 3 = 3 2 x 3 = 6 3 x 3 = 9
1 x 4 = 4 2 x 4 = 8 3 x 4 = 12 4 x 4 = 16
1 x 5 = 5 2 x 5 = 1we write a line at the current position
we write another line below
we add third line below the two lines already written
4 3 x 7 = 21 4 x 7 = 28 5 x 7 = 35 6 x 7 = 42 7 x 7 = 49
1 x 8 = 8 2 x 8 = 16 3 x 8 = 24 4 x 8 = 32 5 x 8 = 40 6 x 8 = 48 7 x 8
= 56
```

https://doi.org/10.15215/remix/9781998944088.01

Please note that only part of the content in the file is left.

If the file is opened in w or w+ mode, the file will be truncated to a size of 0 regardless.

With all we have learned so far, we are ready to design and code a program to analyze an article stored in a text file. The program is shown in Table 5-8.

Table 5-8: Case study: How to create a word list

The problem	Different people have different styles when writing articles. These styles may include the words, phrases, and even sentences used most often in their writing. In this case study, we will develop a program that analyzes an article stored as a text file to create a list of the words used most often in the article.
The analysis and design	To analyze the article, we need to read the file into memory, build a list of words used in the article, then count how many times each word appears in the article. Because the file needs to be read line by line, we will read, analyze, and count the words in each line. How can we store the result containing the unique words and the number of times each word appeared in the article? Recall what we learned about dictionaries: each unique word can be used as a key, and the number of times the word appears can be the value. We then just need to determine the words used most often in the article. The best way to do that would be to sort the items of the dictionary based on the values (instead of keys), then take the first 10 items as the result. The algorithm is as follows: 1. Prepare by creating a list of punctuation marks and a list of nonessential words that can be ignored. 2. Initialize by setting the counter to 0, w_dict = {}. 3. Read the first line from the article into the memory. 4. Build a list of all words within the line: a. replace all the punctuation marks with whitespace b. split the entire article at whitespace to build a list of words c. remove all nonessential words from the list 5. Add the length of the list to the word counter: a. get a word from the list b. if the word is already in w_dict, increase the value by 1; otherwise, add dictionary item (word:1) to w_dict c. repeat step 5 6. Repeat steps 3–5 on the next line until there are no remaining lines in the file. 7. Sort w_dict based on the values. 8. Print out the first 10 items to show the words used most often by the article's author. Please note that this is just an example of programming with text files and Python dictionaries. The underlying theory about writing style may not be sound. The words that appear most often in an article may also be relevant to the topics covered by the article.

(continued on next page)

https://doi.org/10.15215/remix/9781998944088.01

Table 5-8: Case study: How to create a word list *(continued)*

The code

```
"""
This program finds out the frequencies of all words
 used in a news article stored in a text file.
"""
def news_analysis(article="WritingProposal.txt"):
  f = open(article, 'r')
  w_dict = {}   # we use dictionary to store the
 result, the key is the word, the value is times
 used
  punctuations = ['//', '-', ',', '.', ':', '"',
'[', ']', '(', ')', ';', '!', '?', '\n']
  less_meaningful = ['the', 'a', 'an', 'to', 'of',
'and', 'was', 'is', 'are', 'in', 'on']
  space, words_total, unique_count = ' ', 0, 0
  for m in f:   # this will lead the PVM to read the
 file line by line
    for c in punctuations:
      m = m.replace(c, ' ')
    words = m.split(f"{space}")
    for w in words:
      if w == '':
        continue
      else:
        words_total += 1   # count total number of
 all words
    for w in less_meaningful:
      m = m.replace(f' {w} ', ' ')
      m = m.replace(f' {w.capitalize()} ', ' ')
    words = m.split(f"{space}")
    for w in words:
      if w == '':
        continue
      else:
        if w not in w_dict.keys():
          unique_count += 1
          w_dict.update({w: 1})
        else:
          w_dict[w] += 1
  f.close()
  return words_total, [(k, v) for k, v in sorted(w_
 dict.items(), reverse=True, key=lambda item:
 item[1])]
result = news_analysis()
print(f"The article has {result[0]} words in total")
print(f"The number of unique words used is
 {len(result[1])}")
for i in range(20):
  print(f'word "{result[1][i][0]}" used
 {result[1][i][1]} times.')
```

https://doi.org/10.15215/remix/9781998944088.01

Table 5-8: Case study: How to create a word list *(continued)*

The result	The article has 926 words in total The number of unique words used is 467 word "will" used 20 times. word "be" used 13 times. word "NATO" used 13 times. word "https" used 13 times. word "research" used 10 times. word "interoperability" used 9 times. word "as" used 9 times. word "Information" used 8 times. word "standards" used 7 times. word "military" used 7 times. word "Interoperability" used 7 times. word "for" used 7 times. word "www" used 7 times. word "such" used 6 times. word "This" used 5 times. word "this" used 4 times. word "Canada" used 4 times. word "essay" used 4 times. word "exchange" used 4 times. word "MIP" used 4 times.

Chapter Summary

- Computer intelligence is achieved by computing and information processing.
- Both computing and information processing involve the manipulation of data.
- Simple types of data such as integers, floats, characters, and bools are the fundamental elements for complex data types.
- Unlike other languages, such as C, Python doesn't have a separate data type for single characters. In Python, a single character is a string whose length is 1.
- Python has two constants, True and False, defined as values of type bool. However, Python also treats 0, None, and empty string, empty list, empty tuple, empty set, and empty dictionary as False and treats all other values/objects as True.
- Python has some special data values/constants that don't belong to any ordinary data type. These special values/constants include None, NotImplemented, Ellipsis, and __debug__.
- Strings, lists, tuples, sets, and dictionaries are the compound data types in Python.

https://doi.org/10.15215/remix/9781998944088.01

- A string is a sequence of characters (ASCII, Unicode, or another encoding standard).
- A list is a sequence of data in a pair of square brackets [], such as [1, 2, 3, 4, 5].
- A tuple is a sequence of data in a pair of parentheses (), such as (1, 2, 3, 4, 5).
- The difference between a list and a tuple is that a list is mutable whereas a tuple is immutable, which means that once created, the data member of a tuple cannot be changed.
- Characters in a string and members of a list or a tuple are indexed from 0 to n − 1, where n is the length of the string, list, or tuple.
- The character at place j of string s can be accessed using s[j].
- Similarly, a data member at place j of a list or tuple x can be accessed using x[j].
- Strings, lists, and tuples are collectively called sequences.
- A slice of sequence (string/list/tuple) x can be taken using x[i:j], in which i specifies where the slice starts and j specifies that the slice should end right before location j.
- A number of operators and functions are available for constructing and manipulating strings, lists, or tuples.
- String, list, and tuple objects also have a number of methods available for constructing and manipulating strings, lists, or tuples.
- Some operators, functions, and methods are common for strings, lists, and tuples.
- A set is a collection of unique data enclosed by a pair of curly brackets {}, such as {1, 2, 3, 5}.
- Members of a set s are unordered, which means that they cannot be accessed using notion S[j], for example.
- A set has some very special functions and methods from other compound data types in Python.
- A dictionary is a collection of keys: value pairs enclosed by a pair of curly brackets {}, such as {'one':1, 'two':2, 'three':3, 'five':5}.
- Members of a set s are unordered, which means that in a dictionary, there is no such thing as a member at location j, for example.
- However, the value of a dictionary d can be accessed using the key associated with the value with the notion of d[k], which refers to the value whose associated key is k.
- Some special methods are defined for the operations of dictionaries.
- Files are important for storing data and information permanently.
- Files include text files and binary files.

https://doi.org/10.15215/remix/9781998944088.01

- The basic operations of files include create, read, write, append, and expand.
- A new file can be created when open with the w or x flag, when the file doesn't already exist. Opening a file with the w flag will overwrite the existing content of the file.
- To prevent data already in a file from being overwritten, open a file with the a flag or x flag. The a flag will open the file for appending new data to the end of the existing content of the file, while the x flag will not open the file if it already exists.
- Open a file with the t flag to indicate that the file is a text file.
- Open a file with the b flag to indicate that the file is a binary file.
- After reading or writing a file, use the close() file object method to close the file.

Exercises

1. Mentally run the following code blocks and write down the output of each code block.
   a.
```
course = 'comp218 – introduction to programming in
Python'
print(f'The length of \'{course}\' is
{len(course)}')
```
   b.
```
course = 'comp218 – introduction to programming in
Python'
print(f'The length of \'{course[10:22]}\' is
{len(course[10:22])}')
```
   c.
```
ls = list(range(9))
print(ls[2:5])
```
   d.
```
asc = {chr(c) for c in range(ord('A'),
ord('Z')+1)}
print(asc)
```
   e.
```
l0 = [i*2+1 for i in range(10)]
print(l0[2])
```
   f.
```
combo = [year + str(month+1) for year in ['2015',
'2016'] for month in range(6)]
print(combo)
```
   g.
```
s0 = 'Python '
s1 = 'is my language!'
print(s0+s1)
```

https://doi.org/10.15215/remix/9781998944088.01

Projects

1. Write a program that reads a text from a user, then counts and displays how many words and how many alphanumeric letters are in the text.
2. Write a program that
   a. reads a series of numbers that are separated by whitespace and uses a new line to end the input, then converts the numbers in the input string and puts them into a list.
   b. sorts the numbers in the list in descending order, using the sort() list object method.
   c. sorts the numbers in the list in descending order, using the Python built-in function sorted().

   Write your own code to sort the numbers in the list in ascending order without using the sort() method or sorted() function.
3. Sorting is a very important operation in computing and information processing because it is much easier to find a particular item (a number or a word) from a large collection of items if the items have been sorted in some manner. In computer science, many algorithms have been developed, among which selection sort, bubble sort, insertion sort, merge sort, quick sort, and heap sort are the fundamental ones. For this project, search the internet for articles about these sorting algorithms. Choose one to sort a list of integers.
4. Every course offered at universities has a course number and a title. For this project, write an application that uses a dictionary to save course information, allows users to add a course into the dictionary, and allows a user to get the title of a course for a given course number. The application should perform the following functions:
   a. Get a course number and name from a user and add an item, with the course number as key and the name as value, to the dictionary if the course doesn't already exist in the dictionary.
   b. Get a course number from a user, then find out the name of the course.
   c. Display a list of all the courses in the dictionary showing the course numbers as well as names.
   d. Quit the application.

   Hint: You will need a top-level ***while*** loop, which displays a menu showing the four options then acts accordingly.
5. This project is about text analysis. Find a news article on the internet, analyze the content, and generate and display some statistical data from the article. The detailed requirements are as follows:

https://doi.org/10.15215/remix/9781998944088.01

   a. Find a news article on the internet and save it as a text file on your computer.
   b. Have your program build a list of words in the article while reading the news content from the file.
   c. Generate and display the following statistics of the article:
      i. the total number of words in the article
      ii. a list of unique words
      iii. the frequency of each unique word in the article
      iv. a short list of words that represent the essence of the article
      v. a table with the above data nicely presented

6. Cryptography is the study of theory and technology for the protection of confidential documents in transmission or storage. It involves both encryption and decryption. In any cryptographic scheme, encryption is the process of converting plaintext to ciphertext according to a given algorithm using an encryption key, whereas decryption is the process of converting encrypted text (ciphertext) back to plaintext according to a given algorithm using a decryption key. If the encryption key and decryption key are the same in a cryptographic scheme, the scheme is a symmetric cryptographic scheme; if the two keys are different, the scheme is an asymmetrical cryptographic scheme.

   Among the many cryptographic schemes, substitution is a classic one, though the scheme is prone to frequency analysis attack. Write a program that can
   a. automatically generate a substitution key and add it to a key list stored in a file.
   b. display the substitution keys in the file.
   c. allow the user to choose a key in the list and encrypt some text taken from the user.
   d. allow the user to choose a key to decrypt an encrypted text taken from the user.
   e. allow the user to choose a key, encrypt the content of a text file, and save the encrypted content into a different file.
   f. allow a user to choose a key, decrypt the encrypted content in a file, and display the plaintext decrypted content.

https://doi.org/10.15215/remix/9781998944088.01

This page intentionally left blank

Chapter 6

Define and Use Functions

In programming or software development, program codes must be well structured to be manageable. Some program codes can be reused to make programming and software development more efficient. Functions and modules serve these two goals. Chapter 6 shows you how to define and use functions in Python and how to make and use modules in programming.

Learning Objectives

After completing this chapter, you should be able to

- explain what functions are in Python.
- define new functions correctly.
- use functions, including both built-in and programmer-defined functions.
- use ***return*** statements properly to return various values from a function.
- use positional arguments in defining and using functions.
- use variable-length lists of arguments in defining and using functions.
- use keyed arguments in defining and using functions.
- use positional arguments, keyed arguments, and variable-length lists of arguments.
- explain what recursive functions are and how they work.
- define and use recursive functions.
- explain what anonymous/***lambda*** functions are.
- define and use anonymous/***lambda*** functions.
- use special functions such as mapping, filtering, and reducing.
- explain what generators are and what advantages they have.
- define a function as a generator.

https://doi.org/10.15215/remix/9781998944088.01

- explain what closures and decorators are and how they are used.
- define and use closures and decorators.
- describe the properties of functions and use them properly.

6.1 Defining and Using Functions in Python

You have already seen and used some built-in functions in the previous chapters. These built-in functions are built into Python Virtual Machine (PVM) in its standard distribution so that you can use them without importing any Python modules. For example, the built-in function sum can be used directly to calculate the sum of a sequence of numbers, as shown in the following code sample:

```
>>> sum([12, 23, 25, 65, 52])
177
```

As you will see, many Python modules, available via either Python distribution or a third party, have also defined functions ready for you to use in your programs.

For functions defined in a standard or third-party module, you must import the module before you can call the function by using the dot notation or operator, such as m.f(...), where m is a name referring to the module, and f is the name of the function to be used. Within the pair of parentheses are data as arguments of the function call, to be passed to the function for processing. The following is an example calling the pow() function from the math module:

```
>>> import math
>>> math.pow(35,12)
3.7922050805664 05e+18
```

Note that when calling a function, a pair of parentheses must be attached to the name of the function, even if there is no argument to pass. Otherwise, the name of the function will be evaluated as a first-class object, and the type of the object and the name will be returned, as shown in the following example:

```
>>> print(sum)
<built-in function sum>
>>> print(id)
<built-in function id>
```

https://doi.org/10.15215/remix/9781998944088.01

Although many modules have been developed by others and many functions have been made available, programmers do need to define their own functions for their specific purposes.

To define a function in Python, the ***def*** compound statement is used. The general syntax is as follows:

```
def <function name>(parameters):
  <code block>
```

Where function name is a legitimate identifier in the local scope, parameters are legitimate variable names that can be used to pass values (arguments) to the function, and a code block (function body, in this compound statement) is the real program code that does the computing or information processing. What makes a code block in a function definition different from code blocks in other compound statements such as ***for***, ***while***, and ***if*** is that it will always return a value with the ***return*** statement. Even if you do not have anything to return from a function definition and do not have a ***return*** statement, special value None will still be automatically returned from the function.

The following is a real example showing how a function is defined in Python:

In [ ]:
```
def factorial(n):
"""This function calculates and returns n!, the
  factorial of an integer n > 0."""
  if (not isinstance (n, int)) or (n < 1):
    return None
  r = 1
  for i in range(n):
    r *= (i + 1)
    return r
```

As documented in the docstring, the function is to calculate and return the factorial of an integer if the number is greater than 0; otherwise, it returns None.

The function you defined can be used in the same way as built-in functions. The following example shows how to call the factorial function defined above:

In [ ]:
```
N = 16
fn = factorial(N)
print(f'factorial of {N} is {fn}.')
```

Out [ ]:
```
factorial of 16 is 20922789888000.
```

https://doi.org/10.15215/remix/9781998944088.01

Sometimes you need to return more than one value from a function. To do that, you can either put the values in a compound data type such as a list, tuple, set, or dictionary, or just put the value all behind ***return***. In the latter case, the values will be automatically packed in a tuple by the ***return*** statement.

The following example calculates and returns both the quotient and remainder of two integers at the same time.

In [ ]:
```
def idivmod(n, m):
  if (not isinstance(n, int)) or (not isinstance(n,
 int)) or (m == 0):
    return None
  return n // m, n % m

n, m = 23, 5
print(f'The value returned from idivmod({n}, {m}) is
 {idivmod(n, m)}.')
```

Out [ ]:
```
The value returned from idivmod(23, 5) is (4, 3).
```

In the remainder of this section, we show how to program to solve slightly more complicated problems with Python.

A perfect number is an integer that equals the sum of all its factors excluding the number itself. For example, 6 is a perfect number because 6 = 1 + 2 + 3. So is 28. It sounds simple, but the next perfect number is very far from 28: 496. A program for finding perfect numbers is shown in Table 6-1.

Table 6-1: Case study: How to find perfect numbers

The problem	In this case study, we are going to write a program to ask for a big integer from the user, then find all the perfect numbers smaller than the big integer.
The analysis and design	Step 1. Take an input from user, and convert it into an integer Step 2. Loop from 2 to the big integer a. Test each integer to see if it is a perfect number b. If yes, print the number and all its factors Step 3. Finish Steps to check if a number is a perfect number: Step 4. Find all its factors, including 1 but excluding the number itself, and put them into a list Step 5. Sum up the factors with sum(factors) Step 6. If the number == sum(factors), then return True.

https://doi.org/10.15215/remix/9781998944088.01

Table 6-1: Case study: How to find perfect numbers *(continued)*

The code

```
"""
This program is used to find all the perfect
 numbers that are less than N given by a user.
"""
def perfect(n):
  factor = [1]   # create a list with 1 as the
  first factor
  for j in range(2, (n // 2) + 1):   # only need
  loop to n // 2 + 1
    if n % j == 0:   # if j is a factor
      factor.append(j)   # add j to the list
    if n == sum(factor):   # if the sum of the
  factors = n
      return [True, factor] # return True as well
  as factors
    else:
      return [False, []]
upper_bound = int(input("Tell me the upper
 bound:"))
for i in range(2, upper_bound):
  test = perfect(i)
  if test[0]:
    print(f"{i} = {test[1]}")
```

The result

```
Tell me the upper bound:32198765
6 = [1, 2, 3]
28 = [1, 2, 4, 7, 14]
496 = [1, 2, 4, 8, 16, 31, 62, 124, 248]
8128 = [1, 2, 4, 8, 16, 32, 64, 127, 254, 508, 1016, 2032, 4064]
```

6.2 Parameters and Arguments in Functions

When you define a function, you can use variables within the parentheses right next to the function name to specify what values can be taken when the function is called. These variables within the parentheses are called parameters, and the values to be passed to the parameters in a function call are called arguments.

In Python, a function call may take positional arguments, keyword arguments, variable-length lists of nonkeyword arguments, variable-length lists of keyword arguments, and default arguments. These are determined in the definition of the function.

In a function definition, a simple variable name can be used for a parameter expecting a positional argument or keyword argument, to give a parameter, such as x, a default V using assignment x = V. When a parameter is given a default

https://doi.org/10.15215/remix/9781998944088.01

value, the default value will be used when no value is given to the parameter in a function call.

To indicate that parameter y will hold a variable-length list of nonkeyword arguments, use *y; to indicate that parameter z will take variable-length of keyword arguments, use **z. Keyword arguments will be explained shortly.

When calling a function, positional arguments are the arguments passed to their respective parameters in accordance with their positions. That is, the first parameter in a function call is passed to the first argument in the definition of the function, the second parameter is passed to the second argument, and so on. This is illustrated in the example below:

```
In [ ]:   def func_demo1(a, b, c):
            print(f'a = {a}, b = {b}, c = {c}')

          func_demo1(1, 2, 3)

Out [ ]:  a = 1, b = 2, c = 3
```

This shows that the first argument was passed to the first parameter a, the second argument was passed to the second parameter b, and the third argument was passed to the third parameter c.

When calling a function, its parameter name, such as x, can be used as a keyword to explicitly indicate that a specific value, such as v, will be passed to x. This is done using x = v syntax, called a keyword argument in a function call. The following example shows how keyword arguments are used.

```
In [ ]:   def func_demo1(a, b, c):
            print(f'a = {a}, b = {b}, c = {c}')

          func_demo1(b =1, a = 2, c = 3)

Out [ ]:  a = 2, b = 1, c = 3
```

In this case, the order of the arguments does not matter because the code explicitly indicates which argument is given to which parameter. Please note that in a function call when keyword argument is used, no more positional arguments, except variable-length nonkeyword arguments, may follow. An error will occur otherwise, as shown below:

https://doi.org/10.15215/remix/9781998944088.01

```
In [ ]:  def func_demo1(a, b, c):
           print(f'a = {a}, b = {b}, c = {c}')

         func_demo1(b=1, 2, 3)
```

```
Out [ ]: File "<ipython-input-51-6539f4d878e5>", line4
         func_demo1(b = 1, 2, 3)
           ^
         SyntaxError: positional argument follows keyword argument
```

Also, in a function definition, parameters expected to be used as keywords must be placed behind those expecting positional arguments. An error will occur otherwise, as shown in the following example:

```
In [ ]:  def func_demo1(a, b, c):
           print(f'a = {a}, b = {b}, c = {c}')

         func_demo1(3, 1, a = 2)
```

```
Out [ ]: ---------------------------------
         TypeError   Traceback (most recent call last)
         <ipython-input-185-f540d2127799> in <module>
          2 print(f'a = {a}, b = {b}, c = {c}')
          3 ----->
          4 func_demo1(3, 1, a = 2)
         TypeError : func_demo1() got multiple values for argument 'a'
```

When a parameter in a function definition has a default value, the argument for the parameter can be omitted if the default value is to be used. The function defined in the following example will calculate the square of number x by default, but it can also calculate x power of y by passing a particular value to y:

```
In [ ]:  def powerof(x, y = 2):
           return f'{x} ** {y} = {x ** y}'

         print(powerof(12))
         print(powerof(23, 5))
         print(powerof(13, y = 6))
         print(powerof(y = 21, x = 3))   # with keyword
          arguments, the order doesn't matter
```

```
Out [ ]: 12 ** 2 = 144 23 ** 5 = 6436343 13 ** 6 = 4826809 3 ** 21 = 10460353203
```

https://doi.org/10.15215/remix/9781998944088.01

The following example demonstrates how to define a function that can take variable-length nonkeyword arguments. The function is to calculate the product of a series of numbers:

```
In [ ]:  def product(*n):
           s = 1
           for i in n:
             s *= i
           return f'Product of all numbers in {n} is {s}'

         print(product(1,2,5,32,67))
         print(product(11,32,25,3,7))
         print(product(19,12,15,322,6))
```

```
Out [ ]: Product of all numbers in (1, 2, 5, 32, 67) is 21440
         Product of all numbers in (11, 32, 25, 3, 7) is 184800
         Product of all numbers in (19, 12, 15, 322, 6) is 6607440
```

As can be seen, using variable-length nonkeyword positional arguments has made the function more powerful.

Sometimes in a function call, you may also want to use variable-length keyword arguments. The following example shows how this can be done:

```
In [ ]:  def reporting(**kwargs):
           for k, v in kwargs.items():
             print(f'{k}:{v}')
         reporting(First_name ='John', Last_name ='Doe', Sex
          ='Male')
         reporting()
```

```
Out [ ]: First_name:John
         Last_name:Doe
         Sex:Male
```

When calling a function with variable-length keyword arguments, the keywords look like parameter names to which the values are passed. However, in the case of variable-length keyword arguments, these keywords cannot be used as variables inside the function definition because they are not in the parameter list when the function is defined.

While variable-length nonkeyword arguments are passed to a function as a tuple, variable-length keyword arguments are passed to the function as a dictionary. As such, what can be achieved by using variable-length keyword

https://doi.org/10.15215/remix/9781998944088.01

arguments can also be achieved by passing a dictionary instead. For example, the above example can be easily rewritten as follows:

```
In [ ]:  def dict_reporting(ps):   # kw is a dictionary
           for k, v in ps.items():
             print(f'{k}:{v}')

         pdict = {'First_name':'John', 'Last_name':'Doe',
          'Sex':'Male'}
         dict_reporting(pdict)
```

```
Out [ ]: First_name:John
         Last_name:Doe
         Sex:Male
```

Similarly, functions taking variable-length nonkeyword arguments can be easily rewritten to take a tuple. So the example of list_product function can be rewritten as follows:

```
In [ ]:  def product_tuple(nt):
           s = 1
           for i in nt:
             s *= i
           return f'Product of all numbers in {nt} is {s}'

         print(product_tuple((1,2,5,32,67)))
         print(product_tuple((11,32,25,3,7)))
         print(product_tuple((19,12,15,322,6)))
```

```
Out [ ]: Product of all numbers in (1, 2, 5, 32, 67) is 21440
         Product of all numbers in (11, 32, 25, 3, 7) is 184800
         Product of all numbers in (19, 12, 15, 322, 6) is 6607440
```

The difference between using variable-length arguments and passing a tuple or dictionary is that, because the length of variable-length arguments can be 0, you do not have to pass any argument at all if you don't have one. With a tuple or dictionary, on the other hand, you would have to have a legitimate argument for the corresponding parameter in the function definition unless you have set a default value to it. Also, as can be seen from the two examples above, in some applications, using variable-length arguments is more natural and elegant.

https://doi.org/10.15215/remix/9781998944088.01

CODING PRACTICE

We know that given a, b, and c for equation $ax^2 + bx + c = 0$, x is $(-b + \text{sqrt}(b^2 - 4ac)) / 2a$, or $(-b - \text{sqrt}(b^2 - 4ac)) / 2a$.

Design a program that will take three numbers with the ***input*** statement, and then solve quadratic equation ax2 + bx + c = 0 by calling a function solve quadratic(a, b, c) which you need to design. The two solutions x1, x2 should be returned in a tuple as (x1, x2).

6.3 Recursive Functions

A function is recursive if it calls itself either directly or indirectly. Recursion is a powerful concept in computing and computational theory. In computational theory, it has been proven that any problem computable by modern computers can be represented as a recursive function.

In programming, recursive functions do not make your programs run fast. However, they do provide a powerful means of algorithm design to solve a problem and make neater program code.

Take the factorial function n! as an example. You know that n! is defined as 1 * 2 * 3 * ... * n. If you use fac(n) to refer to the factorial function, you cannot simply define the function in Python as

```
def fac(n):
  return 1 * 2 * 3 * … * n   # this does not work in
Python
```

because the dots (...) do not make sense to computers in this context. With what you have learned so far, the function can be defined as

```
def fac(n):
  if n == 0:
    return 1
  product = 1
  for i in range(n):   # using loop
    product *= (i + 1)
  return product
```

The function above has seven lines of code.

https://doi.org/10.15215/remix/9781998944088.01

Since you know that 0! = 1, and n! can be computed as n * (n − 1)! if n > 0, you can program a recursive function in Python to solve the factorial problem, as shown in the case study in Table 6-2.

Table 6-2: Case study: How to use a recursive function for calculating factorial n

The problem	Define a recursive function to calculate factorial n as n!
The analysis and design	We know that 0! = 1, and n! can be computed as n * (n − 1)! when n > 0. So if we use fac(n) to denote n!, this will be translated as fac(n) = 1 if n = 0 and as fac(n)= n * fac(n − 1) if n > 0. Accordingly, we can define a recursive factorial function in Python as follows:

The code	`def fac(n):`
	`  if n == 0:`
	`  return 1`
	`  else:`
	`  return n * fac(n - 1)`
	
	`n = 9`
	`print(f"{n}! = {fac(n)}")`

The result	9! = 362880

As you can see, the function has become shorter and neater, although it often takes more memory and more time to run.

The next case study, in Table 6-3, shows how a recursive function can be used to find the greatest common divisor of two integers.

CODING ALERT

What will happen if fac(n) is called with n < 0? For example, as fac(−9)?

Table 6-3: Case study: How to use a recursive function

The problem	This simple problem aims to find the greatest common divisor (GCD) for two given integers. A common divisor of two integers is an integer that can divide both integers, and the GCD is the biggest one among the common divisors. For example, 1 and 2 are common divisors of 4 and 6, and 2 is the GCD of 4 and 6.
The analysis and design	At first glance, a straightforward approach is to find all the divisors for each integer,

(continued on next page)

https://doi.org/10.15215/remix/9781998944088.01

Table 6-3: Case study: How to use a recursive function *(continued)*

The code

```
"""
Ask for two integers and find the GCD of the two
 using the improved Euclidean algorithm.
"""


def my_gcd(a, b):
  global depth
  if b == 0:   # condition to finish
    return b
  else:
    b, a = sorted((abs(a - b), b))   # sort and
 reassign
    depth += 1   # recursion depth increased by 1
    print(f"recursion #{depth} for{(a, b)}")
    return my_gcd(a, b)


i = int(input("Tell me the first integer:"))
j = int(input("Tell me the second integer:"))
if i < j:
  i, j = j, i
depth = 0
print(f"The greatest common divisor of {i} and {j}
 is {my_gcd(i, j)} after {depth} recursions.")
```

The result

```
Tell me the first integer:3238
Tell me the second integer:326
recursion #1 for (2912, 326)
recursion #2 for (2586, 326)
recursion #3 for (2260, 326)
recursion #4 for (1934, 326)
recursion #5 for (1608, 326)
recursion #6 for (1282, 326)
recursion #7 for (956, 326)
recursion #8 for (630, 326)
recursion #9 for (326, 304)
recursion #10 for (304, 22)
recursion #11 for (282, 22)
recursion #12 for (260, 22)
recursion #13 for (238, 22)
recursion #14 for (216, 22)
recursion #15 for (194, 22)
recursion #16 for (172, 22)
recursion #17 for (150, 22)
recursion #18 for (128, 22)
recursion #19 for (106, 22)
recursion #20 for (84, 22)
recursion #21 for (62, 22)
recursion #22 for (40, 22)
recursion #23 for (22, 18)
recursion #24 for (18, 4)
recursion #25 for (14, 4)
recursion #26 for (10, 4)
recursion #27 for (6, 4)
recursion #28 for (4, 2)
recursion #29 for (2, 2)
recursion #30 for (2, 0)
The greatest common divisor of 3238 and 326 is 2 after 30
  recursions.
```

https://doi.org/10.15215/remix/9781998944088.01

As noted, the program took 30 recursions to find the GCD of 3238 and 326. Can we make it more efficient? The answer is yes, and it is fun to design and code a better and faster program to solve a problem, as shown in the case study of this same problem in Table 6-4.

Table 6-4: Case study: How to use a recursive function—revised

The problem	This simple problem aims to find the greatest common divisor (GCD) for two given integers. A common divisor of two integers is an integer that can divide both integers, and the GCD is the biggest one among the common divisors. For example, 1 and 2 are common divisors of 4 and 6, and 2 is the GCD of 4 and 6.
The analysis and design	The original Euclidean algorithm is great because it only needs subtraction to find out the greatest common divisor, but sometimes it will involve too many steps, especially if we do the calculation manually. For example, to find the greatest common divisor of 4 and 40000, one needs to complete 10000 subtractions to find out that 4 is the GCD. An obvious and straightforward improvement to the algorithm is to use modular operation in place of subtraction.

The code	`"""`
	`Ask for two integers and find the GCD of the two using the improved Euclidean algorithm.`
	`"""`
	
	`def my_gcd(a, b):`
	`  global depth`
	`  if b == 0:  # condition to exit from recursion`
	`  return a`
	`  else:`
	`  b, a = sorted(((a % b), b))  # sort and reassign`
	`  depth += 1  # recursion depth increased by 1`
	`  print(f"recursion # {depth} for {(a, b)}")`
	`  return my_gcd(a, b)`
	
	`i = int(input("Tell me the first integer:"))`
	`j = int(input("Tell me the second integer:"))`
	`if i < j:`
	`  i, j = j, i`
	`depth = 0`
	`print(f"The greatest common divisor of {i} and {j} is {my_gcd(i, j)} after {depth} recursions.")`

The result	Tell me the first integer:3238 Tell me the second integer:326 recursion #1 for (326, 304) recursion #2 for (304, 22) recursion #3 for (22, 18) recursion #4 for (18, 4) recursion #5 for (4, 2) recursion #6 for (2, 0) The greatest common divisor of 3238 and 326 is 2 after 6 recursions.

https://doi.org/10.15215/remix/9781998944088.01

As you can see, for the same numbers, the improved algorithm took only six recursions, whereas the original took 30. Another benefit of using the improved algorithm is that because Python Virtual Machine (PVM) has a limit on the maximum depth of recursions due to the limitations of computer memory, the improved algorithm will be able to handle much bigger numbers than the original algorithm. You may run the two programs on 24230336504090 and 356879542 to see the difference between the two algorithms.

6.4 Anonymous Functions: *lambda* Expressions

In previous sections, you saw functions with names, which make it possible to call a function by its name. Sometimes, especially when the operations of the function are simple and used only once, it is more convenient to simply use a small code block as a function without defining a function with a name. This is where anonymous functions or ***lambda*** expressions come to play.

The word *lambda* originates from lambda calculus, which has played an important role in the development of modern computational theory and functional programming. You are encouraged to search for lambda calculus on the internet for more details.

In Python, an anonymous function can be defined using the following syntax:

> lambda <formal argument list> : <expression whose value is to be returned>

In the above syntax, a formal argument list is a list of variables separated by commas but without surrounding parentheses, and everything behind the colon takes the role of the code block in the regular function definition. But it must be a single expression whose value is to be returned by the ***lambda*** function without a keyword return.

The following is an example of a ***lambda*** function in Python that is used to construct an odd number from a given integer:

```
>>> lambda n: 2 * n + 1
<function <lambda> at 0x012CBD20>
>>>
```

Because an anonymous function is meant to have no name, the common use of such a function is to have it called directly when it is defined, as shown below:

https://doi.org/10.15215/remix/9781998944088.01

```
>>> (lambda n: 2 * n + 1)(4)
9
```

Note that a pair of parentheses encloses the entire ***lambda*** expression to signify the end of the ***lambda*** expression.

A ***lambda*** expression can also have two or more formal arguments, as shown below:

```
>>> (lambda x, y: x + y)(3, 5)
8
```

In the example above, we first defined a ***lambda*** function within a pair of parentheses, then applied it to a list of two actual arguments within a pair of parentheses.

An anonymous function can even be defined to take a variable-length list of arguments, as shown in the following example:

```
>>> (lambda& * x: sum(x) * 3)(1, 2, 3, 4, 5)
15
```

Although an anonymous function is meant to have no name, that does not stop you from giving it a name, as shown in the next example:

```
>>> double = lambda x: 2 * x
```

We can then call the function with the name, as shown below:

```
>>> double(23)
46
```

Our next anonymous function takes two arguments and checks if one is a multiple of the other:

```
>>> is_multiple = lambda m, n: m % n == 0
>>> is_multiple(32, 4)
True
>>> is_multiple(32, 5)
False
```

The next section will show how ***lambda*** expressions can be used to program more effectively.

https://doi.org/10.15215/remix/9781998944088.01

6.5 Special Functions: Mapping, Filtering, and Reducing

As mentioned, Python treats everything as objects, including functions, which can be accessed in the same way as ordinary objects. The following is an example:

```
>>> f_objects = [abs, len, open]   # the list has three
functions as its members
>>> f_objects[0](-2)   # f_objects[0] refers to the first
item in the list, which is built-in function abs
12
>>> f_objects[1](f_objects)   # f_objects[1] refers to the
second item in the list, which is built-in function len
3
```

This has provided programmers with great possibilities. For example, Python has three special built-in functions that can take other functions as arguments and apply them to a list. These special functions include mapping, filtering, and reducing.

Mapping

It is easier to explain what mapping does with a code example:

```
>>> integers = [-12, 32, -67, -78, -90, 88]   # this list
has negative numbers
>>> list(map(abs, integers))   # this maps the function
abs to each integer in the list
[12, 32, 67, 78, 90, 88]
```

Note that abs() has been applied to every member of the list.

Basically, the map function can apply any function to as many lists as required for its arguments. The following is an example:

```
>>> def sum(a, b):   # this function requires two
arguments
… return a + b
…
>>> list(map(sum, [1, 2, 3], [5,8,9]))   # this maps the
sum function to two lists for the two arguments
[6, 10, 12]
```

https://doi.org/10.15215/remix/9781998944088.01

Given what you have already learned about anonymous functions, you can generate a list of odd numbers neatly, as follows:

```
>>> list(map(lambda n: 2 * n + 1, range(10)))
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19]
```

Similarly, we can generate all multiples of a number k, such as 3 in the example below:

```
>>> list(map(lambda n: 3 * n, range(10)))
[0, 3, 6, 9, 12, 15, 18, 21, 24, 27]
```

Filtering

The filtering function takes the same form as the mapping function but is used to extract items from the iterable that satisfy the Boolean filtering condition at hand. The following sample code keeps only even numbers in the generated list:

```
>>> def even(n):
… return n % 2 == 0   # if it can be divided by 2, then
return true; return false otherwise
…
>>> list(filter(even, range(10)))   # filter applies to
each even number and keeps only some
[0, 2, 4, 6, 8]
```

The code above can be neatly rewritten by using a ***lambda*** expression, as follows:

```
>>> list(filter(lambda n: n % 2 == 0, range(10)))
[0, 2, 4, 6, 8]
```

The filter function may play an important role in selecting items from a given list based on certain criteria, as shown in the following example:

```
>>> st = "The filter function may play an important role
in selecting words from a given text. In the following
example, only words that contain the letter o are selected"
>>> list(filter(lambda s: 'o' in s, st.split()))
['function', 'important', 'role', 'words', 'from',
'following', 'only', 'words', 'contains', 'o']
```

Note that only the words that include the letter o are selected.

https://doi.org/10.15215/remix/9781998944088.01

Reducing

The reduce function is not a built-in function but defined in the functools module. It takes the same form as the mapping and filtering functions but applies the function to items in the iterable progressively until the list is exhausted and reduced to a single value or object, as shown in the following example in JupyterLab:

```
In [ ]:  def pow(m, n):
           return m ** n

         from functools import reduce
         reduce(pow, range(2, 6))   # equal to pow(pow(pow(2,
          3), 4), 5)
```

```
Out [ ]: 1152921504606846976
```

We can recode the above operation with the ***lambda*** expression (anonymous function) we learned in the previous section, as shown below:

```
In [ ]:  reduce(lambda n, m: n ** m, range(2, 6))
```

```
Out [ ]: 1152921504606846976
```

As seen from the examples above, a ***lambda*** expression becomes handy when you use a function only once and the function is simple enough to be written with an expression.

6.6 Generators: Turning a Function into a Generator of Iterables

As you have seen, sequences (including lists, tuples, and strings) are an important way of organizing data. In addition to having a relatively static list accessible through a variable, Python also provides a means to make a generator that can generate members of a sequence dynamically.

Assume we want to find out a sequence of perfect numbers within a given range. Instead of finding all perfect numbers within the range and returning them in a list, we can define a generator of perfect numbers using the ***yield*** statement in place of the ***return*** statement, as shown in the following example:

https://doi.org/10.15215/remix/9781998944088.01

```
In [ ]:  def isPerfect(n):
           factor = [1]
           for j in range(2, (n // 2) + 1):
             if n % j == 0:
               factor.append(j)
             if n == sum(factor):
               return [True, factor]
             else:
               return [False, []]

         def perfectGenerator(m = 100):
           for i in range(m):
             testResult = isPerfect(i)
             if testResult[0]:
               yield [i, testResult[1]]

         myPerfects = perfectGenerator(10000)
         print(myPerfects)

Out [ ]: <generator object perfectGenerator at 0x00000191402373C8>
```

As we can see, instead of returning a list of all the perfect numbers and factors within the given range, the function actually returned a generator.

To get the next perfect number in the generator, we use the built-in function next, as shown below:

```
In [ ]:  print(next(myPerfects))
         print(next(myPerfects))
         print(next(myPerfects))
         print(next(myPerfects))
         print(next(myPerfects))

Out [ ]: [1, [1]]
         [6, [1, 2, 3]]
         [28, [1, 2, 4, 7, 14]]
         [496, [1, 2, 4, 8, 16, 31, 62, 124, 248]]
         [8128, [1, 2, 4, 8, 16, 32, 64, 127, 254, 508, 1016, 2032, 4064]]
```

Note that in the output above, we found five perfect numbers between 1 and 10000. Output on each line is a list, and the first member is a perfect number, whereas the second member contains a list of its factors (whose sum is equal to the first number).

Recall that we have used the built-in function range() with ***for*** loops. We can use user-defined generators with ***for*** loops as well, as shown below:

https://doi.org/10.15215/remix/9781998944088.01

In [ ]:
```
myPerfects = perfectGenerator(10000)
for p in myPerfects:
  print(p)
```

Out [ ]:
```
[1, [1]]
[6, [1, 2, 3]]
[28, [1, 2, 4, 7, 14]]
[496, [1, 2, 4, 8, 16, 31, 62, 124, 248]]
[8128, [1, 2, 4, 8, 16, 32, 64, 127, 254, 508, 1016, 2032, 4064]]
```

You may wonder why we need generators instead of returning a list of objects. The reason is for performance in terms of both speed and memory usage. The next two examples show the difference in speed between a generator and a normal function returning a list of perfect numbers.

In [ ]:
```
import time   # time module for time and timing
  related functions
from memory_profiler import profile   # a module
  for profiling memory usage
def isPerfect(n):
  factor = [1]
  for j in range(2, (n // 2) + 1):
    if n % j == 0:
      factor.append(j)
    if n == sum(factor):
      return [True, factor]
    else:
      return [False, []]

def getAllPerfects(n = 10):
  perfects=[]
  for i in range(1, n + 1):
    testResult = isPerfect(i)
    if testResult[0]:
      perfects.append((i, testResult[1]))
  return perfects

t0 = time.process_time()
perfectNumbers = getAllPerfects(10000)
t1 = time.process_time()

print(f"{t1 - t0} seconds are used to get a list
  of perfect numbers")
```

Out [ ]:
```
1.1875 seconds are used to get a list of perfect numbers
```

https://doi.org/10.15215/remix/9781998944088.01

In [ ]:
```
import time   # time module for time and timing related
 functions
from memory_profiler import profile   # a module for
 profiling memory usage

def isPerfect(n):
  factor = [1]
  for j in range(2, (n // 2) + 1):
    if n % j == 0:
      factor.append(j)
    if n == sum(factor):
      return [True, factor]
    else:
      return [False, []]

def perfectGenerator(m = 100):
  for i in range(m):
    testResult = isPerfect(i)
    if testResult[0]:
      yield [i, testResult[1]]

t0 = time.process_time()
myPerfects = perfectGenerator(10000)
t1 = time.process_time()
print(f"{t1 - t0} seconds are used to get a generator
 of perfect numbers")
```

Out [ ]:
```
0.0 seconds are used to get a generator of perfect numbers
```

The first code sample is to find a list of perfect numbers within a given range using a normal function to return a list of perfect numbers, whereas the second code sample uses a generator instead. As you can see, to get a generator of perfect numbers took no time (0.0 seconds), whereas using a function to return a list of perfect numbers took 1.1875 seconds.

The gain from returning a generator instead of a complete list from a function is even more obvious in terms of memory usage, because a list would usually take a bigger chunk of computer memory, which increases drastically along with the growth of the list, whereas for a list generator, the memory usage is determined by the coding of the function and will not change. In our examples above, the generator that generates perfect numbers from 1 to 100000 has almost the same number of lines of code as the function that produces a list of perfect numbers in the same range, but the function consumes much more memory than the generator because of the use of list. To get a sense of how much memory would be consumed by a bigger list, see the following example:

https://doi.org/10.15215/remix/9781998944088.01

In [ ]:
```
import sys
print(f"The size of a list containing numbers from 0 to
  100 is {sys.getsizeof(list(range(100)))}Bytes")
print(f"The size of list containing numbers from 0 to
  1000 is {sys.getsizeof(list(range(1000)))}Bytes")
print(f"The size of list containing numbers from 0 to
  10000 is {sys.getsizeof(list(range(10000)))}Bytes")
```

Out [ ]:
```
The size of a list containing numbers from 0 to 100 is 1008Bytes
The size of a list containing numbers from 0 to 1000 is 9112Bytes
The size of a list containing numbers from 0 to 10000 is 90112Bytes
```

As you can see, a simple list containing integer numbers from 0 to 10000 takes up almost 90KB of memory.

6.7 Closures: Turning a Function into a Closure

In Python and some other programming languages such as JavaScript, closures are the result of a nested function—that is, one function that is defined inside another—as shown in the following example:

In [ ]:
```
def outer(greeting):
    print('Good morning!')
    def inner(msg):
        print(msg)
    inner(greeting)

outer('Good day!')
```

Out [ ]:
```
Good morning!
Good day!
```

In the example above, the outer function can also return the inner function with the parameter list as a first-order object, as shown below:

In [ ]:
```
def outer():
    greeting = 'Good morning,'
    def greet(who):
        print(greeting, who)
    return greet

cl = outer() # cl will hold function greet
name = input('What is your name?')
cl(name) # local greeting in outer still attached
```

Out [ ]:
```
Good morning, Joe
```

The output was produced when you input Joe for the name.

https://doi.org/10.15215/remix/9781998944088.01

What has been returned from call of outer() is a function object. However, because of the way the function is defined and returned, the value of the variable greeting, defined locally within the outer function, has been attached to the inner function object. This is called closure—the binding of certain data to a function without actually executing the function.

6.8 Decorators: Using Function as a Decorator in Python

Python has many powerful means and program constructs that you can use to solve problems and get the jobs done. Decorators are one of them.

In Python, a decorator can be a function, but it is used to modify the functionality of other functions. Suppose we want to keep a log of the time a function is called, the parameters passed, and the time the call took to run. A decorator function can be defined as follows:

In [ ]:
```
import time # import the time module
# define a function as a decorator
def calllog(function_logged):   # calllog function will
 be used as a high order function and decorator
    def wrapper_function(*args, **kwargs):
        t0 = time.asctime()
        t1 = time.time()   # time started in seconds as
 a float number
    func_handle = function_logged(*args, **kwargs)
        t2 = time.time()   # time ended in seconds as a
 float number
        call_args = ''
        if args:
            call_args += str(args)
    if kwargs:
            call_args += str(kwargs)
        with open("calllog.txt", "w") as logs:
            log_str = f"Call to {function_logged.__
 name__}{call_args}\n"
            log_str += f"was made at {t0}, taking {t2
 - t1} \n"
            logs.write(log_str)
            print(f"Logging string written to file
 is:\n {log_str}")

    return func_handle

 return wrapper_function

@calllog   # calllog is used as decorator
def real_function(message = "Operation",m=2, n=3):
  print(message,'\n', f'{m}**{n}=\n', m**n)

real_function(message="Operation m to the power of n",
 m=123456789, n=199)
```

https://doi.org/10.15215/remix/9781998944088.01

```
Out [ ]:  Operation m to the power of n
          123456789**199=
          16273480830928460132417950594596569558877352891957597527186
            51786556685290423092376199892583061061074707965678477776694
            49795786420004667120197404189525472460559895652852033423201
            16359876319939446187437522130137245495140379083438313330693
            33398771946314482405115292896034144040950197611826514223063
            32156945596330253949476551315738321715899063841618733322264
            49224813255256278312840473976263295612792832084345917448681
            15534257875994135148844270652116711345095108067990108790434
            40495811615899347233266832259494809834578242654938317667418
            84144954140872264177848916671946547383801435396524232967278
            83973362461915655926318744303423013911299410323414821550184
            48536484622035552484585206711938245617330681531041555477839
            90870245864089998438555977549083028217284953868425189944782
            25208287785427826266153311574748096737981752260652732675754
            24891334268200609354995752235840657510900157354216769991450
            08147555771409008130451113932601682234782006419441548442926
            63737092586685507183627770985061231585045643636266669993334
            40139466855495571754209658880623405580275371529517831532932
            22302199784060472701645370398274690922242108750777206411131
            13343915986575402082165491312670042574380485586788891387415
            53958572750458055219420260828367451370760080678595928252256
            64179900246170273659285795738065271914865782904333906031177
            70203387698844541654074113994318381551031299751865000164318
            88532344712691926338633208743182412959495658513719924678709
            97451146655754268905549434601930247921212078151140658777821
            21526884588239248166212678825159729763398885864666994249198
            86181537563809867375268900553658471623544234047965668237624
            14167062062297627346560365999071034663278109
          Logging string written to file is:
          Call to real_function{'message': 'Operation m to the power of n', 'm':
            123456789, 'n': 199}
          was made at Mon Mar 13 12:07:27 2023, taking 0.0
```

As shown above, the log written to the calllog.txt file is as follows:

```
Call to real_function{'message': 'Operation m to the power of n', 'm':
123456789, 'n': 199}
was made at Mon Mar 13 12:07:27 2023, taking 0.0
```

Our next example is to use the calllog function as a decorator, as defined above, to record the time it takes to find all perfect numbers in a given range, the example we have worked on in Chapter 4.

This time, since we have learned how to define and use functions, we will define functions for factorization and perfect number testing, respectively, so that we can log the time on calls to the function. The code is as follows:

https://doi.org/10.15215/remix/9781998944088.01

In [ ]:
```
def factors(n): # function for finding factors of a given
 number
   # the function will return a list of factors for n
   factor_list = [1] # make a list with 1 as a single element
   for f in range(2,n): # start from 2, with n as excluded
 from factors
       if n%f == 0: # f is a factor of n
           if not f in factor_list:
               factor_list.append(f)
   # now we have a list of factors for n
   return factor_list


# function to find all perfect numbers in a given range
def perfect_numbers(a, b):
    if a>b: # then we need to swap a and b
        c = a; a = b; b = c
    perfect_list = [] # make an empty list ready to hold all
 perfect numbers
    for n in range(a, b+1): # b is included
        factor_list = factors(n)
        if n == sum(factor_list):
            perfect_list.append([n, factor_list]) # keep
 factors too for checking
    return perfect_list


# now the main
@calllog   # calllog is used as a decorator
def do_perfect():
    success = False
    # use a while loop to keep asking for inputs until success
 is True
    while not success:
        num1 = input("Enter the first number: ")
        num2 = input("Enter the second number: ")


        # try to convert the inputs to floats and divide them
            try:
            a, b = int(num1), int(num2)
            # set success to True if no error has occurred by
 now
            success = True
            perfect_list = perfect_numbers(a, b)
        # now we have found all the perfect numbers in the
 range
            print(f"Perfect numbers found between {a} and
 {b}:")
            for n in perfect_list:
                print(n, end=" ")
        # handle the possible errors and exceptions
        except ValueError:
            print("Invalid input. Please enter numbers only.")
        except ZeroDivisionError:
            print("Cannot divide by zero. Please enter a
 nonzero number.")
        except Exception as e:
            print(f"An unexpected error occurred: {e}")
    # end of do_perfect


do_perfect()
```

https://doi.org/10.15215/remix/9781998944088.01

```
Out[]: Perfect numbers found between 3 and 10000:
       [6, [1, 2, 3]]
       [28, [1, 2, 4, 7, 14]]
       [496, [1, 2, 4, 8, 16, 31, 62, 124, 248]]
       [8128, [1, 2, 4, 8, 16, 32, 64, 127, 254, 508, 1016,
        2032, 4064]] Logging string written to file is:
       Call to do_perfect
       was made at Mon Mar 13 13:18:15 2023, taking
        5.2627036571502686
```

As shown above, the log written to the calllog.txt is

```
Call to do_perfect
was made at Mon Mar 13 13:18:15 2023, taking 5.2627036571502686
```

As you can see, although a decorated function is called in the same way as other functions, a lot of other things can be done through a decorator during the call.

It is also worth noting that perfect numbers are very rare. Between 3 and 10000, there are only four perfect numbers, and it took a little over 5 seconds to find them. Tested on the same machine, it took more than 100 second to look for all the perfect numbers between 3 and 100000, and there are no perfect numbers between 10000 and 100000.

6.9 Properties of Functions

Functions are important building blocks of programs in all programming languages. In Python, a function will have the following properties, and most of them can be written by the programmer of the function.

__DOC__

This holds the function's docstring, written by the programmer, or None if no docstring is written by the coder. The docstring will show up when the help() function is called on a function, though help() will also show some standard information even if no docstring is available.

__NAME__

This contains the name of the function.

__QUALNAME__

This contains the qualified name of the function and is new to Python 3.3 and later versions. By *qualified name*, we mean the name that includes the path

https://doi.org/10.15215/remix/9781998944088.01

leading to the location where the function is defined. For example, a function F can be a method defined in a class C, which is defined in a module M, in which case the qualified name will be M.C.F.

__MODULE__

This stores the name of the module in which the function was defined. It contains None if module info is unavailable.

__DEFAULTS__

This stores a tuple containing default argument values for those arguments that have defaults or stores None if no arguments have a default value.

__CODE__

This contains the actual code object representing the compiled function body.

__GLOBALS__

This contains a reference to the dictionary that holds the function's global variables, the global namespace of the module in which the function was defined. It is a read-only function automatically generated by Python Virtual Machine (PVM).

__DICT__

This stores a dictionary describing names of attributes and their values in the namespace of the function. In Python, a namespace is a mapping from names to objects and is implemented as a Python dictionary.

__CLOSURE__

This stores a tuple of cells that contain bindings for the function's free variables. Each cell has an attribute called cell_contents from which a cell's value can be retrieved. It is automatically generated by PVM.

__ANNOTATIONS__

This stores a dictionary containing annotations of parameters. The keys of the dictionary are the parameter names, and ***return*** as key for the return annotation, if provided, and the values of the dictionary are the expected data type of the parameters as well as the expected data type of the object to be returned by the function.

__KWDEFAULTS__

This stores a dictionary containing default values for keyword-only parameters.

https://doi.org/10.15215/remix/9781998944088.01

Chapter Summary

- Functions are important building blocks of programs in almost all programming languages.
- Different languages use different keywords to signify the definition of a function. Python uses ***def*** to start the definition of a function (and method within a class definition).
- In Python, the definition of one function can contain the definition of other function or functions.
- A function can have positional arguments, variable-length lists of arguments, and keyword arguments.
- When calling a function, the parameters for positional arguments must come first, followed by variable-length lists of arguments. Keyword arguments come last.
- A recursive function is a function that calls itself within its definition.
- Anonymous functions are functions without a name. They begin with the keyword ***lambda***. Anonymous functions are useful when the function is only used once.
- Mapping, filtering, and reducing are special functions that can used to apply a function, including an anonymous function, to a list of parameters.
- Mapping applies a function to each item of a list.
- Filtering applies some criteria specified in a Boolean function to a list to filter out the items that do not meet the criteria.
- Reducing sequentially applies a function to the members of a list and reduces the list to a single member.
- Functions are often used to process and return the results of information processing using the ***return*** statement.
- In a function definition, the ***return*** statement can be replaced with the ***yield*** statement to turn the function into a generator of a sequence. Note that a function cannot be turned into a generator by simply replacing the ***return*** statement with a ***yield*** statement.

Exercises

1. Python has a built-in input function for taking input from users. However, it treats everything from the user as a string. For this exercise, define a function named getInteger, which takes one optional argument as a prompt and gets and returns an integer from the user.

https://doi.org/10.15215/remix/9781998944088.01

2. Define a function that has one argument, n, that will take a natural number and return the product of all the odd numbers between 1 and n.
3. Define a function that has one argument, n, that will take an integer and return the product of all the odd numbers between 1 and n, or between n and −1 if n is a negative integer, or that will return None if n is not an integer.
4. The Fibonacci sequence (F_n) is well-known in mathematics, and is defined as follows:

   $$F_0 = 1, F_1 = 1, F_n = F_{n-1} + F_{n-2}$$

   Define a recursive function that takes one argument, n, and calculates and returns the nth item (F_n) of the Fibonacci sequence.
5. Define a recursive function that takes one argument, n, and calculates and returns the entire list of all items from F_0 to $F_{n,}$ of the Fibonacci sequence.
6. Define a function that takes a variable-length list of numbers and returns the product of all these numbers.

Projects

1. Study the federal personal income tax rates for the current tax year and define a function that takes one argument as net taxable income and calculates and returns the total federal income tax due.
2. Study both the federal personal income tax rates and the provincial tax rates for your province for the current tax year. Define a function that takes one argument as net taxable income and calculate and return both total federal income tax due and total provincial tax due.
3. Modify the function defined for 6.6 by adding a keyword argument with the default value f or F to tell the function to calculate and return the federal tax due. If the argument passed to the keyword parameter is p or P, then provincial tax due should be calculated and returned. Test the modified function with different values passed to the keyword argument to make sure it works as expected.
4. The Tower of Hanoi is a mental game. It consists of three rods and N disks of different sizes that can move onto any rod one by one, but at no time is a bigger disk allowed to be on top of a smaller disk. The three rods are labelled A, B, and C. The game begins with all disks stack on rod A, and the goal is to move all the disks onto rod C. Write a recursive function that takes one argument as the number of disks initially on rod A and print out the moves to be taken in order to move

https://doi.org/10.15215/remix/9781998944088.01

all the disks from A to C. Each move can be represented as i: Rs Rd, where i is 0, 1, 2..., Rs and Rd is A, B or C, but Rs and Rd cannot be the same. It means that at step i, take one disk from source rod Rs and slide it onto destination rod Rd. For example, if there are three disks on rod A, the moves will be as follows:

1: A C, 2 : A B, 3 : C B, 4 : A C, 5 : B A, 6 : B C, 7 : A C

https://doi.org/10.15215/remix/9781998944088.01

Chapter 7

Object-Oriented Programming with Python

This chapter introduces you to object-oriented programming, including how to define classes and how to use classes as a new data type to create new objects, and will show you how to use objects in programming. *Programming for computers* essentially means modelling the world, or parts of it, for computers. In modern computing and programming, object-oriented programming is a very powerful means of thinking and modelling. In Python, everything can be treated as an object.

Learning Objectives

After completing this chapter, you should be able to

- explain object-oriented programming and list its advantages.
- define a new class to model objects in programming and software development.
- create and use objects and instances of classes in programming and system development.
- use subclasses and superclasses properly when defining and using classes.
- properly define and use public, private, and protected members of a class.
- correctly define and use class attributes.
- effectively define and use class methods and static methods when doing object-oriented programming.
- properly define and use dunder methods.
- use class as a decorator.

https://doi.org/10.15215/remix/9781998944088.01

- explain the built-in property function and use it to add an attribute or property to a class with or without explicit setter, getter, or deleter functions.
- use the property function as a decorator to turn a method into an attribute of a class, and declare explicitly the setter, getter, or deleter of the attribute.

7.1 Introduction to Object-Oriented Programming (OOP)

Object-oriented programming, including analysis and design, is a powerful methodology of thinking of how things are composed and work. The world is made of objects, each of which has certain attributes and contains smaller objects. What is specially offered by object-oriented analysis, design, and programming in a systematic and even scientific manner are abstraction, information hiding, and inheritance.

Abstraction

Abstraction is a very fundamental concept of object-oriented programming. The concept is rather simple. Because an object in the real word can be very complicated, containing many parts and with many attributes, it would be practical to consider only those parts and attributes that are relevant to the programming tasks at hand. This simplified model of a real-world object is an abstraction of it.

Information Hiding or Data Encapsulation

The concept of information hiding is very simple and straightforward. There are two reasons for hiding certain kinds of information: one is to protect the information, and the other is to make things easier and safer by hiding the details. An example of information hiding that you have already seen is defining and using functions. The code block of a function can be very lengthy and hard to understand. After a function is defined, however, a programmer only needs to know what the function does and how to use it, without considering the lengthy code block of how the function works.

In OOP, information hiding is further down within classes. Some OOP languages strictly limit direct access to class members (variables declared within a class), and all access must be done through the setter and getter methods. Python, however, has no such restriction, but you still need to remember the benefit of information hiding: avoiding direct access to the internal members of objects. Python also provides a way to hide, if you want to, by using double underscored names, such as __init__, __str__, and __repr__.

https://doi.org/10.15215/remix/9781998944088.01

Inheritance

Inheritance is a very important concept of object-oriented programming, and inheriting is an important mechanism in problem solving and system development with the object-oriented approach. The underlying philosophy of inheritance is how programmers describe and understand the world. Most often, things are categorized and put in a tree-like hierarchy with the root on the top, as shown in Figure 7-1 below. In such a tree-like hierarchy, the root of the tree is the most generic class or concept, and the leaves are the most specific and often refer to specific objects. From the root down to the leaves, nodes on a lower level will inherit the properties of all the connected nodes at higher levels.

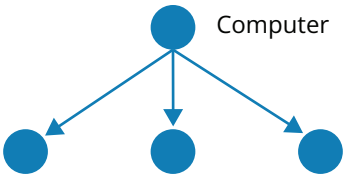


Figure 7-1: Illustration of class inheritance

Within such a hierarchy, it is more convenient to have a model that captures general attributes shared by desktop, laptop, and tablet computers than to have models capturing the specifics of desktop, laptop, and tablet computers, respectively, with these specific models inheriting the common attributes captured by the generic model for computers. In such a hierarchy, the computer is the superclass of the desktop, laptop, and tablet, whereas the desktop, laptop, and tablet are a subclass of the computer.

7.2 Defining and Using Classes in Python

Normally in OOP, a class would include some attributes and methods, as well as a constructor or initiator for creating instances of the class. However, compared to other object-oriented programming languages, especially the earlier generations such as C++ and Java, Python provides some less-restricted ways of defining classes and instantiating objects.

In Python, a class can be easily created with only two lines of code, as shown below:

```
>>> class Computer:
…    pass
```

https://doi.org/10.15215/remix/9781998944088.01

This defines a class named Computer containing no attribute and no method, though it automatically inherits all the attributes and methods of the generic object class in the built-in module of Python. As mentioned before, the ***pass*** statement is simply a placeholder for everything needed to complete the class definition. We can use the ***help*** statement to see that the class has been created, as shown below:

```
>>> help(Computer)
Help on class Computer in module __main__:

class Computer(builtins.object)
| Data descriptors defined here:
|
| __dict__
|   dictionary for instance variables (if defined)
|
| __weakref__
|   list of weak references to the object (if defined)
```

The builtins.object is a built-in class of Python from which all classes automatically inherit by default. In OOP term, the class that inherits from another class is called a subclass of the other class, while the class being inherited from is called a superclass.

Formally in Python, if you are defining a class that needs to inherit from another class, you can put the superclass(es) in a pair of parentheses, as shown in the following example:

```
>>> class PC(Computer):
…   pass   # the pass statement does nothing, but it
completes the class definition
```

We can check the result using the ***help*** statement, as shown below:

```
>>> help(PC)
Help on class PC in module __main__:

class PC(Computer)
| Method resolution order:
|   PC
|   Computer
```

https://doi.org/10.15215/remix/9781998944088.01

```
|   builtins.object
|
| Data descriptors inherited from Computer:
|
| __dict__
|   dictionary for instance variables (if defined)
|
| __weakref__
|   list of weak references to the object (if defined)
```

This shows that the class PC has been created, which is a subclass of Computer.

Although the Computer class contains nothing in its original definition, we can create an instance of the class, add attributes to the instance, and manipulate the attributes using some built-in functions. For example, we can use the setattr function to add an attribute called CPU to an instance of the Computer class, as shown below:

```
>>> c = Computer()   # to create an instance of the
Computer class
>>> setattr(c, 'CPU', 'Intel i6800')
>>> c.CPU
'Intel i6800'
```

In the above example, Computer() is the constructor of class Computer. In Python, X() will be automatically the constructor of class X, but it calls a special method named __init__, which can be defined within the class to instantiate instances of the class. In our simplest definition of class Computer, we did not define the __init__ method, but it has been automatically inherited from class builtins.object.

Now we can use special attribute __dict__ of instance c of class Computer to check the attribute and value of object c, as shown below:

```
>>> print(c.__dict__)
{'CPU': 'intel i6800'}
```

As you can see, the c object has an attribute called CPU and its value intel i6800.

In addition to setattr, Python has a number of built-in functions available for class and object manipulation. These built-in functions are summarized in Table 7-1.

https://doi.org/10.15215/remix/9781998944088.01

Table 7-1: Built-in functions for class and object manipulation

Built-in function	Operation	Coding example

getattr(o, attr)	Return the value of object o's attribute attr, same as o.attr	`>>> getattr(math, 'sqrt')`
		`<built-in function sqrt>`
		`>>> getattr(math, 'e')`
		`2.718281828459045`

hasattr(o, attr)	Test if object o has attribute attr; return True if it does	`>>> hasattr(math, 'e')`
		`True`
		`>>> hasattr(math, 'sqrt')`
		`True`

setattr(o, a, v)	Set/add an attribute a to object o, and assign value v to the attribute	`>>> class student:`
		`…  pass`
		`…`
		`>>> s1 = student()`
		`>>> setattr(s1, 'name',`
		`'John')`
		`>>> s1.name`
		`'John'`

delattr(o, a)	Delete attribute a from object o	`>>> delattr(s1, 'name')`
		`>>> hasattr(s1, 'name')`
		`False`

isinstance(o, c)	Return true if o is an instance of class c or a subclass of c	`>>> class student:`
		`…  pass`
		`…`
		`>>> s1 = student()`
		`>>> isinstance(s1, student)`
		`True`

issubclass()	Return true if class c is a subclass of C	`>>> class`
		`graduate(student):`
		`…  pass`
		`…`
		`>>> issubclass(graduate,`
		`student)`
		`True`

repr(o)	Return string representation of object o	`>>> repr(graduate)`
		`"<class '__main__.`
		`graduate'>"`

The rest of this section describes in detail how to define classes in Python—in particular, how to define attributes and methods in a class definition. Your journey to learn OOP begins with modelling a small world, shapes, which include the circle, rectangle, and triangle.

https://doi.org/10.15215/remix/9781998944088.01

First, define an abstract class called shape, as shown in Table 7-2.

Table 7-2: Example of class definition with overriding

Code sample in Python interactive mode

```
1  """
2  Modelling the world of shapes. First Python app with
     OOP.
3  """
4
5  class Shape:
6    """We design an abstract class for shape."""
7    def circumference(self):
8      """Method to be overridden."""
9      pass
10
11 def area(self):
12 """Method to be overridden."""
13 pass
14
15 class Circle(Shape):
16   def __init__(self, radius):   # two underscores on
     each side
17     self.radius = radius
18
19   def circumference(self):   # this overrides the
     method defined in Shape
20     return self.radius * 2 * 3.14
21
22   def area(self):   # this overrides that defined in
     Shape
23     return self.radius ** 2 * 3.14
24
25 c1 = Circle(35)
26 print(f"A circle with a radius of {c1.radius} has an
     area of {c1.area()},")
27 print(f"and the circumference of the circle is {c1.
     circumference()}")
```

Output

```
A circle with a radius of 35 has an area of 3846.5,
and the circumference of the circle is 219.8
```

In the example above, __init__ is a reserved name for a special method in a class definition. It is called when creating new objects of the class. Please remember, however, that you need to use the name of the class when creating new instances of the class, as shown on line 25 of the example above.

It may have been noted that “self” appears in the list of arguments when defining the __init__ method and other methods of the class, but it is ignored in

https://doi.org/10.15215/remix/9781998944088.01

the calls of all these methods. There is no explanation as to why it is ignored. In the definitions of all these methods, "self" is used to refer to the instance of the class. It is the same as "this" in other OOP languages such as C++ and Java, though "this" doesn't appear in the list of formal arguments of any method definition.

Another detail worth noting when defining classes in Python is that there can be no explicit definition of any of the attributes, as is the case in C++ or Java. Instead, attributes are introduced within the definition of the __init__ method by assignment statements or by using the built-in function setattr(o, a, v), which are all the attributes of the particular instance created by the constructor of the class. Function setattr(o, a, v) sets the value of attribute a of object o to n, if o has attribute a; if not, it will add attribute a to o, then set its value to v.

Next, we define a class for a rectangle, as shown in Table 7-3.

Table 7-3: Example of subclass definition

Code sample in Python interactive mode

```
"""
Modelling the world of shapes. First Python app
  with OOP.
"""

class Shape:
  """We design an abstract class for shape."""
  def circumference(self):
    """Method to be overridden"""
    pass

  def area(self):
    """Method to be overridden"""
    pass

class Rectangular(Shape):
  def __init__(self, length, width):
    self._length = length
    self._width = width

  def circumference(self):
    return (self._length + self._width) * 2

  def area(self):
    return self._length * self._width

  def is_square(self):
```

https://doi.org/10.15215/remix/9781998944088.01

Table 7-3: Example of subclass definition *(continued)*

```
27      return self._width == self._length
28
29  rt1 = Rectangular(35, 56)
30
31  print(f"The circumference of the rectangle is
      {rt1.circumference()}, and")
32  print(f"the area is {rt1.area()}")
33  print(f"Is the rectangle a square? {rt1.
      is_square()}")
```

Output of the program:

```
The circumference of the rectangle is 182, and
the area is 1960
Is the rectangle a square? False
```

Note that class Rectangular not only has overridden two methods of Shape but has also defined a new method named is_square() to test if the rectangular is a square.

Inheritance: Subclass and Superclass

If you want to inherit from other base classes when defining a new class in Python, you need to add all the base classes to the list of inheritances enclosed in a pair of parentheses, as shown below:

```
class myClass(base_1, base_2, base_3):
  pass
```

The new class will inherit all the attributes and methods from base_1, base_2, and base_3 and override the methods defined in the base classes.

In Python, all classes are a subclass of the built-in base object class and inherit all the properties and methods of object, even if object is not explicitly included in the inheritance list. So the following two statements will have the same effects:

```
In [ ]:  class myClassA:
           pass

         dir(myClassA)

Out [ ]: ['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__format__',
          '__ge__', '__getattribute__', '__gt__', '__hash__', '__init__', '__init_subclass__',
          '__le__', '__lt__', '__module__', '__ne__', '__new__', '__reduce__', '__reduce_
          ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__',
          '__weakref__']
```

https://doi.org/10.15215/remix/9781998944088.01

```
In [ ]:   class myClassB(object):
            pass

          dir(myClassB)

Out [ ]: ['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__format__',
           '__ge__', '__getattribute__', '__gt__', '__hash__', '__init__', '__init_subclass__',
           '__le__', '__lt__', '__module__', '__ne__', '__new__', '__reduce__', '__reduce_
           ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__',
           '__weakref__']
```

As you can see, myClassA and myClassB both inherit from the base class object. However, in the list, some of the dunder names such as __le__ and __ge__ inherited from base class object are merely wrapper descriptors. If you want to make real use of these wrapper descriptors in your class, you will need to override them. Some inherited dunder methods can be used, but the values returned are not so useful. In particular, you will need to override the dunder methods __init__ __str__, and __repr__. The method __init__ is a dunder method used as constructor called internally by PVM whenever a new instance of a class needs to be created. The method __str__ is a dunder method called internally whenever an object of a class needs to be converted to a string, such as for printout. Finally, __repr__ is a dunder method called internally when an object needs to be converted to a representation for serialization—a process that converts a Python object into a byte stream that can be stored or transmitted.

Public, Private, and Protected Members of a Class

People familiar with other OO programming languages such as C++ and Java may be wondering how to declare and use public, private, and protected attributes in Python, as they are used to doing in other OO programming languages. Python, however, doesn't differentiate attributes between public, private, and protected members. Instead, Python treats all attributes of a class as public. It is up to the programmers to decide whether a member should be public, private, or protected. To ensure that everyone reading the code understands the intention of the original coder/programmer, Python has a convention for naming protected members, private members, and public members: a name with a leading underscore _ is a protected member of the class in which it is defined, a name with a double underscore __ is a private member of the class in which it is defined, and all other names will be public.

According to the common principles of object-oriented programming, public members of a class can be seen and accessed from outside of the class or the instance of the class; protected members of a class can only be seen

https://doi.org/10.15215/remix/9781998944088.01

and accessible within the class or a subclass of the class; private members can only be accessed within the class. In Python, however, the rule for protected members is not strictly enforced. As such, the following code will not raise any error or exception.

```
In [ ]:  class Student:
           def __init__(self, firstname, lastname):
             self._firstname = firstname
             self._lastname = lastname

         s0 = Student('Jim', 'Carte')
         print(s0._firstname, s0._lastname)
```

```
Out [ ]: Jim Carte
```

In our example about shapes, _width and _length are protected attributes of the class, which should only be accessible within the class or its subclasses and should be hidden from the outside.

In contrast, the rule on private members, those with names prefixed with a double underscore __, is strictly enforced in Python. So if we change firstname and lastname to private members, it will raise exceptions, as shown in the following example:

```
In [ ]:  class Student:
           def __init__(self, firstname, lastname):
             self.__firstname = firstname
             self.__lastname = lastname

         s0 = Student('Jim', 'Carte')
         setattr(Student, '_firstname', 'Richard')
         print(s0.__firstname, s0.__lastname)
```

```
Out [ ]: AttributeError Traceback (most recent call last)
         <ipython-input-5-87cd9bc7801e> in <module>
          6 s0 = Student('Jim', 'Carte')
          7 setattr(Student, '_firstname', 'Richard')
         ----> 8 print(s0.__firstname, s0.__lastname)

         AttributeError: 'Student' object has no attribute '__firstname'
```

If you want to access the private members of a class from outside, the built-in functions setattr and getattr can be used to access both private and protected members of a class or its instance, as shown below:

https://doi.org/10.15215/remix/9781998944088.01

In [ ]:
```
class Student:
  def __init__(self, firstname, lastname):
    self.__firstname = firstname
    self.__lastname = lastname


s0 = Student('Jim', 'Carte')
setattr(s0, '__firstname', 'Richard')
setattr(s0, '__lastname', 'Selot')
print(getattr(s0, '__firstname'), getattr(s0, '__lastname'))
```

Out [ ]:
```
Richard Selot
```

Class Methods

As we have seen, in Python, special method __init__ is used as a constructor of the class. People familiar with C++ or Java might be wondering if __init__ can be overloaded or defined multiple times in Python to make multiple constructors, because a class in C++ and Java can have two or more constructors with different signature. In Python, however, a class cannot have more than one __init__ effectively defined. If you define two or more __init__ within a class definition, only the last one will take effect.

So how can you create an instance of a class differently in Python if there can be only one constructor, the special __init__ method, in a class definition? The solution is to use the class method, as shown in the next example:

In [ ]:
```
class Graduate:
  def __init__(self, fullname):
    firstname, lastname = fullname.split(' ')
    self.firstname = firstname
    self.lastname = lastname


@classmethod # this decorator declaires a class method
def newGraduate(cls, firstname, lastname): # cls as 1st
    return cls(f'{firstname} {lastname}') # return an
 object


def __str__(self): # normal method
    return f'{self.firstname} {self.lastname}'


g0 = Graduate('James Gord') # __init__ is called to
 construct a new object
g1 = Graduate.newGraduate('John', 'Doe')
# newGraduate() is called to create a new object


print(g0) # __str__ is called to convert the object to
 a string
print(g1) # __str__ is called to convert the object to
 a string
```

Out [ ]:
```
James Gord
John Doe
```

https://doi.org/10.15215/remix/9781998944088.01

In the example above, decorator @classmethod is used to declare that method newGraduate() is a class method, which means that the first parameter refers to the class instead of an instance of the class, and the method is bound to the class and can be called through the class. A class method can be used to directly modify the structure of the class. For example, we can have a class method for Graduate class that sets the value of the class attribute at the beginning of a new year, say, 20210101. In this particular example above, however, class method newGraduate() is used as an alternate constructor of class Graduate, which takes first name and last name separately to instantiate an instance of Graduate.

When defining a class method, it is a convention to use cls as the name of the first parameter to refer to the class. Technically, however, it can be anything unique in the scope of a class definition, as long as you know it refers to the class. This is similar to "self" in definitions of regular methods, which is only the conventional name referring to the instance itself being instantiated.

Static Methods

Similar to the class method of a class, a static method can be called directly through the class. The difference is that in the definition of a static method, no parameter refers to the class nor to the instance itself. Its usefulness may be demonstrated by the example below, where we define a class called Convertor, and within the class definition, we define a number of static methods, each of which convert values from one unit to another:

In [ ]:
```
class Convertor:

  @staticmethod
  def kg2lb(w):
    return w * 2.20462

  @staticmethod
  def lb2kb(w):
    return w/2.20462

  @staticmethod
  def metre2feet(l):
    return l * 3.28084

  @staticmethod
  def feet2metre(l):
    return l / 3.28084
```

https://doi.org/10.15215/remix/9781998944088.01

```
    @staticmethod
    def C2F(d):
      return (d * 9 / 5) + 32

    @staticmethod
    def F2C(d):
      return (d - 32) * 5 / 9

    @staticmethod
    def acr2hec(a):
      return a * 0.404686

    @staticmethod
    def hec2arc(a):
      return a / 0.404686

print(Convertor.kg2lb(123))
print(Convertor.C2F(21))
```

Out [ ]:
```
271.16826
69.8
```

As you can see from the above example, the static methods can be called directly through the class Convertor, without an instance. Defining a static method within a class is a way to add utility functions to the class so that it can be used without instantiating an object.

Class Attributes

As previously mentioned, in Python, you can define a class without explicitly declaring attributes of the class. However, Python does allow the explicit declaration of attributes within a class definition. These attributes are called class attributes. Explicit declaration of an attribute within a class definition means that the attribute is declared outside of the __init__ method. In the following example, student_id is declared as a class attribute.

In [ ]:
```
class Graduate:
  student_id = 20201195 # student_id is a class
 attribute
  def __init__(self, fullname):
# self refers to the instance in normal method
    firstname, lastname = fullname.split(' ')
    self.f_name = firstname
    self.l_name = lastname
    self.__class__.student_id += 1
# self.__class__.student_id must be used to
# refer class attribute
```

https://doi.org/10.15215/remix/9781998944088.01

```
  @classmethod   # decorator
  def newGraduate(cls, firstname, lastname):
# cls refers to the class
    return cls(f'{firstname} {lastname}')

  def __str__(self):
    return f'{self.f_name} {self.l_name}, {self.
 student_id}'

g0 = Graduate('James Gord')
print(g0)
# newGraduate() is called from class Graduate
g1 = Graduate.newGraduate('John', 'Doodle')
print(g1)
```

Out [ ]:
```
James Gord, 20201196
John Doodle, 20201197
```

From this example, you can see how a class attribute is used and shared among all the instances of the class. You can create student_id as a class attribute to dynamically track the student IDs allocated to new students—a new instance of the class Graduate. It starts from 20201195 and increases by one every time a new student is enrolled.

Note the dunder name __class__ used in the example. In Python, __class__ refers to the class of an object. So in the example above, self.__class__ refers to the class of object self—that is, Graduate. If you do not have __class__ between self and student_id but simply use self.student_id, student_id would become an attribute of each instance of the class, different from the class attribute, as demonstrated in the following example:

In [ ]:
```
class Graduate:
  student_id = 20201195
  def __init__(self, fullname):
    firstname, lastname = fullname.split(' ')
    self.f_name = firstname
    self.l_name = lastname
    self.student_id += 1
# self.student_id is an attribute of an individual object

  def __str__(self):   # self refers to the instance/
 object
    return f'{self.f_name} {self.l_name}, {self.
 student_id}'

g0 = Graduate('James Gord')
print(g0)
print(f'class attr student_id = {g0.__class__.student_id}')
g1 = Graduate('John Doodle')
print(g1)
```

Out [ ]:
```
James Gord, 20201196
class attribute student_id = 20201195
John Doodle, 20201196
```

https://doi.org/10.15215/remix/9781998944088.01

As you can see from the example above, although value 20201195 of class attribute student_id is used to calculate the student_id (20201195 + 1) when instantiating each instance of the class, the result 20201196 has no effect on the class attribute whose value remains to be 20201195. How did that work? Remember that assignment self.student_id += 1 is short for self.student_id = self.student_id. In this example, self.student_id on the right side of the assignment statement is resolved to be the class attribute according to the rule, whereas self.student_id on the left side of the assignment statement is an attribute of the object being instantiated. That is, the two self.student_id are two different variables.

Class attributes have different behaviours from static attributes in C++ or Java, although they do have certain things in common. In Python, class attributes are shared among all the objects of the class and can also be accessed directly from the class, without an instance of the class, as shown above.

7.3 Advanced Topics in OOP with Python

The previous section showed how classes can be defined and used in problem solving and system development. With what you learned in the previous section, you can certainly solve some problems in an object-oriented fashion. To be a good OOP programmer and system developer, it is necessary to learn at least some of the advanced features offered in Python for object-oriented programming.

Dunder Methods in Class Definition

Python has a list of names reserved for some special methods called dunder methods or magic methods. These special names have two prefix and suffix underscores in the name. The word dunder is short for "double under (underscores)." Most of these dunder methods are used for operator overloading. Examples of the dunder/magic methods that can be used for overloading operators are __add__ for addition, __sub__ for subtraction, __mul__ for multiplication, and __div__ for division.

Some dunder methods have specific meanings and effects when overloaded (defined in a user-defined class). These dunder methods include __init__, __new__, __call__, __len__, __repr__, __str__, and __getitem__.

In the previous section you saw how __init__ method is used as a constructor of class. The following sheds some light on the others.

https://doi.org/10.15215/remix/9781998944088.01

__CALL__

Dunder method __call__ can be used to make an object, an instance of class, callable, like a function. In the following example, a class is defined as Home, which has two attributes: size and value. In addition to __init__ as a constructor, a dunder function __call__ is defined, which turns object h1, an instance of class Home, into a callable function.

```
In [ ]:  class Home():
           def __init__(self, size, value):
             self.size = size
             self.value = value

           def __call__(self, check='average'):   # return the
         size, or value, or average
             if check == 'size':
               return self.size
             elif check == 'value':
               return self.value
             else:
               return self.value/self.size

         h1 = Home(3270, 986500)
         print(h1())   # using default value for the keyword
          argument
         print(h1(check='size'))   # check the size of the home
         print(h1(check='value'))   # check the value of the
          home
```

```
Out [ ]: 301.68195718654437
         3270
         986500
```

What can be done through dunder method __call__ can also be done through a normal method, say checking. The obvious benefit of using the dunder method __call__ is cleaner and neater code.

__NEW__

Dunder method __new__ is a method already defined in the object base class. It can be overridden to do something extra before the creation of a new instance of a class. Dunder method __ new__ itself doesn't create or initialize an object for a class. Instead, it can check, for example, if a new object can be created. If the answer is yes, then it will call __init__ to do the actual work of creation and initialization, as shown in the following example:

https://doi.org/10.15215/remix/9781998944088.01

In [ ]:
```
class TravelPlan():
  _places = dict()
  _step = 0
  def __new__(cls, newPlace):
    if newPlace in TravelPlan._places.values():
      print(f"{newPlace} is already in the plan!")
      return newPlace
    else:
      return super(TravelPlan, cls).__new__(cls)

  def __init__(self, newPlace):
    TravelPlan._places[TravelPlan._step] = newPlace
    print(f'{TravelPlan._places[TravelPlan._step]} is
added.')
    TravelPlan._step += 1

  @staticmethod
  def printPlan():
    for pl in TravelPlan._places:
      print(f'Stop {pl + 1}: {TravelPlan._places[pl]}')

TravelPlan('Calgary')
TravelPlan('Toronto')
TravelPlan('Calgary')
TravelPlan.printPlan()
```

Out [ ]:
```
Calgary is added.
Toronto is added.
Calgary is already in the plan!
Stop 1: Calgary
Stop 2: Toronto
```

The example above makes a travel planning system by making a list of places to be visited. The __new__ method is defined to control the addition of a new city to the list. It checks whether the city has already been added to the plan and will not add the city if it is already on the list.

Note that we also defined and used two protected class attributes in the definition of class TravelPlan. One is a dictionary storing the places in the plan and their order. We also used a static method so we can print out the entire plan when it is needed. Because the class attributes are shared among the instances of the class, changes to the class attributes will be retained, and at any time before the application is stopped, one can use the static method to print out the plan.

__STR__

Dunder method __str__can be used to implement the string representation of objects for a class so that it can be printed out by using the ***print*** statement. The method will be invoked when str() function is called to convert an object

https://doi.org/10.15215/remix/9781998944088.01

of the class to a string. In our definition of class Graduate in our earlier example, the __str__ has been implemented to just return the full name of a student. You may, of course, choose to return whatever string you think would better represent the object for a given purpose.

__LEN__

Dunder method __len__ can be used to return the length of an object for a class. It is invoked when function len(o) is called on object o of the class. It is up to the programmer to decide how to measure the length, though. In our definition of class Graduate, we simply use the sum of the first name, last name, and id length.

__REPR__

Dunder method __repr__ can be used to return the object representation of a class instance so that, for example, the object can be saved to and retrieved from a file or database. An object representation can be in the form of a list, tuple, or dictionary, but it has to be returned as a string in order to be written to and read from a file or database. The __repr__ method will be invoked when function repr() is called on an object of the class. The following example extended from the definition of class Graduate shows how dunder methods __str__, __len__, and __repr__ can be defined and used.

In [ ]:
```
class Graduate:
  student_id = 20201195
  def __init__(self, fullname):
    firstname, lastname = fullname.split(' ')
    self.firstname = firstname
    self.lastname = lastname
    self.__class__.student_id += 1


  @classmethod
  def newGraduate(cl, firstname, lastname):
    return cl(f'{firstname} {lastname}')


  def __str__(self):
    return f'{self.firstname} {self.lastname}'


  def __len__(self):
    return len(self.firstname) + len(self.lastname) +
 len(str(self.student_id))


  def __repr__(self):
    return "{'firstname':" + self.firstname +",
 'lastname':" + self.lastname + ", 'student_id':" +
 str(self.student_id) + "}"

g0 = Graduate('James Gord'); print(g0, g0.student_id)
g1 = Graduate.newGraduate('John', 'Doodle')   #
  newGraduate() is called from class Graduate
print(g1, g1.student_id); print(len(g0));
  print(str(repr(g1)))
```

https://doi.org/10.15215/remix/9781998944088.01

```
Out [ ]:  James Gord 20201196
          John Doodle 20201197
          17
          {'firstname':John, 'lastname':Doodle, 'student_id':20201197}
```

__GETITEM__ AND __SETITEM__

These two methods are used to turn a class into an indexable container object. __getitem__ is called to implement the evaluation of self[key] whereas __setitem__ is called to implement the assignment to self[key].

__DELITEM__

This is called to implement the deletion of self[key].

__MISSING__

This is called by dict.__getitem__() to implement self[key] for dict subclasses when the key is not in the dictionary.

__ITER__

This is called when an iterator is required for a container.

__REVERSED__

This can be implemented and called by the reversed() built-in function to implement a reverse iteration.

__CONTAIN__

This is called to implement membership test operators. It should return True if the item is in self and False if it is not.

__DELETE__

This is called to delete the attribute on an instance of the owner class.

Tables 7-4, 7-5, 7-6, and 7-7 show a comprehensive list of dunder methods and their respective operators that can be overridden by programmers when defining new classes.

Table 7-4: Binary operators

Overridden operator	Dunder method
+	object.__add__(self, other)
-	object.__sub__(self, other)
*	object.__mul__(self, other)

https://doi.org/10.15215/remix/9781998944088.01

Table 7-4: Binary operators *(continued)*

Overridden operator	Dunder method
//	object.__floordiv__(self, other)
/	object.__truediv__(self, other)
%	object.__mod__(self, other)
**	object.__pow__(self, other[, modulo])
<<	object.__lshift__(self, other)
>>	object.__rshift__(self, other)
&	object.__and__(self, other)
^	object.__xor__(self, other)
|	object.__or__(self, other)

Table 7-5: Augmented assignment operators

Overridden operator	Dunder method
+=	object.__iadd__(self, other)
-=	object.__isub__(self, other)
*=	object.__imul__(self, other)
/=	object.__idiv__(self, other)
//=	object.__ifloordiv__(self, other)
%=	object.__imod__(self, other)
**=	object.__ipow__(self, other[, modulo])
<<=	object.__ilshift__(self, other)
>>=	object.__irshift__(self, other)
&=	object.__iand__(self, other)
^=	object.__ixor__(self, other)
|=	object.__ior__(self, other)

https://doi.org/10.15215/remix/9781998944088.01

Table 7-6: Unary operators

Overridden operator	Dunder method
-	object.__neg__(self)
+	object.__pos__(self)
abs()	object.__abs__(self)
~	object.__invert__(self)
complex()	object.__complex__(self)
int()	object.__int__(self)
long()	object.__long__(self)
float()	object.__float__(self)
oct()	object.__oct__(self)
hex()	object.__hex__(self)

Table 7-7: Comparison operators

Operator	Dunder method
<	object.__lt__(self, other)
<=	object.__le__(self, other)
==	object.__eq__(self, other)
!=	object.__ne__(self, other)
>=	object.__ge__(self, other)
>	object.__gt__(self, other)

Using Class as Decorator

In Chapter 6, we saw how decorators can be used as powerful and useful tools that wrap one function with another to change the behaviour of the wrapped function without modifying the wrapped function. In this section, we show how to use classes as decorators to modify the behaviour of an existing function.

As we know, a class has two types of members. One is attributes and the other is methods. Our first example uses the methods of a class as a decorator, as shown below:

https://doi.org/10.15215/remix/9781998944088.01

In [ ]:
```
class MyClass:
  def __init__(self, decorated):
    self.func = decorated

  def __call__(self, *args, **kwargs):   # variable-
 length arguments
    print("More code can be added before calling the
 decorated function")
    self.func(*args, **kwargs)   # call the decorated
 function
    print("More code can be added after calling the
 decorated function")

# use MyClass as a decorator
@MyClass
def myFunc(name, message ='Welcome to COMP218'):
  print(f"Hello {name}, {message}")

myFunc("Joe", "Welcome to the world of Python")
```

Out [ ]:
```
More code can be added before calling the decorated function
Hello Joe, Welcome to the world of Python
More code can be added after calling the decorated function
```

As shown in the example above, the __init__ method is used to pass a function to be decorated or wrapped to the class, and the __call__ method is used to actually wrap the real function passed to the class.

We learned in Chapter 6 how to calculate the actual execution time of a program using a function as a decorator. The same can be done with a class as a decorator when we time finding all the primes within a given range using an algorithm called "sieve of Eratosthenes."

In [ ]:
```
from time import time
class ProgramTimer:
     def __init__(self, func):
          self.function = func
     def __call__(self, *args, **kwargs):
          program_start_time = time()
          result = self.function(*args, **kwargs)
          program_end_time = time()
          print(f"Execution of {self.function} took
            {program_end_time- program_start_time} seconds"
          return result
```

https://doi.org/10.15215/remix/9781998944088.01

```
# add a decorator to the class

def sieving(n, l):
  """Remove all n's multiples from list l."""
  return list(filter(lambda m: m == n or m%n != 0, l))

def i_list(max_l_int):
  """Create initial list."""
  l0 = [2] + list(range(3, max_l_int, 2))
  return l0

@ProgramTimer
def primesBySieving(upper_bound):
  nl = i_list(upper_bound)
  sl = nl[1: len(nl) // 2 + 1]   # initial sieve
  flag = True
  while flag:
    d = sl[0]
    nl = sieving(d, nl)
    sl = sl[1:]
    if d**2 > nl[-1]:
      flag = False
      return nl

ub = 2**19
pl = primesBySieving(ub)
print(f"{len(pl)} prime numbers have been found between
  2 and {ub}")
```

Out [ ]:
```
Execution of <function primesBySieving at 0x0000014363F8AEE8> took
  3.1760756969451904 seconds
43390 prime numbers have been found between 2 and 524288
```

In the above examples, we used class to decorate a function. Within the class, a special dunder function __call__ is used to execute the decorated function and calculate the time spent on the execution of the function that finds all the prime numbers between 2 and 32768 (2 ** 15).

Built-In Property() Function and Property Decorator

Python has a special built-in function named property, as mentioned in Chapter 2. It can be called directly to create an attribute of a class with potentially added setter, getter, and deleter methods and documentation. In this case, the function can either be called within a class definition or outside a class definition while adding the created property to a class or object of a class.

A call of the property function may take none or any combination of the four optional keyword arguments, as shown below:

https://doi.org/10.15215/remix/9781998944088.01

```
Property_name = property(fget=None, fset=None, fdel=None,
doc=None)
# fget, fset, and fdel take functions only
firstname = property()   # call of the property function
with default
lastname = property(set_lastname, get_lastname,
delete_lastname)
# call with three function names
firstname.setter(set_firstname)   # add a set function to
property firstname. set_firstname must be a function
firstname.getter(get_firstname)
# add a get function to property firstname. get_firstname
must be a function
```

If property() function is called with no argument, you can add a setter function, getter function, and/or deleter function later by calling the setter, getter. and deleter methods.

The following example demonstrates how the built-in property function is used to add a property/attribute to a class:

```
In [ ]:  class Employee:
           def __init__(self, firstname, lastname):
             self.firstname = firstname
             self.lastname = lastname

           def setFullname(self, fullname):
             names = fullname.split()
             self.firstname = names[0]
             self.lastname = names[1]

           def getFullname(self):
             return f'{self.firstname} {self.lastname}'

           fullname = property(getFullname, setFullname)

         e1 = Employee('Jack', 'Smith')
         print(e1.fullname)
         e1.fullname = 'John Doe'
         print(e1.fullname)
         print(f'{e1.lastname}, {e1.firstname}')
```

```
Out [ ]: Jack Smith
         John Doe
         Doe, John
```

https://doi.org/10.15215/remix/9781998944088.01

A built-in property function can also be used as a decorator within a class definition to make a method name to be used as an attribute/property of the class.

Similar to the example above, suppose we want to define a class named Student for a management system at a university. The attributes should include the first name, last name, full name, email address, and some other information. We know that a full name is made of a first name and a last name and that the university often has a rule of assigning an email address based on the first name and last name. If we define full name and email address as ordinary attributes, the dependencies would not be reflected because a change to the first name or last name of a student will not automatically result in the change to the full name and email address. Using property() function as a decorator can nicely solve the problem, as shown in the following example:

In [ ]:
```
class Student:
  def __init__(self, firstname, lastname):
    self.firstname = firstname
    self.lastname = lastname

  @property
  def fullname(self):
    return f"{self.firstname} {self.lastname}"

  @property
  def emailaddress(self):
    return f"{self.firstname}.{self.lastname}@
 globalemail.com"

s0 = Student('John', 'Doe')
print(f'First name: {s0.firstname}')
print(f'Last name: {s0.lastname}')
print(f'Full name: {s0.fullname}')
print(f'Email address: {s0.emailaddress}')
s0.lastname = 'Smith'
print(f'First name: {s0.firstname}')
print(f'Last name: {s0.lastname}')
print(f'Full name: {s0.fullname}')
print(f'Email address: {s0.emailaddress}')
```

Out [ ]:
```
First name: John
Last name: Doe
Full name: John Doe
Email address:John.Doe@globalemail.com

First name: John
Last name: Smith
Full name: John Smith
Email address:John.Smith@globalemail.com
```

https://doi.org/10.15215/remix/9781998944088.01

In the example above, we use built-in property() function as a decorator to decorate method fullname() and emailaddress(). By doing that, the function name can be used as a property or attribute of the object, but access to the attribute invokes a call to the function so that an updated full name or email address is retrieved. In the example, when the last name is changed to Smith, the full name is automatically changed to John Smith, and the email address is automatically changed to John.Smith@globalemail.com.

Using decorator, we can also further add setter, getter, and deleter methods to the property fullname, as shown below:

In [ ]:
```
class Student:
  def __init__(self, firstname, lastname):
    self.firstname = firstname
    self.lastname = lastname

  @property
  def fullname(self):
    return f"{self.firstname} {self.lastname}"

  @fullname.setter
  def fullname(self, fullname):
    """
    Set the value of fullname. However,
    you cannot do self.fullname = fullname
    because fullname is not true attribute.
    """
    names = fullname.split()
    self.firstname = names[0]
    self.lastname = names[1]

s0 = Student('John', 'Doe')
print(s0.fullname)
s0.fullname = 'Kevin Smith'
print(s0.fullname)
```

Out [ ]:
```
John Doe
Kevin Smith
```

Note that using a property decorator can make a method to be used like an attribute/property without the need to allocate memory space to keep the value of the attribute. As such, you cannot assign a value directly to such an attribute.

https://doi.org/10.15215/remix/9781998944088.01

Creating a New Class Dynamically and Modify a Defined Class or Instance

In Python, you have the freedom to make changes to an already defined class or an instance of the class, and you can even create a new class dynamically.

To create a new class dynamically in your Python program, a built-in function type is used, as shown below:

```
In [ ]:  Shape = type('Shape', (object,), dict(points=[(1,1)]))
         s0=Shape()
         s1=Shape()
         print(f's0.points = {s0.points}')
         print(f's1.points = {s1.points}')
         s0.points += [(12, 23)]   # add a new point to s0
         print(f's0.points = {s0.points}')
         print(f's1.points = {s1.points}')
         s1.points = [(35, 67)]   # this adds an attribute point
          to s1
         print(f's0.points = {s0.points}')
         print(f's1.points = {s1.points}')

Out [ ]: s0.points = [(1, 1)]
         s1.points = [(1, 1)]
         s0.points = [(1, 1), (12, 23)]
         s1.points = [(1, 1), (12, 23)]
         s0.points = [(1, 1), (12, 23)]
         s1.points = [(35, 67)]
```

In the example above, we dynamically create a new class called Shape and set one attribute called points, which is a list of points initially assigned one point (1, 1). We then create two instances of the Shape class, s0 and s1, in the same way that we do with other classes defined with the ***class*** statement. We then even added one more point to s0. Because attributes added to the class with built-in function type() are automatically treated as class attributes and shared by all instances of the class, changes to points of s0 are also reflected in the value of points of s1. However, the third-to-last statement above adds an attribute with the same name as the class attribute to object s1, which only belongs to s1. As a result, a change to this attribute of s1 has no effect on s0.

As we have already seen above, with a defined class or an instance of a defined class, you can modify the class or instance by adding or deleting attributes to/from the class or instance. Moreover, you can even add or delete a new method dynamically to or from a class or instance of a class.

One way to add an attribute to an object, either a class or an instance of a class, is to directly name a new attribute and assign a value in the same way that we

https://doi.org/10.15215/remix/9781998944088.01

introduce a new variable to a program. For example, we can add a new attribute called shape_type to s0 instantiated above with the first statement of code below:

```
In [ ]:  s0.shape_type = 'line'
         print(f'The shape is a {s0.shape_type}, with points
           {s0.points}')

Out [ ]: The shape is a line, with points [(1, 1), (12, 23)]
```

The second statement has proved that a new attribute shape_type has been added to object s0, and now this particular instance of Shape is called line.

A new attribute can also be added by using the built-in function setattr(). To add the shape_type attribute to s0 with setattr(), run the following statement:

```
In [ ]:  setattr(s0, 'shape_type', 'rectangle')
         # this will only add attribute shape_type to object s0
         s0.shape_type

Out [ ]: 'rectangle'
```

An attribute can also be deleted from an object using the built-in function delattr(). When you want to add or delete an attribute, but you are not sure if the attribute exits, you can use built-in function hasattr() to check, as shown below:

```
In [ ]:  print(hasattr(s0, 'shape_type'))
         delattr(s0, 'shape_type')
         print(hasattr(s0, 'shape_type'))

Out [ ]: True
         False
```

Remember, attributes added to an instance of a class will not be seen by other instances of the same class. If we want to make the shape_type attribute visible to all instances of the Shape class because every shape should have a shape type, we need to add the shape_type attribute to the class to make it a class attribute. This is done with the following statement:

```
Shape.shape_type = 'point'
```

From now on, all instances of the Shape class will have an attribute called shape_type, as shown in the examples below:

https://doi.org/10.15215/remix/9781998944088.01

```
In [ ]:  Shape.shape_type = 'point'
         print(f's0.shape_type = {s0.shape_type}')
         print(f's1.shape_type = {s1.shape_type}')

Out [ ]: s0.shape_type = point
         s1.shape_type = point
```

As you may have noted, the attribute shape_type and its value, added to the Shape class, have been propagated to both s0 and s1 because the shape_type attribute was added as a class attribute. By comparison, the attribute later added to an individual instance of the class is the attribute of that instance only. This is shown in the following code sample:

```
In [ ]:  print(s0.points, s1.points)
         s0.weight = 1
         print(s0.weight)
         hasattr(s1, 'weight')

Out [ ]: [(1, 1), (12, 23), (2, 3), (2, 3)] [(1, 1), (12, 23), (2, 3), (2, 3)]
         1
         False
```

The example shows that the new attribute weight was only added to object s0, and s1 does not have the attribute. Again, if you want the weight attribute and its value to be shared by all instances of the class, you have to add the attribute to the class directly, as shown below:

```
In [ ]:  Shape.weight = 1
         print('s0 weight = ', s0.weight)
         print('s1 weight = ', s1.weight)

Out [ ]: s0 weight = 1
         s1 weight = 1
```

How can you add a new method to an already defined class? You can do this in almost the same way as you would add new attributes to a class or instance of a class, as shown in the following example:

```
In [ ]:  def print_points(self):   # we define a function with a
          parameter
           for i, p in enumerate(self.points):
             print(f'point {i} at {p}')

         Shape.print_points = print_points   # we now attach the
          function

         s0.print_points()   # the method is called on s0
```

https://doi.org/10.15215/remix/9781998944088.01

```
Out [ ]:  Point 1 at (1, 1)
          Point 2 at (12, 23)
```

This provides programmers with so much power, as they can create new classes and modify all aspects of classes dynamically. For example, we know we can only define one __init__() in a class definition, so we only have one constructor to use when creating new instances of the class. By attaching a different properly defined method to the __init__ attribute, we are now able to construct a new instance of a class in whatever way we want, as shown in the following example:

```
In [ ]:  def print_shape(self):
           print(f'A {self.shape_type} has {len(self.points)}
           point(s):')
           for i, p in enumerate(self.points, 1):
             print(f'Point {i} at {p}')

         Shape.print_shape = print_shape

         def c1(self, *points):   # a point
           self.points = list(points)
           self.shape_type = 'point'

         def c2(self, *points):   # a line, a shape made of two
           points
           self.points = list(points)
           self.shape_type = 'line'

         def c3(self, *points):   # a triangle, a shape made of
           three points
           self.points = list(points)
           self.shape_type = 'triangle'

         Shape.__init__= c1   # constructor for one point
         p1 = Shape((3,5))
         Shape.__init__= c2 # constructor for a line
         l1 = Shape((3,5), (12, 35))
         Shape.__init__= c3   # constructor for a triangle
         t1 = Shape((3,5), (12, 35), (26, 87))

         print(p1.print_shape())
         print(l1.print_shape())
         print(t1.print_shape())
```

https://doi.org/10.15215/remix/9781998944088.01

```
Out [ ]:  A point has 1 point(s):
          Point 1 at (3, 5)
          None
          A line has 2 point(s):
          Point 1 at (3, 5)
          Point 2 at (12, 35)
          None
          A triangle has 3 point(s):
          Point 1 at (3, 5)
          Point 2 at (12, 35)
          Point 3 at (26, 87)
          None
```

In this example, we defined three methods to use as constructors or initiators for the shape class we previously defined. Constructor c1 is for creating point objects, c2 is for creating line objects, and c3 is for creating triangle objects. We then attach each method to the __init__ attribute of the shape class to create the shape object we want. We also defined a method called print_shape() for the class, just to show the results of the three different constructors.

As you may imagine, however, the consequence of modifying instances of a class is that different instances of the same class may have totally different attributes. In an extreme case, two instances of the same class can have totally different attributes. This is something you need to keep in mind when enjoying the freedom offered by Python.

Keeping Objects in Permanent Storage

As mentioned previously, classes and objects are a very powerful way to model the world. Hence, in a program, classes and objects can be used to represent information and knowledge for real-world application. You do not want to lose that information and knowledge whenever you shut down the computer. Instead, you want to keep this information and knowledge in permanent storage and reuse it when you need it. For example, you may have developed a management system using the Student class, defined earlier in this section, and created a number of objects of the Student class containing information about these students. You need to reuse the information about these students contained in those student objects next time you turn on the computer and run the system. How can you do that?

Previously, we discussed defining the __repr__ dunder method, which returns a string representation of an object that can be in the form of list, tuple, or dictionary, as shown in the following example:

https://doi.org/10.15215/remix/9781998944088.01

In [ ]:
```
class Employee:
  age : int = 20
  salary : float = 30000
  def __init__(self, firstname, lastname):
    self.firstname = firstname
    self.lastname = lastname

  def __str__(self):
    return f'{self.firstname} {self.lastname}, {self.
  age}, {self.salary}'

  def __repr__(self):
    rdict = {'firstname':self.
  firstname,'lastname':self.lastname, 'age':self.age,
  'salary':self.salary}
    return f"{rdict}"

e1 = Employee('Jack', 'Smith')
e1.age =37
e1.salary = 56900
e2 = Employee('Jone', 'Doe')
print(e1)   # this will call the __str__ method
e2 # this will call the __repr__ method
```

Out [ ]:
```
Jack Smith, 37, 56900
{'firstname': 'Jone', 'lastname': 'Doe', 'age': 20, 'salary': 30000}
```

With the __repr__() method for a class, you can save the representation of these objects into a file. To use the object representations stored in a file, you will need to write another function/method to pick up the information from each object representation and add it to its respective object. The two processes must work together to ensure that objects can be correctly restored from files. Neither process is easy. Fortunately, there is a Python module called pickle already developed just for this purpose.

In formal terms, saving data, especially complex data such as objects in OOP, in permanent storage is called serialization, whereas restoring data from permanent storage back to its original form in programs is called deserialization. The pickle module is a library implemented for serializing and deserializing objects in Python. The pickling/serializing process converts objects with hierarchical structure into a byte stream ready to save on a binary file, send across a network such as the internet, or store in a database, whereas the unpickling/deserializing process does the reverse: it converts a byte stream from a binary file or database, or received from a network, back into the object hierarchy.

https://doi.org/10.15215/remix/9781998944088.01

There are some good sections and documents on the internet that explain how to use the pickle module for your Python programming needs. When you do need to use it, be aware of the following:

1. Unpickling with the pickle module is not secure. Unpickling objects from unknown and untrusted sources can be dangerous because harmful executable code may be deserialized into your computer memory.
2. Not all objects can be pickled. You need to know what data types and objects can be pickled before you pickle them. Pickling unpicklable objects will raise exception.

Chapter Summary

- Object-oriented programming is an important approach to object modelling and programming.
- Abstraction, information hiding, and inheritance are important concepts and principles in OOP.
- New classes can be defined with the ***class*** statement.
- A class contains attributes/properties and methods.
- In Python, there are attributes called class attributes, which can have default values.
- The __init__ method is called upon when instantiating new instances of classes.
- Except for class attributes, attributes of individual objects of a class don't need to be explicitly declared in a class definition.
- Instead, attributes of a class are introduced in the definition of the __init__ method.
- In Python, each class can only have one constructor for the initializing instance of the class. That means that you can define only one __init__ method within a class.
- Methods of a class are defined the same way as functions, except that the first parameter of a method definition needs to be self, which refers to the object, an instance of the class on which the method is being called.
- There are methods called class methods in Python.
- There are also methods called static methods in Python class definition.
- A number of dunder methods can be defined in a class to achieve neat and powerful programming effects.
- Some dunder methods can be redefined in a class to mimic arithmetic operators.

https://doi.org/10.15215/remix/9781998944088.01

- Some important and interesting dunder methods include __repr__, __init__, __call__, __new__, and __len__.
- Class can also be used as a decorator of a function/method.
- The built-in function property() has two interesting uses: one is to attach specific getter, setter, and deleter functions to a property/attribute of an object, and the other is to use it as a decorator—to create the name of a method to be used as a property/attribute name.

Exercises

1. Run the following code in a cell of one of the Jupyter Notebooks created for the chapter and answer the questions below:

```
class myClassB(object):
    pass

print(myClassB.__dict__)
dir(myClassB)
```

   a. What does each statement do?
   b. What is the output from print(myClassB.__dict__) statement?
   c. What does dir(myClassB) return?
   d. Python dir() function returns a list of the attributes and methods of any object. In the code above, no attribute or method is defined in the definition of class myClassB. Why does the list returned from dir(myClassB) have so many items in it? Find out and explain what each item is.

2. Mentally run the code below and write down the output of the program:

```
class Employee:
    def __init__(self, firstname, lastname):
         self.firstname = firstname
         self.lastname = lastname

    def setFullname(self, fullname):
         names = fullname.split()
         self.firstname = names[0]
         self.lastname = names[1]

    def getFullname(self):
```

https://doi.org/10.15215/remix/9781998944088.01

```
        return f'{self.firstname} {self.lastname}'

 fullname = property(getFullname, setFullname)

e1 = Employee('Jack', 'Smith')
print(e1.fullname)
e2 = e1
e2.fullname = 'John Doe'
print(e2.fullname)
```

3. For the Employee class defined in Exercise 2, define method __str__() so that the statement print(e1) will display the full name of the employee.
4. For the Employee class defined below, define setter and getter methods for attribute age and salary, respectively.

```
class Employee:
    age : int = 20
    salary : float = 30000
    def __init__(self, firstname, lastname):
         self.firstname = firstname
         self.lastname = lastname
```

5. For the Employee class defined in Exercise 4, define method __repr__() to return a dictionary whose item is a pair of that includes the attribute name and its value, such as 'firstname': 'John'.
6. Define a class named Quiz_question that models multiple-choice questions, including the correct answers. In addition to a constructor—the __init__ method to construct the objects of the class—there should be other methods allowing the user to change the description of the question, the individual choices, and the correct answers.
7. If we allow the number of choices to vary for the multiple-choice questions modelled by the Quiz_question class defined for Exercise 6, what changes need to be made to the Quiz_question class?
8. Define a class named Quiz that uses the Quiz_question class defined above to model a quiz that contains a number of quiz questions. It should have methods for a user to create a quiz, to add quiz questions to the quiz, to display a list of all quiz questions in a quiz for review, and to execute a quiz on a user and calculate the score.

https://doi.org/10.15215/remix/9781998944088.01

Project

1. Using the Quiz_question and Quiz classes you developed in Exercises 7 and 8 above, develop a terminal-based quiz system that will allow the user to do the following:
    - Create a quiz.
    - Select a quiz and add new quiz questions.
    - Select a quiz and preview all the quiz questions.
    - Select a quiz and execute the quiz by presenting the quiz questions one by one.
    - At the end of the quiz, calculate the user's score as a percentage and show the correct answers to incorrectly answered questions. The quiz questions should be displayed to the user one by one during the quiz.

https://doi.org/10.15215/remix/9781998944088.01

This page intentionally left blank

Chapter 8

Modules and Packages

Divide-and-conquer is a fundamental but effective strategy in problem solving as well as system design and development. This is because big problems can often be divided into smaller problems that can be easily solved or have already been solved, and large systems can often be made of smaller ones that can be easily created or are already readily available. Not only that, but the divide-and-conquer method also results in easier system maintenance, better quality insurance, quicker error detection and correction, and better reusability.

In Chapter 6, we learned how functions can be used in problem solving, and in Chapter 7, we studied object-oriented programming and learned how classes and objects can be used in system development and to solve problems. In computing, both functions and objects are common programming technologies that implement divide-and-conquer strategies.

In this chapter, we will study modules and packages that can also be used to implement the divide-and-conquer strategy in programming. In this way, we will learn how to create and use modules in programming and software development. We will also study some Python modules and packages that are readily available and often needed to solve problems and develop computer applications.

Learning Objectives

After completing this chapter, you should be able to

- describe modules.
- explain what packages are, what files are required, and how they are structured in a file system.
- import and use modules already in the Python programming/development environment.

https://doi.org/10.15215/remix/9781998944088.01

- import and use specific parts from a module or specific modules from a package.
- explain the functionalities of some standard and widely used modules, and use them comfortably in programming.
- write and use your own modules and packages.

8.1 Creating Modules and Packages

As mentioned in previous sections, a module can be just a Python script file defining functions, classes, and other program constructs such as variables and constants. The following is an example of a Python module:

Code sample in VS Code IDE

```
"""
The module in this file defines some functions often used
  in calculations related to circles

Author: John Doe
Date: March 30, 2019
Version: 1.0
"""


PAI = 3.1415926


def area(r):
  """Calculate the area of a circle with radius r."""
  return PAI * r ** 2   # calculate the area


def circumference(r):
  """Calculate the circumference of a circle with radius
  r."""
  return 2 * PAI * r

```

The file can then be imported into a Python program and used as a module, as shown below:

```
In [ ]:  import circle

         print(f'The area of a circle with a radius of 12 is
           {circle.area(12)}')
```

```
Out [ ]: The area of a circle with a radius of 12 is 452.3893344
```

https://doi.org/10.15215/remix/9781998944088.01

How do you create a module or package and publish it at https://pypi.org/ for the Python community at large so that it can be found and installed with the pip command?

Suppose you want to create a package for developing e-learning applications. You would need to take the following steps to develop the package to make it available for the others in the Python community:

1. First, create a directory called mypackages, which will contain all the packages and modules that you will develop for yourself or the Python community at large.
2. Suppose you would like the package to be called elearn. You first need to check to see if the name has been used by others to name any top-level module or package published at http://pypi.python.org.
3. Under the mypackages directory, create a directory named elearn, which will contain everything for your elearn package or module.
4. Under this elearn directory, you can create Python script files (.py) and other subdirectories under which you may have other packages or modules.
5. To distinguish this elearn directory from other ordinary directories of the file system so that it can be searchable through a Python interpreter, create a special file called __init__.py right under the elearn directory. The __init__.py file should contain all the names and objects defined in the package or module, either directly or indirectly (by importing individual names and objects from each of the other files). The must-have __init__.py file is the entry point of each package or module.
6. Under the mypackages directory, write a special Python script file called setup.py, which imports a special module called setuptools and calls the setup function to prepare the package to be sent to the repository. The following is an example of the setup.py file.

```
import setuptools

setup(name = 'elearn',
version = '1.0beta',
description = 'A package for developing elearn
applications',
url = 'http://github.com/AU.CA/elearn',
author = 'SCIS Athabasca',
author_email = 'scis@athabascau.ca',
license = 'FSF free software foundation',
```

https://doi.org/10.15215/remix/9781998944088.01

```
packages = ['elearn'],
zip_safe = False)
```

7. If you are not really ready to submit the package to the repository but would rather test it or just want to use it by yourself, you can install the package locally so that it can be found with the ***import*** statement. The following is the command to be used to install the package when your current working directory is mypackages:

```
$ pip install .
```

8. To publish your elearn package, you need to
   a. Register the package with PyPi so that the online repository will know about your package and create an entry-point link to the elearn package at GitHub, as specified in the setup.py file.
   b. Create a single zip file that contains all Python script files.
   c. Upload the zip file to PYPI repository.

   By running the following command also under mypackages directory:

```
$ python setup.py register sdist upload
```

Upon completing the above steps, anyone on the internet will be able to install and use your elearn library to develop elearn applications using the following command:

```
$ pip install elearn
```

Please note that for your package to be available in a GitHub repository so that the link used in your entry at PYPI is valid, you will need to create or sign into an account with GitHub and do the following:

1. Create the project within your account at GitHub
2. Install Git on your computer and use the commands to manually synchronize your work on your computer with GitHub

VS Code IDE can work with Git and GitHub through respective GitHub extensions that can be downloaded and installed in VS Code so that your project in VS can be easily synchronized with GitHub.

https://doi.org/10.15215/remix/9781998944088.01

8.2 Using Modules and Packages

To use a module, we first import it, as shown in the following example:

Code sample in Python interactive mode

```
1
2 import circle
3
4 radius = float(input("Tell me the radius:"))
5
6 print(f"The area of a circle with the radius
    {radius} is {circle.area(radius)}")
7 print(f"The circumference of a circle with the
    radius {radius} is {circle.circumference(radius)}")
```

The result

```
Tell me the radius: 12.3
The area of a circle with the radius 12.3 is 475.2915444540001
The circumference of a circle with the radius 12.3 is 77.28317796
```

When importing a module, you can also give the module an alias to use, especially if the original name is too long or hard to remember. In the above code sample, the ***import*** statement ***import circle*** can be changed to ci, as in the following:

```
>>> import circle as ci
```

Then in the program file, you can use ci in place of circle, as shown below:

```
>>> print(f"The area of a circle with the radius {radius}
is {ci.area(radius)}")
```

When designing and coding a system, you may create and use as many modules as needed, but do so wisely and do not make a simple system too complicated. Managing too many unnecessary modules and files will consume time and resources as well, especially since many Python modules have already been developed by the great Python community for almost every application domain you can think of. In most cases, all you need is to know is what modules are available out there and what each module does, even if you do not want to learn about the details now.

In general, the ***import*** statement may take one of the following forms.

https://doi.org/10.15215/remix/9781998944088.01

IMPORT <THE NAME OF THE MODULE>

This is the simplest form, although you do need to know what the module has defined and know exactly what the name is, which is just the file name without the extension py.

IMPORT <THE NAME OF THE MODULE> AS <SIMPLER ALIAS FOR THE MODULE NAME>

Sometimes, the name of a module can be long and hard to remember, and giving the module an alias would make programming more efficient. For example, there is a module for mathematical plotting called matplotlib if you give an alias to the module when importing it as follows:

```
import matplotlib as mpl
```

A name such as o defined in the module can then be referred to using mpl.o, which is much simpler than matplotlib.o.

FROM <MODULE/PACKAGE NAME> IMPORT <NAME OF OBJECT OR MODULE>

When importing from a module, you can import a specific name you want to use instead of the entire module. For example, a mathematical module may have defined a number of mathematical constants such as PI and E. You may import only the one you want to use in your program.

As previously mentioned, a package usually contains other packages and modules, which can often be depicted as a tree. You can import just the package, module or even the name you want to use by using the ***import*** statement above. Again, the important thing is that you need to know where the thing you want to import is located within the tree. Assume from the root r the module m is located at r.n.m; then the ***import*** statement can be written as follows:

```
from r.n import m
```

This dot notation can also be used to import a module from a package without using from, as shown in the following example:

```
import matplotlib.pyplot as ppl
```

which imports the pyplot module from the matplotlib package and assigns an alias to the module.

https://doi.org/10.15215/remix/9781998944088.01

FROM <MODULE/PACKAGE NAME> IMPORT <NAME OF OBJECT OR MODULE> AS <ALIAS>

This is the last form an ***import*** statement can take. It gives an alias to the module/name imported from a package or module.

8.3 Install and Learn About Modules Developed by Others

To learn about modules already developed by others, your starting point can be a website called Python Package Index (PyPi) at https://pypi.org, where you can find, install, and publish Python packages and modules. By browsing this site, you will quickly get an idea what packages and modules have been already developed and published and are ready for use.

To see what modules and packages have already been installed on your computer, you can simply run pip list on a shell terminal such as PowerShell, as shown below:

```
PS C:\> pip list
Package              Version
-------------------  -----------
appdirs              1.4.4
argon2-cffi          20.1.0
asgiref              3.3.1
async-generator      1.10
attrs                20.3.0
backcall             0.2.0
bleach               3.2.1
certifi              2020.11.8
cffi                 1.14.3
chardet              3.0.4
colorama             0.4.4
decorator            4.4.2
defusedxml           0.6.0
distlib              0.3.1
Django               3.1.5
entrypoints          0.3
filelock             3.0.12
idna                 2.10
ipykernel            5.3.4
ipython              7.19.0
```

https://doi.org/10.15215/remix/9781998944088.01

```
ipython-genutils    0.2.0
ipywidgets          7.5.1
jedi                0.17.2
Jinja2              2.11.2
json5               0.9.5
jsonschema          3.2.0
jupyter             1.0.0
jupyter-client      6.1.7
jupyter-console     6.2.0
jupyter-core        4.6.3
jupyterlab          2.2.9
jupyterlab-pygments 0.1.2
jupyterlab-server   1.2.0
MarkupSafe          1.1.1
mistune             0.8.4
nbclient            0.5.1
nbconvert           6.0.7
nbformat            5.0.8
nest-asyncio        1.4.2
notebook            6.1.5
packaging           20.4
pandocfilters       1.4.3
parso               0.7.1
pickleshare         0.7.5
Pip                 21.1.3
pipenv              2020.11.15
prometheus-client   0.8.0
```

To find out what a particular installed module or package does, you can import the package or module into a Python interactive shell or a Jupyter Notebook cell, then run the dir command/statement to find out the names defined in the module or package and use the help command/statement to see more detailed information for a particular name.

```
PS C:\> python
Python 3.9.0 (tags/v3.9.0:9cf6752, Oct 5 2020, 15:34:40)
[MSC v.1927 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more
information.
>>> import math
```

https://doi.org/10.15215/remix/9781998944088.01

```
>>> dir(math)
['__doc__', '__loader__', '__name__', '__package__',
'__spec__', 'acos', 'acosh', 'asin , asinh , atan',
'atan2', 'ata
nh', 'ceil', 'comb', 'copysign', 'cos', 'cosh', 'degrees',
'dist', 'e', 'erf', 'erfc', 'exp', 'expmr, 'fabs', 'factoria
l', 'floor', 'fmod', 'frexp', 'fsum', 'gamma', 'gcd', 'hypot',
'inf', 'isclose', 'isfinite', 'isinf', 'isnan', 'isqrt',
'1cm', 'Idexp', 'lgamma', 'log', 'log10', 'Ioglp',
'log2', 'modf', 'nan', 'nextafter', 'perm', 'pi', 'pow',
'prod', 'rad
ians', 'remainder', 'sin', 'sinh', 'sqrt', 'tan', 'tanh',
'tau', 'trunc', 'ulp']
>>> help(math)
Help on built-in module math:

NAME
    math

DESCRIPTION
    This module provides access to the mathematical functions
    defined by the C standard.

FUNCTIONS
    acos(x, /)
        Return the arc cosine (measured in radians) of x.

        The result is between 0 and pi.

    acosh(x, /)
        Return the inverse hyperbolic cosine of x.

    asin(x, /)
        Return the arc sine (measured in radians) of x.

        The result is between -pi/2 and pi/2.

    asinh(x, /)
        Return the inverse hyperbolic sine of x.
```

https://doi.org/10.15215/remix/9781998944088.01

```
    atan(x, /)
        Return the arc tangent (measured in radians) of x.

        The result is between -pi/2 and pi/2.

    atan2(y, x, /)
        Return the arc tangent (measured in radians) of y/x.

        Unlike atan(y/x), the signs of both x and y are
considered.

    atanh(x, /)
        Return the inverse hyperbolic tangent of x.
```

Another way to learn about and navigate through the available modules and packages is to use a module called pydoc. However, this module is usually run from a command shell or PowerShell terminal as shown below:

```
(base) PS C:\Users\james> python pydoc
pydoc - the Python documentation tool

pydoc <name> …
    Show text documentation on something. <name> may be
the name of a
    Python keyword, topic, function, module, or package,
or a dotted
    reference to a class or function within a module or
module in a
    package. If <name> contains a '\', it is used as the
path to a
    Python source file to document. If name is
'keywords', 'topics',
    or 'modules', a listing of these things is displayed.

pydoc -k <keyword>
    Search for a keyword in the synopsis lines of all
available modules.

pydoc -n <hostname>
```

https://doi.org/10.15215/remix/9781998944088.01

```
    Start an HTTP server with the given hostname
(default: localhost).

pydoc -p <port>
    Start an HTTP server on the given port on the local
machine. Port
    number 0 can be used to get an arbitrary unused port.

pydoc -b
    Start an HTTP server on an arbitrary unused port and
open a Web browser
    to interactively browse documentation. This option
can be used in
    combination with -n and/or -p.

pydoc -w <name> …
    Write out the HTML documentation for a module to a
file in the current
    directory. If <name> contains a '\', it is treated as
a filename; if
    it names a directory, documentation is written for
all the contents.

(base) PS C:\Users\james>
```

Note that when you want to run a Python module as a normal script, run Python with the -m switch before the module name.

As shown above, if you want to get documentation on something, just add the something behind pydoc. For example, if you want to see the documentation on the math module, run the following command in the command shell:

```
python -m pydoc math
```

The resulting documentation is shown below:

```
Help on built-in module math:

NAME
  math
```

https://doi.org/10.15215/remix/9781998944088.01

```
DESCRIPTION
  This module provides access to the mathematical
functions
  defined by the C standard.

FUNCTIONS
  acos(x, /)
   Return the arc cosine (measured in radians) of x.

  acosh(x, /)
   Return the inverse hyperbolic cosine of x.

  asin(x, /)
   Return the arc sine (measured in radians) of x.

  asinh(x, /)
   Return the inverse hyperbolic sine of x.

  atan(x, /)
   Return the arc tangent (measured in radians) of x.

  atan2(y, x, /)
   Return the arc tangent (measured in radians) of y/x.

   Unlike atan(y/x), the signs of both x and y are
considered.

  atanh(x, /)
   Return the inverse hyperbolic tangent of x.
```

With some installations of Python, such as those installed with Anaconda, pydoc has been made available as executable directly from command shell, as shown here:

```
(base) PS C:\Users\james> pydoc math
Help on built-in module math:

NAME
  math
```

https://doi.org/10.15215/remix/9781998944088.01

```
DESCRIPTION
  This module provides access to the mathematical functions
  defined by the C standard.

FUNCTIONS
  acos(x, /)
   Return the arc cosine (measured in radians) of x.

  acosh(x, /)
   Return the inverse hyperbolic cosine of x.

  asin(x, /)
   Return the arc sine (measured in radians) of x.

  asinh(x, /)
   Return the inverse hyperbolic sine of x.

  atan(x, /)
   Return the arc tangent (measured in radians) of x.

  atan2(y, x, /)
   Return the arc tangent (measured in radians) of y/x.

   Unlike atan(y/x), the signs of both x and y are considered.

  atanh(x, /)
   Return the inverse hyperbolic tangent of x.
```

pydoc has some switches that can be used when running it as a Python module or an executable directly from the command shell. These switches include the following.

-K <KEYWORD>

Used to search for a keyword in the synopsis lines of all available modules installed. The following example searches for all modules that has the word "hash" in the synopsis.

```
PS C:\Users\james> python -m pydoc -k hash
```

-N <HOSTNAME>

Used to start an HTTP server with the given <hostname>. When no hostname is given, localhost is default.

https://doi.org/10.15215/remix/9781998944088.01

-P <PORT>

Used to start an HTTP server on the given <port> on the local machine. Port number 0 can be used to get an arbitrary unused port.

-B

Used to start an HTTP server on an arbitrary unused port and open a web browser to interactively browse documentation. This option can be used in combination with -n and/or -p.

The following example starts a server on the local machine at an available port and launches a web browser to browse all available modules and packages.

```
(base) PS S:\Dev\learn-python> pydoc -b
Server ready at http://localhost:30128/
Server commands: [b]rowser, [q]uit
server>
```

Please note that if you type q to quit from the program, the server will be down and no longer available.

The browser opened by the example above will look like this:

Index of Modules

Built-In Modules

_abc	_imp	_stat	
_ast	_io	_statistics	builtins
_bisect	_json	_string	cmath
_blake2	_locale	_struct	errno
_codecs	_lsprof	_symtable	faulthandler
_codecs_cn	_md5	_thread	gc
_codecs_hk	_multibvtecodec	_tokenize	itertools
_codecs_iso2022	_opcode	_tracemalloc	marshal
_codecs_jp	_operator	_typing	math
_codecs_kr	_pickle	_warnings	mmap
_codecs_tw	_random	_weakref	msvcrt
_collections	_shal	_winapi	nt
_contextvars	_sha256	_xxsubinterpreters	sys
_csv	_sha3	array	time
_datetime	_sha512	atexit	winreg
_functools	_signal	audioop	xxsubtype
_heapq	_sre	binascii	zlib

https://doi.org/10.15215/remix/9781998944088.01

Please note that the above only shows the built-in modules. There is more about nonbuilt-in modules if you scroll down to read further.

-W <NAME>...

Used to write out the HTML documentation for a module to a file in the current directory. If the name contains a backslash \, it is treated as a filename; if it names a directory, documentation is written for all the contents. The following example generates documentation in HTML for a module called timeit:

```
(base) PS S:\Dev\learn-python > pydoc -w timeit
wrote timeit.html
```

As can be seen, using the pydoc -b in command shell can easily access a server and start to browse documentation for all the available modules and packages installed on your computer. For example, if we want to learn more about the timeit module, we can browse or search for timeit within the first page of the browser window launched by the pydoc -b command, then click the link to see the details of the documentation.

The remainder of this chapter introduces some Python modules that you may need to use in the development of different computer applications.

8.4 Module for Generating Random Numbers

In certain computer applications, you often need to generate random numbers. For example, you need random numbers to automatically generate quizzes. In computer security, big random numbers play very important roles.

The random module comes with the standard Python distribution library. It provides functions and class definitions related to generating pseudorandom numbers, though they are not good enough to be used for security purposes. To see what names are defined at the top level of the module, run the dir(random) statement after importing the module to get a list of the names, as shown below:

```
>>> import random
>>> dir(random)
['BPF', 'LOG4', 'NV_MAGICCONST', 'RECIP_BPF',
'Random', 'SG_MAGICCONST', 'SystemRandom', 'TWOPI', '_
BuiltinMethodType', '_MethodType', '_Sequence', '_Set',
'__all__', '__builtins__', '__cached__', '__doc__',
'__file__', '__loader__', '__name__', '__package__',
```

https://doi.org/10.15215/remix/9781998944088.01

```
'__spec__', '_acos', '_bisect', '_ceil', '_cos', '_e',
'_exp', '_inst', '_itertools', '_log', '_os', '_pi',
'_random', '_sha512', '_sin', '_sqrt', '_test', '_
test_generator', '_urandom', '_warn', 'betavariate',
'choice', 'choices', 'expovariate', 'gammavariate',
'gauss', 'getrandbits', 'getstate', 'lognormvariate',
'normalvariate', 'paretovariate', 'randint',
'random', 'randrange', 'sample', 'seed', 'setstate',
'shuffle', 'triangular', 'uniform', 'vonmisesvariate',
'weibullvariate']
```

In the list, names with leading underscores are often intended to be hidden. It's important to know only what those without leading underscores are and what they do. You can use the ***help*** statement to find out what randint is, for example:

```
>>> help(random.randint)
```

Running help on method randint in the random module outputs the following:

randint(a, b) method of random.Random instance

This returns a random integer within the range [a, b], including both end points.

As you can see with the ***help*** statement, all these names defined in the random module are functions and methods. These functions and methods can be categorized as bookkeeping methods, random integer generating methods, random real/float number generating methods, random item generating methods, or items from a sequence. The module also provides alternate random number generators such as the one provided by the operating system.

In the following, you will run through the functions provided in the random module. These functions are categorized into four groups: bookkeeping functions, functions randomly generating integers, functions randomly generating float/real numbers, and functions randomly selecting items from a sequence.

https://doi.org/10.15215/remix/9781998944088.01

Functions for Bookkeeping

SEED(A = NONE, VERSION = 2)

This initializes the random number generator by seeding the generator. If there is no argument for a, the current time is used to seed the generator. Version 2 is the default version of the algorithm.

```
>>> import random
>>> random.seed(a = 3)
```

GETSTATE()

This gets and returns the current internal state of the random number generator.

```
>>> s1 = random.getstate()
```

SETSTATE(STATE)

This sets the internal state of the random number generator to state.

```
>>> random.setstate(s1)
```

GETRANDBITS(K)

This generates and returns an integer with k random bits.

```
>>> bin(random.getrandbits(9))
'0b100101111'
>>> bin(random.getrandbits(9))   # will get a different
number
'0b10111101'
```

Functions for Generating Random Integers

RANDRANGE(START, STOP=NONE, STEP = 1)

This generates and returns a random number within the given range.

```
>>> random.randrange(99)
69
>>> random.randrange(99)
```

https://doi.org/10.15215/remix/9781998944088.01

RANDINT(A, B)

This generates and returns a random integer within the given range.

```
>>> random.randint(1, 100)
42
>>> random.randint(1, 100)
85
```

Functions for Randomly Generating Float Numbers

RANDOM()

This randomly generates and returns a random float number between 0 and 1, including 0 but excluding 1.

```
>>> random.random()
0.4084252811341471
```

UNIFORM(A, B)

This randomly generates and returns a float number between a and b.

```
>>> random.uniform(2, 99)
73.1658416986511
>>> random.uniform(2, 99)
92.92610150048253
```

TRIANGULAR(LOW = 0.0, HIGH = 1.0, MODE = NONE)

This randomly generates and returns a random float number between low and high. The third argument for mode parameter can be used to indicate the preference for the outcome. If it is closer to low, it is more likely to get a random float number on the low end, for example. Internally, the default value for mode is the midpoint between low and high.

```
>>> random.triangular(2, 99)
84.02716580051677
>>> random.triangular(2,99,35)
50.303535641546
```

BETAVARIATE(ALPHA, BETA)

This randomly generates and returns a random float number between 0 and 1 based on the beta distribution of statistics. Parameters alpha and beta

https://doi.org/10.15215/remix/9781998944088.01

(both > 0) are used to set the conditions of the distribution, as used in the beta distribution function.

```
>>> random.betavariate(2, 99)
0.011368344795580798
>>> random.betavariate(2, 99)
0.019428131869773747
```

EXPOVARIATE(LAMBDA)

This randomly generates and returns a random float number between 0 and 1, or between 0 and −1 if ***lambda*** is negative, based on the exponential distribution of statistics.

```
>>> random.expovariate(2)
0.379317249922913
```

GAMMAVARIATE(ALPHA, BETA)

This randomly generates and returns a random float number between 0 and 1 based on the gamma distribution of statistics. Parameters alpha and beta (both > 0) are used to set the conditions of the distribution, as used in the gamma distribution function.

```
>>> random.gammavariate(2,99)
43.06391063895096
```

GAUSS(MU, SIGMA)

This randomly generates and returns a random float number between 0 and 1 based on the Gaussian distribution of probability theories. Parameter mu is the mean, and sigma is the standard deviation, as used in the distribution function.

```
>>> random.gauss(2, 99)
38.05513497609059
```

LOGNORMVARIATE(MU, SIGMA)

This generates and returns a random float number between 0 and 1 based on a log-normal distribution of probability theories. Parameter mu is the mean, and sigma is the standard deviation, as used in the log normal distribution function.

```
>>> random.lognormvariate(2, 99)
9.252497191266324e-41
```

https://doi.org/10.15215/remix/9781998944088.01

NORMALVARIATE(MU, SIGMA)

This generates and returns a random float number between 0 and 1 based on the normal distribution of probability theories. Parameter mu is the mean, and sigma is the standard deviation, as used in the normal distribution function.

```
>>> random.normalvariate(2, 99)
155.45854862650918
```

VONMISESVARIATE(MU, KAPPA)

This generates and returns a random float number between 0 and 1 based on the von Mises distribution of directional statistics. Parameter mu is the mean angle, expressed in radians between 0 and 2 * pi, whereas kappa is the concentration.

```
>>> random.vonmisesvariate(2, 99)
1.9289474404869416
```

PARETOVARIATE(ALPHA)

This generates and returns a random float number between 0 and 1 based on the Pareto distribution of probability theories. Parameter alpha is used to indicate the shape.

```
>>> random.paretovariate(2)
1.7794461337233882
```

WEIBULLVARIATE(ALPHA,BETA)

This generates and returns a random float number between 0 and 1 based on the Weibull distribution of statistics. Parameter alpha is the scale, and beta is the shape, as in its mathematical function.

```
>>> random.weibullvariate(2, 99)
2.0164248554211417
```

Functions for Randomly Selected Item(s) from Sequences

Functions in this group are often used in statistics.

CHOICE(POPULATION)

This generates and returns a random element from the given population (in the form of a Python sequence).

https://doi.org/10.15215/remix/9781998944088.01

```
>>> random.choice(range(1, 1000))
536
>>> random.choice(list("The message will look like this
but"))
'l'
>>> random.choice(list("The message will look like this
but"))
't'
```

CHOICES(POPULATION, WEIGHTS = NONE, *, CUM_WEIGHTS = NONE, K = 1)

This generates and returns a list with k randomly selected items from the given sequence, with weights or cumulative weights considered if they are given. An optional argument for weights should be a list of integers specifying how likely the corresponding items are to be selected. The number of integers in weights must match the number of items in the population; optional argument for cum_weights is also a list. A value in the list is shown as cumulation of weights so far. Argument weights and cum_weights are just different ways of representing the same preferences.

```
>>> random.choices(range(1, 1000), k = 5)
[456, 79, 57, 51, 110]
>>> random.choices(range(1, 10), weights = [2, 3, 5, 8,
2, 3, 3, 2, 10], k = 5)
[9, 4, 6, 3, 8]
```

This function can be very useful when you want to generate a quiz by randomly selecting questions from a question bank. Assume the question bank is called q_bank, and each quiz will have 10 questions. The random choices can be easily made by calling the function as follows:

```
>>> random.choices(q_bank, k = 10)
```

SHUFFLE(POPULATION, RANDOM = NONE)

This takes a sequence, shuffles the members, then returns the sequence with the members randomly shuffled.

```
>>> l = list(range(10))
>>> l
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

https://doi.org/10.15215/remix/9781998944088.01

```
>>> random.shuffle(l)
>>> l
[2, 9, 0, 4, 1, 6, 5, 3, 7, 8]
```

SAMPLE(POPULATION, K)

This generates and returns a sample for a given population. It seems similar to choices(), but internally the algorithm should check whether the choices would make a sample of the population according to some sampling criteria in statistics.

```
>>> random.sample(range(1, 1000), k = 5)
[251, 337, 822, 499, 853]
```

In the following programming problem in Table 8-1, you will study how well the random generator works. You will randomly generate 1000 integer numbers within the range of 0 to 999 and visualize the randomness of these numbers.

Table 8-1: Case study: Random number generator quality

The problem	This case study will find out how good the random number generator is.
The analysis and design	Random numbers generated by computers are not completely random. Rather, they are pseudorandom sequences. To study how good a generator is, you need to see if the numbers generated by the generator look random. We will show that in a two-dimensional chart to visualize it.

The code	`import random`
	`import matplotlib.pyplot as pyp`
	`data = [random.randint(0,100) for i in range(1000)]`
	`pyp.plot(list(range(1000)), data)`
	`pyp.xlabel('Run #')`
	`pyp.ylabel('Random Integer')`
	`pyp.title('Visualization of Randomness')`
	`pyp.show()`

The result	Figure 8-1 below is plotted by the program. It seems the random generator works very well.

https://doi.org/10.15215/remix/9781998944088.01

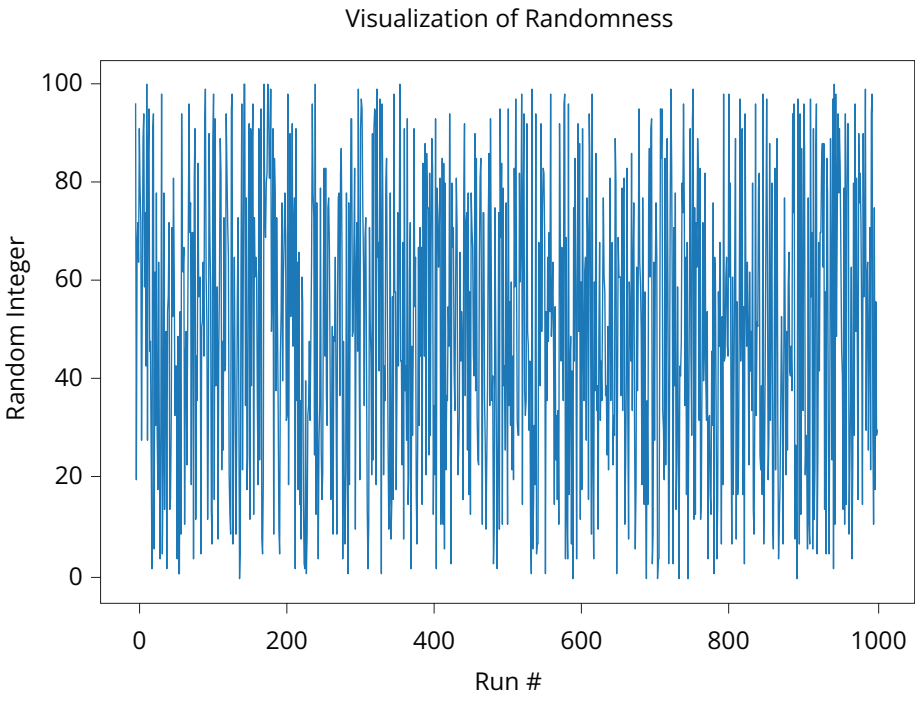


Figure 8-1: Visualization of randomness

8.5 Module for Mathematical Operations

The math module also comes with the standard Python distribution library and provides many useful mathematical functions that you may need. To learn what the module has for you, type the following statement in a Jupyter Notebook cell and click run or press Shift+Enter at the same time. You will see the documentation for all these functions as well as the constants defined in the math module.

To learn what is defined in the module, run the dir statement on math after import, as we did before with the random module:

```
>>> import math
>>> dir(math)
['__doc__', '__loader__', '__name__', '__package__',
'__spec__', 'acos', 'acosh', 'asin', 'asinh', 'atan',
'atan2', 'atanh', 'ceil', 'copysign', 'cos', 'cosh',
'degrees', 'e', 'erf', 'erfc', 'exp', 'expm1', 'fabs',
'factorial', 'floor', 'fmod', 'frexp', 'fsum',
'gamma', 'gcd', 'hypot', 'inf', 'isclose', 'isfinite',
'isinf', 'isnan', 'ldexp', 'lgamma', 'log', 'log10',
```

https://doi.org/10.15215/remix/9781998944088.01

```
'log1p', 'log2', 'modf', 'nan', 'pi', 'pow', 'radians',
'remainder', 'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'tau',
'trunc']
```

Then run help on a particular name to find out what that name, such as ceil, does:

```
>>> help(math.ceil)
```

Running help on the built-in function ceil in the math module outputs the following:

ceil(x, /)
Return the ceiling of x as an Integral.

This returns the smallest integer >= x.

In the following, we run through the functions available in the math module.

ACOS(X)

This calculates and returns the arc cosine (measured in radians) of x, where 0 <= x < = 1.

```
>>> import math
>>> math.acos(0.5)
1.0471975511965979
```

ACOSH(X)

This calculates and returns the inverse hyperbolic cosine of x. (Google "hyperbolic functions" to learn more.)

```
>>> math.acosh(1000)
7.600902209541989
```

ASIN(X)

This calculates and returns the arc sine (measured in radians) of x.

```
>>> math.asin(0.5)
0.5235987755982989
```

https://doi.org/10.15215/remix/9781998944088.01

MATH.ASINH(X)

This calculates and returns the inverse hyperbolic sine of x.

```
>>> math.asinh(5)
2.3124383412727525
```

MATH.ATAN(X)

This calculates and returns the arc tangent (measured in radians) of x.

```
>>> math.atan(0.5)
0.4636476090008061
```

MATH.ATAN2(Y, X)

This calculates and returns the arc tangent (measured in radians) of y/x. Unlike atan(y/x), the signs of both x and y are considered.

```
>>> math.atan2(3, -5)
2.601173153319209
```

MATH.ATANH(X)

This calculates and returns the inverse hyperbolic tangent of x.

```
>>> math.atanh(0.5)
0.5493061443340549
```

MATH.CEIL(X)

This calculates and returns the ceiling of x as an Integral—the smallest integer >= x.

```
>>> math.ceil(0.5)
1
```

MATH.COPYSIGN(X, Y)

This calculates and returns a float with the magnitude (absolute value) of x but the sign of y. On platforms that support signed zeros, copysign(1.0, −0.0) returns −1.0.

```
>>> math.copysign(0.5, -1.2)
-0.5
```

https://doi.org/10.15215/remix/9781998944088.01

MATH.COS(X)

This calculates and returns the cosine of x (measured in radians).

```
>>> math.cos(0.5)
0.8775825618903728
```

MATH.COSH(X)

This calculates and returns the hyperbolic cosine of x.

```
>>> math.cosh(50)
2.592352764293536e+21
```

MATH.DEGREES(X)

This converts angle x from radians to degrees and returns the result.

```
>>> math.degrees(0.5)
28.64788975654116
```

MATH.ERF(X)

This is the error function at x, as defined in statistics.

```
>>> math.erf(0.5)
0.5204998778130465
```

MATH.ERFC(X)

This is the complementary error function at x, so that erf(x) + erfc(x) = 1.

```
>>> math.erfc(0.5)
0.4795001221869534
```

MATH.EXP(X)

This calculates and returns e raised to the power of x. This is the same as math.e**x, or math.pow(math.e, x).

```
>>> math.exp(5)
148.4131591025766
```

MATH.EXPM1(X)

This calculates and returns exp(x)−1. This function avoids the loss of precision involved in the direct evaluation of exp(x)−1 for small x.

https://doi.org/10.15215/remix/9781998944088.01

```
>>> math.expm1(5)
147.4131591025766
```

MATH.FABS(X)

This calculates and returns the absolute value of the float x.

```
>>> math.fabs(-23.6)
23.6
```

MATH.FACTORIAL(X)

This calculates and returns x!. It will raise a ValueError if x is negative or nonintegral.

```
>>> math.factorial(23)
25852016738884976640000
```

MATH.FLOOR(X)

This calculates and returns the floor of x as an integer—that is, the return value is the largest integer < = x.

```
>>> math.floor(2.3)
2
```

MATH.FMOD(X, Y)

This calculates and returns fmod(x, y), according to platform C. x % y may differ.

```
>>> math.fmod(2.3, 1.5)
0.7999999999999998
```

MATH.FREXP(X)

This calculates the mantissa and exponent of x and returns a pair (m, e). m is a float, and e is an integer, such that x = m * 2** e. If x is 0, m and e are both 0. Otherwise, 0.5 <= abs(m) < 1.0.

```
>>> math.frexp(2.3)
(0.575, 2)
```

MATH.FSUM(SEQ)

This calculates and returns an accurate floating-point sum of values in the iterable seq. It assumes IEEE-754 floating-point arithmetic. It is a lossless sum.

https://doi.org/10.15215/remix/9781998944088.01

```
>>> math.fsum([2.3, 2, 53454, 6.71232])
53465.01232
```

MATH.GAMMA(X)

This returns the value of the gamma function at x.

```
>>> math.gamma(2.3)
1.16671190519816
```

MATH.GCD(X, Y)

This calculates and returns the greatest common divisor of x and y.

```
>>> math.gcd(222, 780)
6
```

MATH.HYPOT(X, Y)

This calculates and returns the Euclidean distance—that is, the value of sqrt(x * x + y * y).

```
>>> math.hypot(3, 4)
5.0
```

MATH.ISCLOSE(A, B, *, REL_TOL = 1E-09, ABS_TOL = 0.0)

Determine whether two floating-point numbers are close in value. The rel_tol argument sets the maximum difference for being considered "close" relative to the magnitude of the input values, whereas abs_tol argument sets the maximum difference for being considered "close" regardless of the magnitude of the input values. Return True if a is close in value to b, and False otherwise. For the values to be considered close, the difference between them must be smaller than at least one of the tolerances set by rel_tol and abs_tol. -inf, inf, and NaN behave similarly to the IEEE 754 Standard. That is, NaN is not close to anything, even itself, and inf and -inf are only close to themselves.

```
>>> math.isclose(3.5, 3.51)
False
>>> math.isclose(3.5, 3.500000001)
True
>>> math.isclose(3.5, 3.50000001)
False
```

https://doi.org/10.15215/remix/9781998944088.01

MATH.ISFINITE(X)

This returns True if x is neither an infinity nor a NaN, and False otherwise.

```
>>> math.isfinite(3.5)
True
```

MATH.ISINF(X)

This returns True if x is a positive or negative infinity, and False otherwise.

```
>>> math.isinf(3.5)
False
```

MATH.ISNAN(X)

This returns True if x is a NaN (not a number), and False otherwise.

```
>>> math.isnan(3.5)
False
```

MATH.ISQRT(N)

This returns the integer square root of the nonnegative integer n, which is the floor of the exact square root of n, or equivalently the greatest integer is such that $a^2 \leq n$. This function is only available in Python 3.8.0 or later.

```
>>> math.isqrt(43)
6
```

MATH.LDEXP(X, I)

This calculates and returns x* (2 ** i). The function is essentially the inverse of frexp().

```
>>> math.ldexp(3, 12)
12288.0
```

MATH.LGAMMA(X)

This calculates and returns the natural logarithm of the absolute value of the gamma function at x.

```
>>> math.lgamma(3)
0.693147180559945
```

https://doi.org/10.15215/remix/9781998944088.01

MATH.LOG(X, BASE = MATH.E)

This calculates and returns the logarithm of x to the given base. If the base is not specified, it returns the natural logarithm (base-e) of x.

```
>>> math.log(3)
1.0986122886681098
>>> math.log(3,3)
1.0
>>> math.log(3,5)
0.6826061944859854
```

MATH.LOG10(X)

This calculates and returns the base-10 logarithm of x.

```
>>> math.log10(3)
0.47712125471966244
```

MATH.LOG1P(X)

This calculates and returns the natural logarithm of 1 + x (base-e). The result is computed in a way that is accurate for x near 0.

```
>>> math.log1p(3)
1.3862943611198906
```

MATH.LOG2(X)

This calculates and returns the base-2 logarithm of x.

```
>>> math.log2(3)
1.584962500721156
```

MATH.MODF(X)

This calculates and returns the fractional and integer parts of x. Both results carry the sign of x and are floats.

```
>>> math.modf(32.6)
(0.6000000000000014, 32.0)
```

MATH.POW(X, Y)

This calculates and returns x ** y (x to the power of y).

https://doi.org/10.15215/remix/9781998944088.01

```
>>> math.pow(32,6)
1073741824.0
```

MATH.RADIANS(X)

This converts angle x from degrees to radians and returns the result.

```
>>> math.radians(32)
0.5585053606381855
```

MATH.REMAINDER(X, Y)

This calculates and returns the difference between x and the closest integer multiple of y, which is x − n * y, where n * y is the closest integer multiple of y. In the case where x is exactly halfway between two multiples of y, the nearest even value of n is used. The result is always exact.

```
>>> math.remainder(32,7)
-3.0
>>> math.remainder(30,7)
2.0
>>> math.remainder(31,7)
3.0
```

MATH.SIN(X)

This calculates and returns the sine of x (measured in radians).

```
>>> math.sin(0.31)
0.3050586364434435
```

MATH.SINH(X)

This calculates and returns the hyperbolic sine of x.

```
>>> math.sinh(31)
14524424832623.713
```

MATH.SQRT(X)

This calculates and returns the square root of x.

```
>>> math.sqrt(31)
5.5677643628300215
```

https://doi.org/10.15215/remix/9781998944088.01

MATH.TAN(X)

This calculates and returns the tangent of x (measured in radians).

```
>>> math.tan(31)
-0.441695568020698
```

MATH.TANH(X)

This calculates and returns the hyperbolic tangent of x.

```
>>> math.tanh(31)
1.0
```

MATH.TRUNC(X)

This truncates the float number x to the nearest Integral toward 0. It uses the __trunc__ magic method.

```
>>> math.trunc(3.561)
3
```

In addition, the module also defines the following constants used in math:

```
math.e = 2.718281828459045
math.inf = inf
math.nan = nan
math.pi = 3.141592653589793
math.tau = 6.283185307179586
```

Note that to use these functions and constants, you will have to use the dot notation shown in the sample code above to indicate that it is from the math module.

If you only need to use one or some functions from the math module, you may import the particular functions from it to save computer memory and use the functions without using the dot notation. The following is an example, generating a table of square roots for numbers from 1 to 100, in which only the sqrt function has been imported from the math module:

```
In [ ]:  from math import sqrt
         for i in range(100):
           print('{:<5.3}'.format(sqrt(i + 1)), end = ' ')
           if (i + 1) % 10 == 0:
             print('\n')
```

https://doi.org/10.15215/remix/9781998944088.01

```
Out [ ]:  1.0 1.41 1.73 2.0 2.24 2.45 2.65 2.83 3.0 3.16
          3.32 3.46 3.61 3.74 3.87 4.0 4.12 4.24 4.36 4.47
          4.58 4.69 4.8 4.9 5.0 5.1 5.2 5.29 5.39 5.48
          5.57 5.66 5.74 5.83 5.92 6.0 6.08 6.16 6.24 6.32
          6.4 6.48 6.56 6.63 6.71 6.78 6.86 6.93 7.0 7.07
          7.14 7.21 7.28 7.35 7.42 7.48 7.55 7.62 7.68 7.75
          7.81 7.87 7.94 8.0 8.06 8.12 8.19 8.25 8.31 8.37
          8.43 8.49 8.54 8.6 8.66 8.72 8.77 8.83 8.89 8.94
          9.0 9.06 9.11 9.17 9.22 9.27 9.33 9.38 9.43 9.49
          9.54 9.59 9.64 9.7 9.75 9.8 9.85 9.9 9.95 10.0
```

8.6 Modules for Time, Date, and Calendar

Date and time are often used and referred to in many applications. You may also want to include a calendar in an application. Python has modules in its standard distribution that allow you to import these modules right away.

The Datetime Module

The first of these modules is the datetime module that comes in the standard Python library. To use the module, simply import it as shown below:

```
>>> import datetime
```

To find out what is defined and available in the module, run the following dir statement:

```
>>> dir(datetime)

['MAXYEAR', 'MINYEAR', '__builtins__', '__cached__',
'__doc__', '__file__', '__loader__', '__name__',
'__package__', '__spec__', 'date', 'datetime', 'datetime_
CAPI', 'sys', 'time', 'timedelta', 'timezone', 'tzinfo']
```

To further find out what each name is defined for, use the ***help*** statement on each. As usual, we will go through some of the important names defined in the module, with examples.

DATETIME.DATE(<YEAR, MONTH, DAY>)

This is the constructor of the date class defined in the datetime module and used to construct and return a date object for the day of the month of the year.

https://doi.org/10.15215/remix/9781998944088.01

```
>>> d1 = datetime.date(2020, 7, 1)   # create a date
object
>>> d1.year   # get the year attribute of the date object
2020
>>> d1.month   # get the month attribute of the date
object
7
>>> d1.day   # get the day attribute of the date object
1
```

A date object has the following methods defined.

DATETIME.DATE.CTIME()

This returns a ctime()-style string.

```
>>> d1.ctime()
'Wed Jul 1 00:00:00 2020'
```

DATETIME.DATE.ISOCALENDAR()

This returns a three-tuple containing an ISO year, week number of the year, and day number of the week. In the datetime module, Monday is 1, Tuesday is 2,..., Sunday is 7.

```
>>> d1.isocalendar()
(2020, 27, 3)
```

DATETIME.DATE.ISOFORMAT()

This returns a date string in ISO 8601 format, YYYY-MM-DD.

```
>>> d1.isoformat()
'2020-07-01'
```

DATETIME.DATE.ISOWEEKDAY()

This returns an integer from 1 to 7 as the day of the week represented by the date.

```
>>> d1.isoweekday()
3
```

https://doi.org/10.15215/remix/9781998944088.01

DATETIME.DATE.REPLACE(...)

This returns the date with new specified fields.

DATETIME.DATE.STRFTIME(...)

This changes the date format and returns a strftime()-style string.

DATETIME.DATE.TIMETUPLE(...)

This returns a time-tuple that is compatible with time.localtime().

DATETIME.DATE.TOORDINAL(...)

This returns a proleptic Gregorian ordinal. January 1 of year 1 is day 1.

DATETIME.DATE.WEEKDAY(...)

This returns the day of the week represented by the date: Monday is 0...Sunday is 6.

The following are all class methods of the date class defined in the datetime module, which means they can be called from the class name date.

DATETIME.DATE.FROMISOFORMAT(<ISO_DATE_FORMAT STRING>)

This will construct a date object from an ISO date format string, which is YYYY-MM-DD.

```
>>> d2 = datetime.date.fromisoformat('2020-07-01')
>>> d2.ctime()
'Wed Jul 1 00:00:00 2020'
```

DATETIME.DATE.FROMORDINAL(<DAYS IN RELATION TO A PROLEPTIC GREGORIAN ORDINAL>)

This constructs a date object from an integer >= 1 representing the days after the proleptic Gregorian ordinal, which is January 1 of year 1, with ordinal 1.

```
>>> from datetime import date   # after this, you don't
need to have datetime in the reference to date class
>>> d3 = date.fromordinal(1235)
>>> d3.ctime()
'Wed May 19 00:00:00 0004'
```

https://doi.org/10.15215/remix/9781998944088.01

DATETIME.DATE.FROMTIMESTAMP(<TIMESTAMP>)

This constructs a local date object from a POSIX timestamp (a big positive float number), such as returned time.time(), which will be explained shortly.

```
>>> import time
>>> time.time()   # it returns a timestamp for now
1593745988.6121984
>>> date.fromtimestamp(1593745988.6121984)
datetime.date(2020, 7, 2)
```

DATETIME.DATE.TODAY()

This returns a date object of for the current date.

```
>>> from datetime import date
>>> print(date.today())
2020-07-05
```

The datetime module also has a class called time. The following is the constructor of the time class.

DATETIME.TIME(HOUR = 0, MINUTE = 0, SECOND = 0, MICROSECOND = 0, TZINFO = NONE)

This returns a time object. All arguments with 0 as their default value must be in their reasonable range or the program will raise a value error. If no argument is provided, they are all 0, except tzinfo (for time zone information, which needs to be an instance of tzinfo class if given). The default value of tzinfo is None.

```
>>> from datetime import time
>>> print(time(hour = 23))
23:00:00
>>> t1 = time(11, 25, 5)
time is 11:25:05
>>> print(f'time is {t1}')
>>> print(f'hour is {t1.hour}')   # this is to get the
hour of a time
hour is 11
>>> print(f'minute is {t1.minute}')   # this is to get
the minute
minute is 25
```

https://doi.org/10.15215/remix/9781998944088.01

```
>>> print(f'second is {t1.second}')   # this is to get
second
second is 5
```

The following is the only class method of the date class in the datetime module.

TIME.FROMISOFORMAT(...)

This class method will construct a date object from a string passed in the parameter.

```
>> import datetime
>> canada_day_str = "2022-07-01"
>> canada_day_object = datetime.date.
fromisoformat(canada_day_str)
>> print(f"{canada_day_object} as
{type(canada_day_object)}")
2022-07-01 as <class 'datetime.date'>
```

The datetime module also has a class called datetime, which is a combination of date and time. The following is the constructor of the datetime objects.

DATETIME.DATETIME(YEAR, MONTH, DAY, HOUR = 0, MINUTE = 0, SECOND = 0, MICROSECOND = 0, TZINFO = NONE, *, FOLD = 0)

This returns a datetime object for the date and time given in the arguments. If no time is given, the default is the beginning of the day of the month of the year.

The following are the methods defined in the datetime class.

DATETIME.CTIME(...)

This returns a ctime()-style time string.

```
>>> from datetime import datetime
>>> dt1 = datetime.now()
>>> print(f'now it is {dt1.ctime()}')
now it is Mon Jul 6 14:07:01 2020
```

https://doi.org/10.15215/remix/9781998944088.01

DATETIME.ASTIMEZONE(TZ)

This converts to the local time with the time zone set to <tz> or local.

```
>>> from datetime import datetime
>>> dt1 = datetime.now()
>>> print(dt1)
2020-07-06 14:07:01.046202
>>> print(dt1.astimezone())
2020-07-06 14:07:01.046202-06:00
>>> import pytz
>>> asiachina=pytz.timezone('Asia/Chongqing')
>>> print(dt1)
2020-07-06 14:07:01.046202
>>> print(dt1.astimezone(asiachina)) # print time in
China
2020-07-07 04:07:01.046202+08:00
```

A complete list of time zone names can be found at https://en.wikipedia.org/wiki/List_of_tz_database_time_zones.

DATETIME.DATE()

This returns a date object of the date portion of the datetime object with the same year, month, and day.

```
>>> from datetime import datetime
>>> tm1 = datetime.now()
>>> print(f'now is {dtm1.ctime()}')
now is Tue Jul 7 08:50:25 2020
>>> dt1 = dtm1.date()
>>> print(f'the date is {dt1}')
the date is 2020-07-07
```

DATETIME.DST()

This returns the DST (daylight saving time) status of a given tzinfo.

```
>>> print(f'the date is {dt1}')
the date is 2020-07-07
>>> print(f'the dst status is {dtm1.dst()}')
None
```

https://doi.org/10.15215/remix/9781998944088.01

DATETIME.ISOFORMAT(SEP = 'T')

This returns a date and time string in ISO 8601 format, YYYY-MM-DDT[HH[:MM[:SS[.mmm[uuu]]]]][+HH:MM]. sep is a single character used to separate the year from the time, and defaults to T. timespec specifies what components of the time to include. The allowed values include the following: auto, hours, minutes, seconds, milliseconds, and microseconds.

```
>>> from datetime import datetime
>>> dt1 = datetime.now()
>>> print(dt1.isoformat(sep='@'))
2023-03-13@18:51:29.324588
```

DATETIME.REPLACE(<FIELD>=<VALUE>)

This returns a datetime object with the named field(s) replaced.

```
>>> dt2 = dt1.replace(year=2025)
>>> print(f'new datetime becomes {dt}')
new datetime becomes 2021-07-07 09:00:37.138388
```

DATETIME.TIME()

This returns a time object for the time portion of the datetime object but with tzinfo = None.

```
>>> dt1 = dtm1.date()
>>> tm1=dtm1.time()
>>> print(f'the date is {dt1}')
the date is 2020-07-07
>>> print(f'the time is {tm1}')
the time is 09:17:06.055195
```

DATETIME.TIMESTAMP()

This returns POSIX timestamp as a float number.

```
>>> tmstamp = dtm1.timestamp()
>>> print(f'The timestamp of {dtm1} is {tmstamp}')
```

DATETIME.TIMETUPLE()

This returns a time-tuple compatible with time.localtime().

```
>>> tmtuple = dtm1.timetuple()
>>> print(f'The time-tuple of {dtm1} is {tmtuple}')
```

https://doi.org/10.15215/remix/9781998944088.01

DATETIME.TIMETZ()

This returns a time object with the same time and tzinfo. Note the difference between timetz() and time().

```
>>> tminfo = dtm1.timetz()
>>> print(f'The time of {dtm1} is {tminfo}')
The timezone info of 2020-07-07 09:24:39.517213 is
09:24:39.517213
```

DATETIME.TZNAME(...)

This returns the tzname of tzinfo.

```
>>> import pytz
>>> tz = pytz.timezone('Canada/Mountain')
>>> dtm = datetime.fromisoformat('2020-07-05T21:05:33')
>>> ndtm = dtm.replace(tzinfo = tz)
>>> tmzname = ndtm.tzname()
>>> print(f'The timezone for {ndtm} is {tmzname}')
The timezone for 2020-07-05 21:05:33-07:34 is LMT
```

DATETIME.UTCOFFSET(...)

This returns utcoffset of tzinfo.

```
>>> tmzutcoffset = ndtm.utcoffset()
>>> print(f'The timezone utc offset of {ndtm} is
{tmzutcoffset}')
The timezone utc offset of 2020-07-05 21:05:33-07:34 is
-1 day, 16:26:00
```

The following are some class methods defined in the datetime class.

DATETIME.COMBINE(DT, TM)

This combines the date dt and time tm into a datetime object and returns the datetime object.

```
>>> dt = datetime.date.today()
>>> tm = datetime.time(20,59,12)
>>> dtm = datetime.datetime.combine(dt, tm)
>>> print(f'date is {dt}, time is {tm}, datetime is {dtm}')
```

https://doi.org/10.15215/remix/9781998944088.01

```
date is 2020-07-05, time is 20:59:12, datetime is
2020-07-05 20:59:12
```

DATETIME.FROMISOFORMAT(DTMSTR)

This constructs the datetime object from a date and time string in ISO format and returns the converted datetimeobject. Remember that the ISO time string format is YYYY-MM-DDTHH:MM:SS:mmm:uuu.

```
>>> dtm = datetime.datetime.fromisoformat('2020-07
-05T21:05:33')
>>> print(dtm)
2020-07-05 21:05:33
```

DATETIME.FROMTIMESTAMP(…)

This constructs a datetime object from a POSIX timestamp.

```
>>> tmstamp1 = ndtm.timestamp()
>>> print(f'The time stamp of {ndtm} is {tmstamp1}')
The time stamp of 2020-07-05 21:05:33-07:34 is
1594010373.0
>>> redtmobj = datetime.fromtimestamp(tmstamp1)
>>> print(f'It is a different object and the time value
has changed to {redtmobj}')
It is a different object, and the time value has changed
to 2020-07-05 22:39:33.
```

DATETIME.NOW(TZ = NONE)

This returns a datetime object representing the current time local to tz, which should be a Timezone object if given. If no tz is specified, the local timezone is used.

```
>>> from datetime import datetime   # import the datetime
class from the datetime module
>>> dt1 = datetime.now()
>>> print(f'it is {dt1}')
it is 2020-07-05 11:22:48.876825
```

DATETIME.STRPTIME(<DATE_STRING, FORMAT>)

This returns a datetime object by parsing a date_string, based on a given format.

https://doi.org/10.15215/remix/9781998944088.01

```
>>> dtstring = "7 July, 2020"
>>> dtobj = datetime.strptime(dtstring, "%d %B, %Y")   #
note the date formatting string
>>> print("date object = ", dtobj)

date object = 2020-07-07 00:00:00
```

DATETIME.TODAY()

This returns a datetime object for today.

```
>>> dt2 = datetime.today()
>>> print(f'Today is {dt2}')
Today is 11:34:09.228618
```

DATETIME.UTCFROMTIMESTAMP()

This constructs a naive UTC datetime from a POSIX timestamp.

```
>>> redtmobj = datetime.utcfromtimestamp(tmstamp1)
>>> print(f'{redtmobj} is a UTC datetime from a POSIX
timestamp {redtmobj}')
2020-07-06 04:39:33 is a UTC datetime from a POSIX
timestamp 1594010373.0
```

DATETIME.UTCNOW()

This returns a new datetime representing the UTC day and time.

```
>>> dt2 = datetime.today()
>>> dt3 = datetime.utcnow()
>>> print(f'Today is {dt2}, and the UTC time is {dt3}')
Today is 2020-07-07 11:37:02.862356, and the UTC time is
2020-07-07 17:37:02.862356
```

Sometimes, you need to deal with time intervals such as how long has passed since the last time you saw your best friend. That is what the timedelta class is defined for in the datetime module. The following is the constructor of the class.

DATETIME.TIMEDELTA(DAYS = 0, SECONDS = 0, MICROSECONDS = 0, MILLISECONDS = 0, MINUTES = 0, HOURS = 0, WEEKS = 0)

This constructs and returns a timedelta object. Note that all arguments are optional, and all default to 0 if not provided.

https://doi.org/10.15215/remix/9781998944088.01

```
>>> from datetime import timedelta
>>> ndlt = timedelta(days = 31)
```

TIMEDELTA.TOTAL_SECONDS(…)

This returns the total number of seconds in the duration.

```
>>> print(f'the total number of seconds in 31 days is
{ndlt.total_seconds()}')
the total number of seconds in 31 days is 2678400.0
>>> ndlt = timedelta(31, 25, hours = -3)
>>> print(f'the total number of seconds in 31 days is
{ndlt.total_seconds()}')
the total number of seconds in 31 days and 25 seconds
minus 3 hours is 2667625.0
```

The Time Module

The second of these modules is the time module. It comes with the standard Python distribution, so there is no need for you to install anything in particular. This module can be imported and used directly within your program when needed. The following statements get us a list of the names defined in the module:

```
>>> import time
>>> dir(time)
['_STRUCT_TM_ITEMS', '__doc__', '__loader__', '__
name__', '__package__', '__spec__', 'altzone',
'asctime', 'clock', 'ctime', 'daylight', 'get_clock_
info', 'gmtime', 'localtime', 'mktime', 'monotonic',
'monotonic_ns', 'perf_counter', 'perf_counter_ns',
'process_time', 'process_time_ns', 'sleep',
'strftime', 'strptime', 'struct_time', 'thread_time',
'thread_time_ns', 'time', 'time_ns', 'timezone',
'tzname']
```

In the following, we explain the names, including attributes and functions, available in the time module on a Windows platform. The code samples are all shown as they would appear in a Python interactive shell. If you wish to see all functions defined in the time module, please read the documentation at https://docs.python.org/3/library/time.html.

https://doi.org/10.15215/remix/9781998944088.01

TIME.ALTZONE

This is an attribute that contains the offset of the local DST timezone in seconds west of UTC, if one is defined. The value is negative if the local DST timezone is east of UTC (as in Western Europe, including the UK). Only use this if daylight is nonzero.

```
>>> import time
>>> print("local time zone is %d " % (time.altzone/3600))
local time zone is 6
```

TIME.ASCTIME(TUPLETIME)

This accepts a time-tuple and returns a readable 24-character string such as Tue Dec 11 18:07:14 2008. A time-tuple has nine elements, as returned by gmtime() or localtime().

```
>>> import time   # you only need to import a module
once, so this is just in case
>>> print("local time is %s " % (time.asctime()))
local time is Tue Nov 12 15:10:50 2019
>>> time.asctime(tuple(time.localtime()))
'Tue Nov 12 15:24:05 2019'
```

TIME.CLOCK()

This returns a floating-point number for the CPU time or real time since the start of the process or since the first call to clock(). It is very useful, especially when measuring the computational cost of a code block.

```
>>> time.clock()
428446.1717301
```

TIME.CTIME([SECS])

This returns a time in seconds since the epoch to a string in the local time. Remember that the argument in [] is optional.

This has the same result as asctime(localtime(secs)), and simply a call of asctime(), which will use the current local time in seconds.

```
>>> time.asctime()
'Tue Nov 12 15:24:49 2019'
>>> time.ctime()
'Tue Nov 12 15:24:55 2019'
```

https://doi.org/10.15215/remix/9781998944088.01

TIME.GET_CLOCK_INFO(NAME)

This returns information on the specified clock as a namespace object. Supported clock names and the corresponding functions to read their value are the following: monotonic, perf_counter, process_time, thread_time, and time.

```
>>> time.get_clock_info('monotonic')
namespace(adjustable = False,
implementation='GetTickCount64()', monotonic=True,
resolution = 0.015625)
>>> time.get_clock_info('time')
namespace(adjustable = True, implementation =
'GetSystemTimeAsFileTime()', monotonic=False,
resolution=0.015625)
```

TIME.GMTIME([SECS])

This accepts an instant expressed in seconds since the epoch and returns a time-tuple t with the UTC time. *Note:* t.tm_isdst is always 0.

```
>>> time.gmtime()
time.struct_time(tm_year = 2019, tm_mon = 11, tm_mday =
12, tm_hour = 22, tm_min = 21, tm_sec = 31, tm_wday = 1,
tm_yday = 316, tm_isdst = 0)
>>> tuple(time.gmtime())
(2019, 11, 12, 22, 22, 2, 1, 316, 0)
```

TIME.LOCALTIME([SECS])

This accepts an instant expressed in seconds since the epoch and returns a time-tuple t with the local time (t.tm_isdst is 0 or 1, depending on whether DST applies to instant secs by local rules).

```
>>> time.localtime()
time.struct_time(tm_year = 2019, tm_mon = 11, tm_mday =
12, tm_hour = 15, tm_min = 19, tm_sec = 0, tm_wday = 1,
tm_yday = 316, tm_isdst = 0)
>>> tuple(time.localtime())
(2019, 11, 12, 15, 22, 32, 1, 316, 0)
```

TIME.MKTIME(TUPLETIME)

This accepts a time instant expressed as a time-tuple in the local time and returns a floating-point value, with the instant expressed in seconds since the epoch.

https://doi.org/10.15215/remix/9781998944088.01

```
>>> time.mktime((2019, 11, 12, 22, 22, 2, 1, 316, 0))
1573622522.0
```

TIME.MONOTONIC()

This returns the value of a monotonic clock as a float number, the number of seconds since the previous call. The clock is not affected by system clock updates. The reference point of the returned value is undefined, so that only the difference between the results of consecutive calls is valid.

```
>>> time.monotonic()
1557979.093
```

TIME.MONOTONIC_NS()

This is similar to monotonic() but returns time as nanoseconds.

```
>>> time.monotonic_ns()
1557954406000000
```

TIME.PERF_COUNTER()

This returns the value of a performance counter as a float number since the previous call. A performance counter is a clock with the highest available resolution to measure a short duration. It includes time elapsed during sleep and is system-wide. The reference point of the returned value is undefined, so that only the difference between the results of consecutive calls is valid.

```
>>> time.perf_counter()
429437.6389873
```

TIME.PERF_COUNTER_NS()

This is similar to perf_counter() but returns time as nanoseconds.

```
>>> time.perf_counter_ns()
429556266018100
```

TIME.PROCESS_TIME()

This returns the value (in fractional seconds) of the sum of the system and the user CPU time of the current process. It does not include time elapsed during sleep. It is process-wide by definition. The reference point of the returned value

https://doi.org/10.15215/remix/9781998944088.01

is undefined so that only the difference between the results of consecutive calls is valid.

```
>>> time.process_time()
6.71875
```

TIME.PROCESS_TIME_NS()

This is similar to process_time() but returns time as nanoseconds.

```
>>> time.process_time_ns()
6687500000
```

TIME.SLEEP(SECS)

This suspends the calling thread for secs (seconds). It can be used to delay programs.

```
>>> time.sleep(6)    # sleep 6 seconds
```

TIME.STRFTIME(FMT[,TUPLETIME])

This accepts an instant expressed as a time-tuple in the local time and returns a string representing the instant as specified by string fmt.

```
>>> t = (2019, 11, 17, 17, 3, 38, 1, 48, 0)
>>> t = time.mktime(t)
>>> print(time.strftime("%b %d %Y %H:%M:%S", time.
gmtime(t)))
Nov 18 2019 00:03:38
```

TIME.STRPTIME(STRINGTIME[, FMT])

This parses str according to format string fmt and returns the instant in time-tuple format.

```
>>> time.strptime('Tue Nov 12 15:24:05 2019','%a %b %d
%H:%M:%S %Y')
time.struct_time(tm_year = 2019, tm_mon = 11, tm_mday =
12, tm_hour = 15, tm_min = 24, tm_sec = 5, tm_wday = 1,
tm_yday = 316, tm_isdst = -1)
>>> tuple(time.strptime('Tue Nov 12 15:24:05 2019','%a %b
%d %H:%M:%S %Y'))
(2019, 11, 12, 15, 24, 5, 1, 316, -1)
```

https://doi.org/10.15215/remix/9781998944088.01

TIME.TIME()

This returns the current time instant, a floating-point number of seconds since the epoch.

```
>>> time.time()
1573607220.4043384
>>> time.asctime(time.localtime(time.time()))   # it is
the same as time.asctime()
'Mon Jun 8 13:59:35 2020'
>>> time.asctime()
'Mon Jun 8 13:59:45 2020'
```

The Calendar Module

If you prefer a simple and more direct module to handle time and date, you can use the calendar module, as detailed below.

CALENDAR.CALENDAR(YEAR, W = 2, L = 1, C = 6)

This returns a formatted calendar for *year*—a multiline string formatted into three columns separated by c spaces. w is the width in characters of each date; each line has length 21 * w + 18 + 2 * c. l is the number of lines for each week.

```
>>> import calendar as cl
>>> print(cl.calendar(2021))
                                  2021

      January                 February                March
Mo Tu We Th Fr Sa Su    Mo Tu We Th Fr Sa Su    Mo Tu We Th Fr Sa Su
             1  2  3     1  2  3  4  5  6  7     1  2  3  4  5  6  7
 4  5  6  7  8  9 10     8  9 10 11 12 13 14     8  9 10 11 12 13 14
11 12 13 14 15 16 17    15 16 17 18 19 20 21    15 16 17 18 19 20 21
18 19 20 21 22 23 24    22 23 24 25 26 27 28    22 23 24 25 26 27 28
25 26 27 28 29 30 31                            29 30 31


      April                   May                     June
Mo Tu We Th Fr Sa Su    Mo Tu We Th Fr Sa Su    Mo Tu We Th Fr Sa Su
          1  2  3  4                   1  2        1  2  3  4  5  6
 5  6  7  8  9 10 11     3  4  5  6  7  8  9     7  8  9 10 11 12 13
12 13 14 15 16 17 18    10 11 12 13 14 15 16    14 15 16 17 18 19 20
19 20 21 22 23 24 25    17 18 19 20 21 22 23    21 22 23 24 25 26 27
26 27 28 29 30          24 25 26 27 28 29 30    28 29 30
                        31
```

https://doi.org/10.15215/remix/9781998944088.01

```
      July                 August               September
Mo Tu We Th Fr Sa Su   Mo Tu We Th Fr Sa Su   Mo Tu We Th Fr Sa Su
          1  2  3  4                      1            1  2  3  4  5
 5  6  7  8  9 10 11    2  3  4  5  6  7  8    6  7  8  9 10 11 12
12 13 14 15 16 17 18    9 10 11 12 13 14 15   13 14 15 16 17 18 19
19 20 21 22 23 24 25   16 17 18 19 20 21 22   20 21 22 23 24 25 26
26 27 28 29 30 31      23 24 25 26 27 28 29   27 28 29 30
                       30 31

      October              November             December
Mo Tu We Th Fr Sa Su   Mo Tu We Th Fr Sa Su   Mo Tu We Th Fr Sa Su
             1  2  3    1  2  3  4  5  6  7            1  2  3  4  5
 4  5  6  7  8  9 10    8  9 10 11 12 13 14    6  7  8  9 10 11 12
11 12 13 14 15 16 17   15 16 17 18 19 20 21   13 14 15 16 17 18 19
18 19 20 21 22 23 24   22 23 24 25 26 27 28   20 21 22 23 24 25 26
25 26 27 28 29 30 31   29 30                  27 28 29 30 31
```

CALENDAR.FIRSTWEEKDAY()

This returns an integer that is the current setting for the weekday that starts each week. By default, when the calendar module is first imported, it is 0 for Monday.

```
>>> import calendar as cl
>>> cl.firstweekday()
0
```

CALENDAR.ISLEAP(YEAR)

This tests if a year is a leap year. It returns True if it is; it returns False otherwise.

```
>>> cl.isleap(2022)
False
```

CALENDAR.LEAPDAYS(Y1, Y2)

This returns the total number of leap days in the years within range(y1, y2).

```
>>> cl.leapdays(2020, 2030)
3
```

CALENDAR.MONTH(YEAR, MONTH, W = 2, L = 1)

This returns a multiline string with a calendar for *month* of *year*, one line per week plus two header lines. w is the width in characters of each date; each line has length 7 * w + 6. l is the number of lines for each week.

https://doi.org/10.15215/remix/9781998944088.01

```
>>> print(cl.month(2021, 3))
     March 2021
Mo Tu We Th Fr Sa Su
 1  2  3  4  5  6  7
 8  9 10 11 12 13 14
15 16 17 18 19 20 21
22 23 24 25 26 27 28
29 30 31
```

CALENDAR.MONTHCALENDAR(YEAR, MONTH)

This returns a list of sublists of integers. Each sublist denotes a week starting from Monday. Days outside *month* of *year* are set to 0; days within the month are set to their day-of-month, 1 and up. The result as a list of sublists can be conveniently used in applications. For example, you can easily tell what date it is for Monday of the third week of a month.

```
>>> print(cl.monthcalendar(2020, 6))
[[1, 2, 3, 4, 5, 6, 7], [8, 9, 10, 11, 12, 13, 14], [15, 16, 17, 18,
19, 20, 21], [22, 23, 24, 25, 26, 27, 28], [29, 30, 0, 0, 0, 0, 0]]
```

CALENDAR.MONTHRANGE(YEAR, MONTH)

This returns two integers. The first one is the code of the weekday for the first day of the *month* in *year*; the second one is the number of days in the month. Weekday codes are 0 (Monday) to 6 (Sunday); month numbers are 1 (January) to 12 (December). This is useful if you want to print a calendar that begins on a specific day of the week.

```
>>> print(cl.monthrange(2020, 6))
(0, 30)
```

CALENDAR.PRCAL(YEAR, W = 2, L = 1, C = 6)

This prints a well-formatted calendar of a given year. It is the same as calendar.calendar(year, w, l, c). Remember that w is the width of each date in number of characters and l is the number of lines for each week.

```
>>> cl.prcal(2020, w = 2, l = 1, c = 6)
                                  2020

      January                February                 March

Mo Tu We Th Fr Sa Su    Mo Tu We Th Fr Sa Su    Mo Tu We Th Fr Sa Su
       1  2  3  4  5                    1  2                       1
 6  7  8  9 10 11 12     3  4  5  6  7  8  9     2  3  4  5  6  7  8
13 14 15 16 17 18 19    10 11 12 13 14 15 16     9 10 11 12 13 14 15
20 21 22 23 24 25 26    17 18 19 20 21 22 23    16 17 18 19 20 21 22
27 28 29 30 31          24 25 26 27 28 29       23 24 25 26 27 28 29
                                                30 31
```

https://doi.org/10.15215/remix/9781998944088.01

```
      April                    May                      June
Mo Tu We Th Fr Sa Su     Mo Tu We Th Fr Sa Su     Mo Tu We Th Fr Sa Su
       1  2  3  4  5                  1  2  3      1  2  3  4  5  6  7
 6  7  8  9 10 11 12      4  5  6  7  8  9 10      8  9 10 11 12 13 14
13 14 15 16 17 18 19     11 12 13 14 15 16 17     15 16 17 18 19 20 21
20 21 22 23 24 25 26     18 19 20 21 22 23 24     22 23 24 25 26 27 28
27 28 29 30              25 26 27 28 29 30 31     29 30

      July                     August                   September
Mo Tu We Th Fr Sa Su     Mo Tu We Th Fr Sa Su     Mo Tu We Th Fr Sa Su
       1  2  3  4  5                     1  2         1  2  3  4  5  6
 6  7  8  9 10 11 12      3  4  5  6  7  8  9      7  8  9 10 11 12 13
13 14 15 16 17 18 19     10 11 12 13 14 15 16     14 15 16 17 18 19 20
20 21 22 23 24 25 26     17 18 19 20 21 22 23     21 22 23 24 25 26 27
27 28 29 30 31           24 25 26 27 28 29 30     28 29 30
                         31

      October                  November                 December
Mo Tu We Th Fr Sa Su     Mo Tu We Th Fr Sa Su     Mo Tu We Th Fr Sa Su
          1  2  3  4                        1         1  2  3  4  5  6
 5  6  7  8  9 10 11      2  3  4  5  6  7  8      7  8  9 10 11 12 13
12 13 14 15 16 17 18      9 10 11 12 13 14 15     14 15 16 17 18 19 20
19 20 21 22 23 24 25     16 17 18 19 20 21 22     21 22 23 24 25 26 27
26 27 28 29 30 31        23 24 25 26 27 28 29     28 29 30 31
                         30
```

CALENDAR.PRMONTH(YEAR, MONTH, W = 2, L = 1)

This prints a well-formatted calendar month, the same as the one created by calendar.month(year, month, w, l).

```
>>> cl.prmonth(2020, 6, w = 2, l = 1)
      June 2020
  Mo Tu We Th Fr Sa Su
   1  2  3  4  5  6  7
   8  9 10 11 12 13 14
  15 16 17 18 19 20 21
  22 23 24 25 26 27 28
  29 30
```

https://doi.org/10.15215/remix/9781998944088.01

CALENDAR.SETFIRSTWEEKDAY(WEEKDAY)

This sets the first day of each week. Weekday codes are 0 (Monday by default) to 6 (Sunday by default), so if you change this, you will see the days in the calendar shift.

```
>>> cl.setfirstweekday(6)   # set to start from Sunday
>>> cl.prmonth(2020, 6, w = 2, l = 1)
       June 2020
  Su Mo Tu We Th Fr Sa
      1  2  3  4  5  6
   7  8  9 10 11 12 13
  14 15 16 17 18 19 20
  21 22 23 24 25 26 27
  28 29 30
```

CALENDAR.TIMEGM(TUPLETIME)

This is the inverse of time.gmtime. It accepts a time instant in time-tuple form and returns the same instant as a floating-point number of seconds since the epoch.

```
>>> cl.timegm((2020, 6, 19, 11, 35, 56))
1592566556
```

CALENDAR.WEEKDAY(YEAR, MONTH, DAY)

This returns the weekday code for the given date. Weekday codes are 0 (Monday) to 6 (Sunday); month numbers are 1 (January) to 12 (December).

```
>>> import calendar as cl
>>> cl.weekday(2020, 6, 19)   # it is a Friday
4
```

Our last example in Jupyter Lab is to display a calendar for March of 1961.

```
import calendar
cld = calendar.month(1961, 3)
print(cld)
       March 1961
  Mo Tu We Th Fr Sa Su
         1  2  3  4  5
   6  7  8  9 10 11 12
  13 14 15 16 17 18 19
  20 21 22 23 24 25 26
  27 28 29 30 31
```

https://doi.org/10.15215/remix/9781998944088.01

With the calendar module, you will be able to produce a calendar of any year you want.

8.7 Modules for Data Representation and Exchange

JavaScript Object Notation (JSON) is a lightweight data interchange format widely used today. JSON can be used to represent different types of data, though the most used are objects or associate arrays made of key-value pairs. When used for data interchanges between applications, JSON data are represented in a string so that they can be stored, transmitted, and parsed by different applications.

Python has a built-in module called json to handle JSON data. The following two statements will get us a list of the few names defined in the json module:

```
>>> import json
>>> dir(json)
['JSONDecodeError', 'JSONDecoder', 'JSONEncoder', '__all__',
'__author__', '__builtins__', '__cached__', '__doc__',
'__file__', '__loader__', '__name__', '__package__',
'__path__', '__spec__', '__version__', '_default_decoder',
'_default_encoder', 'codecs', 'decoder', 'detect_encoding',
'dump', 'dumps', 'encoder', 'load', 'loads', 'scanner']
```

In the following, we explain two pairs of important functions provided in the json module. The first pair of functions is used to convert JSON data to Python data, which is called deserialization or decoding. The other is used to convert Python data to JSON data, which is called serialization.

JSON.LOADS(S, *, ENCODING=NONE, CLS=NONE, OBJECT_HOOK=NONE, PARSE_FLOAT=NONE, PARSE_INT=NONE, PARSE_CONSTANT=NONE, OBJECT_PAIRS_HOOK=NONE, **KW)

This deserializes json data in s, which can be a str, byte, or bytearray instance containing a JSON document and returns a Python object of s. Or, in plainer terms, it converts a string of JSON data into a Python object and returns the converted Python object.

```
>>> import json
>>> sj = '{"firstname": "Jone", "lastname": "Doe"}'
>>> pdict = json.loads(sj)
>>> print(pdict)

{'firstname': 'Jone', 'lastname': 'Doe'}
```

https://doi.org/10.15215/remix/9781998944088.01

JSON.LOAD(FP, *, CLS = NONE, OBJECT_HOOK = NONE, PARSE_FLOAT=NONE, PARSE_INT = NONE, PARSE_CONSTANT = NONE, OBJECT_PAIRS_HOOK = NONE, **KW)

This deserializes data in a file object or file-like object, such as a socket, referred to as fp that contains a JSON document, making it a Python object. Note the difference between loads and load function: loads convert JSON data in a string, whereas load converts JSON data in a file.

```
>>> import json
>>> from io import StringIO
>>> sio = StringIO('{"firstname": "Jone", "lastname":
"Doe"}')
>>> json.load(sio)
{'firstname': 'Jone', 'lastname': 'Doe'}
```

The next pair of functions are used to convert Python objects into JSON data. This process is called serialization or encoding.

JSON.DUMPS(OBJ, *, SKIPKEYS=FALSE, ENSURE_ASCII=TRUE, CHECK_CIRCULAR=TRUE, ALLOW_NAN = TRUE, CLS = NONE, INDENT = NONE, SEPARATORS = NONE, DEFAULT = NONE, SORT_KEYS=FALSE, **KW)

This serializes a Python object obj and returns a JSON-formatted string of the object.

```
>>> print(pdict)
{'firstname': 'Jone', 'lastname': 'Doe'}
>>> js = json.dumps(pdict)
>>> print(js)   # please note the double quotation marks
used in JSON-formatted data
{"firstname": "Jone", "lastname": "Doe"}
```

JSON.DUMP(OBJ, FP, *, SKIPKEYS = FALSE, ENSURE_ASCII = TRUE, CHECK_CIRCULAR = TRUE, ALLOW_NAN = TRUE, CLS = NONE, INDENT = NONE, SEPARATORS = NONE, DEFAULT = NONE, SORT_KEYS = FALSE, **KW)

This serializes the Python object obj and writes it as a JSON-formatted stream to a writable file or file-like object, such as a socket, referred to as fp.

https://doi.org/10.15215/remix/9781998944088.01

```
>>> from io import StringIO
>>> io = StringIO()
>>> json.dump({'firstname': 'Jone', 'lastname': 'Doe'}, io)
>>> io.getvalue()
'{"firstname": "Jone", "lastname": "Doe"}'
```

Conversions can be made between JSON and Python on different types of data. Table 8-2 shows those conversions.

Table 8-2: Conversion between JSON and Python data

Python	Sample Python data	JSON	Sample JSON data
dict	`{'firstname': 'Jone', 'lastname': 'Doe'}`	Object	`{"firstname": "Jone", "lastname": "Doe"}`
list	`['James', 'Jone', 'Smith', 'Doe']`	Array	`["James", "Jone", "Smith", "Doe"]`
tuple	`('James', 'Jone', 'Smith', 'Doe')`	Array	`["James", "Jone", "Smith", "Doe"]`
str	`"James, Jone, Smith, Doe"`	String	`"James, Jone, Smith, Doe"`
int	`98765321`	Number	`98765321`
float	`98765.321`	Number	`98765.321`
True	`True`	true	`true`
False	`False`	false	`false`
None	`None`	null	`null`

As you can see, all basic Python data can be converted/serialized into JSON data, and vice versa. However, trying to serialize multiple Python objects by repeatedly calling dump() using the same file handle will result in an invalid JSON file because when doing deserialization with load() from the file, load has no way to find out the boundary between different JSON data. As such, there will be only one serialized Python object per file.

https://doi.org/10.15215/remix/9781998944088.01

8.8 Modules for Interfacing Operating Systems and Python Interpreter

On a computer, all applications run on top of an operating system (OS) such as Windows, MacOS, or Linux. Therefore, quite often when developing applications, you will need to interact with an OS, file paths, and the Python interpreter. In Python, the built-in os module, path submodule of os, and sys module provide powerful and convenient ways to interact with the OS, the file path, the Python interpreter, and the runtime environment.

OS Module for Interacting with the Operating System

Since it is a built-in module, all you need to do to use the os module is import it, as shown below:

```
import os
```

If you use the dir(os) statement, you can get a rather big list of names defined in the module. Note that because the os module is operating-system dependent, you may get a different list of names available depending on your platform (e.g., Windows or Linux).

The following are some functions provided in the os module. You are encouraged to test these functions with your own examples on your own machine.

OS.ACCESS(PATH, MODE)

This tests if access to path is in mode, which is an integer such as 777 (111111111) representing the global, group, and user's executable, write, and read rights.

```
>>> import os
>>> p = os.path.abspath(".")
>>> p
'd:\\workshop'
>>> os.access(p,763)
True
```

OS.CHDIR(PATH)

This changes the current working directory to path.

```
>>> os.chdir('c:\\workbench')
>>> os.getcwd()
' c:\\workbench'
>>> os.listdir()
['myprimes.txt', ' news_01.txt', ' personal']
```

https://doi.org/10.15215/remix/9781998944088.01

OS.CHMOD(PATH, MODE)

This changes the mode of path to the numeric mode.

```
>>> os.chmod('c:\\workbench', 477)
```

OS.CHOWN(PATH, UID, GID)

This changes the owner and group id of path to the numeric uid and gid. Please note that these operations are more similar to what you would do on a Unix/Linux system, all subject to permission by the operating system.

OS.CLOSE(FD)

This closes the file descriptor fd. A file descriptor is returned by the os.open() function.

```
>>> fd = os.open('/home/james/testing.txt')
>>> os.close(fd)
```

OS.CPU_COUNT()

This returns the number of CPUs in the system; it will return None if the number is indeterminable.

```
>>> os.cpu_count()
8
```

OS.GET_EXEC_PATH(ENV=NONE)

This returns the sequence of directories that will be searched for by the named executable.

```
>>> import os
>>> print(os.get_exec_path())
['/opt/tljh/user/bin', '/usr/local/sbin', '/usr/local/
bin', '/usr/sbin', '/usr/bin', '/sbin', '/bin', '/snap/
bin']
```

OS.GETCWD()

This returns a Unicode string representing the current working directory.

```
>>> import os
>>> print(os.getcwd())
/home/jupyter-kevin
```

https://doi.org/10.15215/remix/9781998944088.01

OS.GETCWDB()

This returns a byte string representing the current working directory.

```
>>> import os
>>> print(os.getcwdb())
b'/home/jupyter-kevin'
```

OS.GETENV(KEY, DEFAULT=NONE)

This returns an environment variable and returns None if it does not exist. The optional second argument can specify an alternate default.

OS.GETLOGIN()

This returns the actual login name.

```
>>> os.getlogin()
kevin
```

OS.LINK(SRC, DST)

This creates a hard link pointing to src named dst.

OS.LISTDIR(PATH)

This returns a list containing the names of the entries in the directory given by path.

```
>>> os.listdir('.')
['backups', 'Pipfile', 'snap']
```

OS.MKDIR(PATH, MODE=511, DIR_FD=NONE)

This function is used to create a directory, the mode argument only used on Unix-like systems, and will be ignored on Windows. The path argument is required; the mode argument is optional and takes an integer representing permission for the path to be created. Argument _dir_fd is a file descriptor referring to a directory that the new directory will be under; the path is not an absolute path. The default value is None.

```
>>> import os
>>> os.mkdir('/home/james/learn_python')
>>> os.listdir('.')
['backups', 'learn_python', 'Pipfile', 'snap']
```

https://doi.org/10.15215/remix/9781998944088.01

OS.MAKEDIRS(PATH, MODE = 511, EXIST_OK = FALSE)

This recursively makes directories. For example, if the path is ./comp218/assignment1, it will first make a directory named comp218 under the current working directory, if it doesn't exist, then make assignment1 under comp218. The optional mode argument is the same as the one in os.mkdir(). The optional exist_ok argument tells if the operation will continue if the leaf directory already exists. The default is False, meaning that a FileExistsError will be raised if the leaf directory already exists.

OS.OPEN(PATH, FLAGS, MODE = 511, *, DIR_FD = NONE)

This opens the file path and sets various flags for low-level IO and returns a file descriptor to be used by other functions in the os module. Argument dir_fd should be a file descriptor open to a directory (if not default None) and can be used to provide a directory that the file path is relative to. The flags argument tells what the path is opened for. It can take one or some of the following values joined with ***or*** |: os.O_RDONLY, os.O_WRONLY, os.O_RDWR, os.O_APPEND, os.O_CREAT, os.O_EXCL¶, os.O_TRUNC. These values are available on both Windows and Unix platforms.

OS.PUTENV(NAME, VALUE)

This changes or adds an environment variable if name doesn't exist yet.

```
>>> os.times()
nt.times_result(user = 4.125, system = 1.890625, children_
user = 0.0, children_system = 0.0, elapsed = 0.0)
```

OS.READ(FD, N)

This reads at most n bytes from file descriptor fd and returns a string containing the bytes read. If the end of the file referred to by fd has been reached, an empty string is returned.

OS.UMASK(MASK)

This sets the current numeric umask to mask and returns the previous umask. umask is used by operating systems to determine the default permission for newly created files.

```
>>> import os
>>> os.umask(666)   # from now on all new files created
will have umask 666 till next change
256
```

https://doi.org/10.15215/remix/9781998944088.01

OS.URANDOM(SIZE)

This returns a bytes object containing random bytes suitable for cryptographic use.

```
>>> os.urandom(5)
b'-\x8e\xeb\xf1\x7f'
```

OS.UTIME(PATH, TIMES = NONE)

This sets the access and modified times of path, such as on a file.

OS.WALK(TOP)

This is a directory-tree generator and returns a walk object. For each directory in the directory tree rooted at top, it will yield a three-tuple (dirpath, dirnames, filenames), in which dirpath is a string, the path to the directory; dirnames is a list of the names of the subdirectories in dirpath; and filenames is a list of the names of the nondirectory files in dirpath.

Note that the names in the lists are just names, with no path components. To get a full path (which begins with top) to a file or directory in dirpath, use os.path.join(dirpath, name).

```
In [ ]:  import os
         # loop through all the directories and files
         for dirName, subdirList, fileList in os.walk('.'):
             print('Found directory: %s' % dirName)
             for fname in fileList:
                 print('\t%s' % fname)
```

```
Out [ ]: Found directory: ./samples
             average marks.py
             brutal attack on cipher.py
             circle-v2.py
         Found directory: ./samples/chapter3
           randomness.py
           regex1.py
           scratch-v3.py
           sieve prime.py
```

OS.WALK(TOP, TOPDOWN = TRUE, ONERROR = NONE, FOLLOWLINKS = FALSE)

This generates the file names in a directory tree by walking the tree either from the top down or from the bottom up. The os.walk function will be very useful in completing one of the projects in the textbook.

https://doi.org/10.15215/remix/9781998944088.01

OS.WRITE(FD, STR)

This writes the string str to the file descriptor fd and returns the number of bytes actually written.

The path Submodule from os for Manipulating File Paths

When dealing with files and file systems, we quite often need to manipulate file paths. For that reason, the os module has a submodule called path. To use the path module, run the following statement:

```
>>> from os import path
```

The path module provides functions for joining and splitting paths, getting information about a path or file such as its size and timestamp, and testing whether a path is a file, a directory, a real path, or just a link.

PATH.ABSPATH(P)

This returns the absolute version of p.

```
>>> path.abspath('.')
'd:\\workshop\\comp218'
```

PATH.BASENAME(P)

This returns the final component of a pathname.

```
>>> os.path.basename(p)
'comp218'
```

PATH.COMMONPATH(PATHS)

This returns the longest common subpath for a given sequence of pathnames.

```
>>> os.path.commonpath(['d:/workshop/comp218','d:/
workshop/comp369'])
'd:\\workshop'
```

PATH.COMMONPREFIX(PATHS)

This returns the longest common leading component of a given list of pathnames.

```
>>> os.path.commonprefix(['d:/workshop/comp218','d:/
workshop/comp369'])
'd:/workshop/comp'
```

https://doi.org/10.15215/remix/9781998944088.01

PATH.DIRNAME(P)

This returns the directory component of a pathname.

```
>>> os.path.dirname('d:/workshop/comp218/test.py')
'd:/workshop/comp218'
```

PATH.EXISTS(P)

This tests whether a path exists. It returns False for broken symbolic links.

```
>>> os.path.exists('d:/workshop/comp218/test.py')
False
```

PATH.EXPANDUSER(P)

This expands ~ and ~user constructs, mostly for Unix/Linux systems. If user or $HOME is unknown, it does nothing.

```
>>> os.path.expanduser('~/workshop/comp218/test.py')
'C:\\Users\\kevin/workshop/comp218/test.py'
```

PATH.EXPANDVARS(P)

This expands shell variables of the forms $var, ${var}, and %var%. Unknown variables will be left unchanged.

PATH.GETATIME(FILENAME)

This returns the time a file was last accessed, as reported by os.stat().

PATH.GETCTIME(FILENAME)

This returns the time a file's metadata was last changed, as reported by os.stat().

PATH.GETMTIME(FILENAME)

This returns the time a file was last modified, as reported by os.stat().

PATH.GETSIZE(FILENAME)

This returns the size of a file, as reported by os.stat().

PATH.ISABS(S)

This tests whether a path is absolute.

https://doi.org/10.15215/remix/9781998944088.01

PATH.ISDIR(P)
PATH._ISDIR(P)

These return True if the pathname refers to an existing directory.

```
>>> from os import path
>>> path.isdir('.')
True
```

PATH.ISFILE(P)

This tests whether a path is a regular file.

```
>>> from os import path
>>> path.isfile('.')
False
```

PATH.ISLINK(P)

This tests whether a path is a symbolic link. It will always return False for Windows prior to 6.0.

PATH.ISMOUNT(P)

This tests whether a path is a mount point (a drive root, the root of a share, or a mounted volume).

PATH.JOIN(P1, P2)

This is used to join two paths or a path with a file.

```
>>> from os import path
>>> fullpath = path.join('/comp218/', 'testfile')
>>> print(fullpath)
/comp218/testfile
```

PATH.LEXISTS(P)

This tests whether a path exists. It will return True for broken symbolic links.

PATH.NORMCASE(S)

This normalizes the case of a pathname. That is, it makes all characters lower case and all slashes backslashes.

PATH.NORMPATH(P)

This normalizes the path, eliminating double slashes, etc.

https://doi.org/10.15215/remix/9781998944088.01

PATH.ABSPATH(P)

This returns the absolute version of a path.

PATH.RELPATH(P, START=NONE)

This returns a relative version of a path.

PATH.SAMEFILE(F1, F2)

This tests whether two pathnames reference the same actual file or directory.

PATH.SAMEOPENFILE(FP1, FP2)

This tests whether two open file objects reference the same file.

PATH.SAMESTAT(S1, S2)

This tests whether two stat buffers reference the same file.

PATH.SPLIT(P)

This splits a pathname and returns tuple (head, tail), where tail is everything after the final slash.

PATH.SPLITDRIVE(P)

This splits a pathname into a drive/UNC sharepoint and relative path specifiers and returns a two-tuple (drive_or_unc, path); either part may be empty.

PATH.SPLITEXT(P)

This splits the extension from a pathname. An extension is everything from the last dot to the end, ignoring leading dots. For some paths without a dot, the extension part will be empty.

The sys Module for Interaction Between the Python and Python Interpreter or Python Virtual Machine (PVM)

The os and path modules we studied above provide programmers with ways to interact with the operating system and to access the underlying interface of the operating system. The sys module we are going to study below allows programs to interact with Python interpreter.

The following are the objects defined in the sys module and maintained by Python interpreter. These objects are put into two groups: dynamic objects and static objects.

The following are the dynamic objects defined in the sys module. *Dynamic* means the values can be changed.

https://doi.org/10.15215/remix/9781998944088.01

SYS.ARGV

This holds command-line arguments; argv[0] is the script pathname if known. The following example shows what happens when we test it in Jupyter Lab:

```
# factorial.py
def fac(n):
    if n == 0:
        return 1
    else:
        return n * fac(n-1)

n = 9

print(f"{n}! = {fac(n)}")

import sys
print(f'argv = {sys.argv}')

python -u "d:\workshop\research\books\COMP218\samples\
factorial.py"
9! = 362880
argv = ['d:\\workshop\\research\\books\\COMP218\\
samples\\factorial.py']
```

SYS.PATH

This holds the module search path; path[0] is the script directory. The sys.path for the above Python program/script will be

```
sys.path = ['d:\\workshop\\research\\books\\COMP218\\
samples', 's:\\python\\python311\\python311.zip', 's:\\
python\\python311\\Lib', 's:\\python\\python311\\
DLLs', 'C:\\Users\\james\\AppData\\Roaming\\Python\\
Python311\\site-packages', 'C:\\Users\\james\\AppData\\
Roaming\\Python\\Python311\\site-packages\\win32', 'C:\\
Users\\james\\AppData\\Roaming\\Python\\Python311\\
site-packages\\win32\\lib', 'C:\\Users\\james\\AppData\\
Roaming\\Python\\Python311\\site-packages\\Pythonwin',
's:\\python\\python311', 's:\\python\\python311\\Lib\\
site-packages']
```

https://doi.org/10.15215/remix/9781998944088.01

SYS.MODULES

This is a dictionary of all loaded modules. It will provide a long list of modules it is using.

SYS.DISPLAYHOOK

This contains an executable object and can be called to show the results in an interactive session.

```
>>> sys.displayhook
<ipykernel.displayhook.ZMQShellDisplayHook at
0x15a70b56b48>
```

SYS.EXCEPTHOOK

This contains an executable object and can be called to handle any uncaught exception other than SystemExit.

SYS.STDIN

This contains the standard input file object; it is used by input().

SYS.STDOUT

It contains the standard output file object; it is used by print().

SYS.STDERR

This contains the standard error object; it is used for error messages.

SYS.LAST_TYPE

This contains the type of the last uncaught exception.

```
>>> sys.last_type
AttributeError
```

SYS.LAST_VALUE

This contains the value of the last uncaught exception.

```
>>> sys.last_value
AttributeError("module 'os' has no attribute 'chroot'")
```

SYS.LAST_TRACEBACK

This contains the traceback of the last uncaught exception.

https://doi.org/10.15215/remix/9781998944088.01

```
>>> sys.last_traceback
<traceback at 0x15a70ca9388>
```

The above three objects are only available in an interactive session after a traceback has been printed.

The next group of objects available from the sys module are called static objects, which means the values do not change for the given Python interpreter being used.

SYS.BUILTIN_MODULE_NAMES

This contains a tuple of built-in module names.

SYS.COPYRIGHT

This contains the copyright notice pertaining to the interpreter in use. sys.copyright in our case will produce the following, as an example:

```
Copyright (c) 2001-2022 Python Software Foundation.
All Rights Reserved.

Copyright (c) 2000 BeOpen.com.
All Rights Reserved.

Copyright (c) 1995-2001 Corporation for National Research
Initiatives.
All Rights Reserved.

Copyright (c) 1991-1995 Stichting Mathematisch Centrum,
Amsterdam.
All Rights Reserved.
```

SYS.EXEC_PREFIX

This contains the prefix used to find the machine-specific Python library.

SYS.EXECUTABLE

This contains the absolute path of the executable binary of the Python interpreter.

https://doi.org/10.15215/remix/9781998944088.01

SYS.FLOAT_INFO

This contains a named tuple with information about the float implementation.

SYS.FLOAT_REPR_STYLE

This contains a string indicating the style of repr() output for floats.

SYS.HASH_INFO

This contains a named tuple with information about the hash algorithm.

```
>>> print(sys.hash_info)
sys.hash_info(width = 64, modulus = 2305843009213693951,
inf = 314159, nan = 0, imag = 1000003, algorithm =
'siphash24', hash_bits = 64, seed_bits = 128, cutoff = 0)
```

SYS.HEXVERSION

This contains version information encoded as a single integer.

SYS.IMPLEMENTATION

This contains Python implementation information.

```
>>> print(sys.implementation)
namespace(cache_tag = 'cpython-37', hexversion =
50792432, name = 'cpython', version =s ys.version_
info(major = 3, minor = 7, micro = 7, releaselevel =
'final', serial = 0))
```

SYS.INT_INFO

This contains a named tuple with information about the int implementation.

```
>>> print(sys.int_info)
sys.int_info(bits_per_digit = 30, sizeof_digit = 4)
```

SYS.MAXSIZE

This contains the largest supported length of containers.

```
>>> print(sys.maxsize)
9223372036854775807
```

SYS.MAXUNICODE

This contains the value of the largest Unicode code point.

https://doi.org/10.15215/remix/9781998944088.01

```
>>> print(sys.maxunicode)
1114111
>>> print(chr(1114111))
⦸
>>> print(chr(1114112))   # this is out of range and will
cause an error
-----------------------------------
ValueError   Traceback (most recent call last)
<ipython-input-81-1965bd6642f9> in <module>
----> 1< print(chr<(1114112))
ValueError: chr() arg not in range(0x110000)
```

SYS.PLATFORM

This contains the platform identifier.

```
>>> print(sys.platform)
win32
```

SYS.PREFIX

This contains the prefix used to find the Python library.

```
>>> print(sys.prefix)
C:\ProgramData\Anaconda3
```

SYS.THREAD_INFO

This contains a named tuple with information about the thread implementation.

```
>>> print(sys.thread_info)
sys.thread_info(name = 'nt', lock = None, version =
None)
```

SYS.VERSION

This contains the version of this interpreter as a string.

SYS.VERSION_INFO

This contains the version information as a named tuple.

SYS.DLLHANDLE

This is the integer handle of the Python DLL (Windows only).

https://doi.org/10.15215/remix/9781998944088.01

SYS.WINVER

This contains the version number of the Python DLL (Windows only).

SYS.__STDIN__

This is the original stdin.

SYS.__STDOUT__

This is the original stdout.

SYS.__STDERR__

This is the original stderr.

SYS.__DISPLAYHOOK__

This is the original displayhook.

SYS.__EXCEPTHOOK__

This is the original excepthook.

The following are functions also defined in the sys module.

SYS.DISPLAYHOOK()

This function prints an object to the screen and saves it in builtins.

SYS.EXCEPTHOOK()

This function prints an exception and its traceback to sys.stderr.

SYS.EXC_INFO()

This function returns thread-safe information about the current exception.

SYS.EXIT()

This function exits the interpreter by raising SystemExit.

SYS.GETDLOPENFLAGS()

This function returns flags to be used for dlopen() calls.

SYS.GETPROFILE()

This function returns the global profiling function.

https://doi.org/10.15215/remix/9781998944088.01

SYS.GETREFCOUNT()

This function returns the reference count for an object.

SYS.GETRECURSIONLIMIT()

This function returns the max recursion depth for the interpreter.

SYS.GETSIZEOF()

This function returns the size of an object in bytes.

```
>>> from datetime import datetime
>>> import sys
>>> dt1 = datetime.now()
>>> print(sys.getsizeof(dt1))
48
```

SYS.GETTRACE()

This function gets the global debug tracing function.

SYS.SETCHECKINTERVAL()

This function controls how often the interpreter checks for events.

SYS.SETDLOPENFLAGS()

This function sets the flags to be used for dlopen() calls.

SYS.SETPROFILE()

This function sets the global profiling function.

SYS.SETRECURSIONLIMIT()

This function sets the max recursion depth for the interpreter.

SYS.SETTRACE()

This function sets the global debug tracing function.

As can be seen, the sys module gives programmers a way to find out information about the Python interpreter and the runtime environment in particular.

https://doi.org/10.15215/remix/9781998944088.01

8.9 Module for Logging Events During Program Runtime

In some applications, sometimes it's necessary or desirable to keep a record of what happened with the program for performance improvement, error debugging, security, and audit purposes. Examples of such events/data include, but are not limited to, how many times a function/method has been called, how long a function call takes, how much memory it used, and so on. In Python, a package called logging is available for logging within its standard distribution.

Due to the variety of purposes stated above, logged events can be classified into the following five categories based on the nature or severity of the events, in the view of the users of the application, as shown in Table 8-3.

Table 8-3: Levels of logging

Category of logs	Description	Numeric value of the level
NOTSET	The level hasn't been set.	0
DEBUG	Events are useful for error debugging. This is the lowest level of severity.	10
INFO	Information can be useful for improving the performance of the application/program or for assurance of security and auditing.	20
WARNING	Something needs checking.	30
ERROR	These errors are often logical and should be checked.	40
CRITICAL	The event is critical for the program to perform correctly and should be checked and resolved. This is highest level of severity.	50

The logging library defines several classes and module-level functions, and it is the latter that you would be using directly in your programs and applications. The basicConfig() function is used to set up the logging file and other parameters for the logger. The logging() function is for logging messages describing events in each category, as shown in the following code sample:

https://doi.org/10.15215/remix/9781998944088.01

```
In [ ]:  import logging
         logging.debug('Everything logged with logging.debug is
          labelled as debug')
         logging.info('Everything logged with logging.info is
          labelled as info')
         logging.warning('Everything logged with logging.warning
          is labelled as warning')
         logging.error('Everything logged with logging.error is
          labelled as error')
         logging.critical('Everything logged with logging.
          critical is labelled as critical')
```

The output of the code above is shown below:

```
Out [ ]: WARNING:root:Everything logged with logging.warning is labelled as warning
         ERROR:root:Everything logged with logging.error is labelled as error
         CRITICAL:root:Everything logged with logging.critical is labelled as critical
```

You may have noted that output from the debug and info logging functions are missing from the output. This is because the default configuration of the logging module only logs events at warning level or higher. To change the default logging level, you can call a function of the logging module named basicConfig(), as shown in the following example:

```
1  import logging
2
3  logging.basicConfig(level=logging.DEBUG,
4  filename='c:\\users\\james\\myapp.log',
5  filemode='w', format='%(name)s - %(levelname)s - %(message)
     s')   # this line belongs to the basicConfig call as well
6  logging.debug('Everything logged with logging.debug is
     labelled as debug')
7  logging.info('Everything logged with logging.info is
     labelled as info')
8  logging.warning('Everything logged with logging.warning is
     labelled as warning')
9  logging.error('Everything logged with logging.error is
     labelled as error')
10 logging.critical('Everything logged with logging.critical
     is labelled as critical')
```

Instead of directly printing out to the terminal, this code writes the logs to a file named myapp.log, and the content of the generated logging file myapp.log is as follows:

https://doi.org/10.15215/remix/9781998944088.01

```
root - DEBUG - Everything logged with logging.debug is labelled as debug
root - INFO - Everything logged with logging.info is labelled as info
root - WARNING - Everything logged with logging.warning is labelled as
warning
root - ERROR - Everything logged with logging.error is labelled as error
root - CRITICAL - Everything logged with logging.critical is labelled
as critical
```

Now as can be seen, DEBUG and INFO are recorded in the logging file because we changed the logging level to DEBUG.

Please note that for logging configuration to take full effect, all should be configured in a single basicConfig() method call. If the statement becomes too long to fit in a single line, the statement can take two or more lines, as long as newline is not within the string or word, as shown above.

In the basicConfig() function call shown above, keyword arguments are used to set the level of logging to DEBUG, the logging file name to c:\users\james\myapp.log, and the log file mode to w for write, which means that everything in the log file will be overwritten by new logging messages. If you want to keep the old logs and add the new logs to the end of the old logs, you need to set the file mode to a for append, which is the default set by the logging mode.

It has been noted that the basicConfig() function for logging is not fully functional within Jupyter Notebook. To change the logging level within Jupyter Notebook, you can use the logging.getLogger().setLevel() method. However, you cannot set the logging file name and logging file mode within Jupyter Notebook.

8.10 Modules for Playing and Manipulating Audio and Video Files

This section covers how to develop sound- and music-related applications with Python.

winsound

To play WAV files in your Windows applications, you can use the winsound module included in the standard Python distribution. You can import the module and use the functions defined in it without installing the module. Using the following statements, you can get a list of names defined in the module:

```
>>> import winsound
>>> dir(winsound)
['Beep', 'MB_ICONASTERISK', 'MB_ICONEXCLAMATION', 'MB_
ICONHAND', 'MB_ICONQUESTION', 'MB_OK', 'MessageBeep',
```

https://doi.org/10.15215/remix/9781998944088.01

```
'PlaySound', 'SND_ALIAS', 'SND_APPLICATION', 'SND_
ASYNC', 'SND_FILENAME', 'SND_LOOP', 'SND_MEMORY',
'SND_NODEFAULT', 'SND_NOSTOP', 'SND_NOWAIT', 'SND_PURGE',
'__doc__', '__file__', '__loader__', '__name__', '__
package__', '__spec__']
```

For more details about the module and functionalities provided, run help(winsound) in Python interactive mode, as shown below:

```
>>> import winsound
>>> help(winsound)
 Help on module winsound:
     NAME
     winsound
     DESCRIPTION
     PlaySound(sound, flags)—play a sound
     SND_FILENAME—sound is a wav file name
     SND_ALIAS—sound is a registry sound association name
     SND_LOOP—play the sound repeatedly; must also specify
     SND_ASYNC
     SND_MEMORY—sound is a memory image of a wav file
     SND_PURGE—stop all instances of the specified sound
     SND_ASYNC—PlaySound returns immediately
     SND_NODEFAULT—Do not play a default beep if the sound
     cannot be found
     SND_NOSTOP—Do not interrupt any sounds currently
     playing
     SND_NOWAIT—Return immediately if the sound driver is
     busy
     Beep(frequency, duration)—Make a beep through the PC
     speaker.
     MessageBeep(type)—Call Windows MessageBeep.
     FUNCTIONS
     eep(frequency, duration)
     A wrapper around the Windows Beep API.
     frequency
     Frequency of the sound in hertz.
     Must be in the range 37 through 32,767.
     duration
     How long the sound should play, in milliseconds.
```

https://doi.org/10.15215/remix/9781998944088.01

```
    MessageBeep(type = 0)
    Call Windows MessageBeep(x).
    x defaults to MB_OK.
    PlaySound(sound, flags)
    A wrapper around the Windows PlaySound API.
    sound
    The sound to play; a filename, data, or None.
    flags
    Flag values, ORed together. See module
    documentation.
    DATA
    MB_ICONASTERISK = 64
    MB_ICONEXCLAMATION = 48
    MB_ICONHAND = 16
    MB_ICONQUESTION = 32
    MB_OK = 0
    SND_ALIAS = 65536
    SND_APPLICATION = 128
    SND_ASYNC = 1
    SND_FILENAME = 131072
    SND_LOOP = 8
    SND_MEMORY = 4
    SND_NODEFAULT = 2
    SND_NOSTOP = 16
    SND_NOWAIT = 8192
    SND_PURGE = 64
    FILE
    s:\python\python311\dlls\winsound.pyd
```

Among the functions defined in the module, PlaySound is an important one for playing sound or music files. The following statement will play a WAV file named dj.wav.

```
>>> import winsound
>>> winsound.PlaySound("c:/users/comp218/
dj.wav",winsound.SND_FILENAME)
```

When using the PlaySound function to a play sound file, you must make sure the WAV file exists in the default or specified path. In the example above, an absolute path has been given. You can also use a relative path that makes

https://doi.org/10.15215/remix/9781998944088.01

use of two special notations, a single dot (.) representing the current directory and a double dot (..) representing the parent directory; or you don't need to specify the path at all if the WAV file is in the current directory. In any case, the rule is that you must be clearly aware of where the file is located. This rule is applicable whenever the file is used.

PyGame

The PlaySound function in the standard winsound module can play only WAV files. To play the popular MP3 music files in your Python applications, use the module called mixer in the PyGame package. Because the package is usually included in the standard Python distribution, you can install the package into your Python programming environment using the pip command, as shown below:

```
pip install pygame
```

Then you can import and use the mixer module to load and play MP3 files, as shown below:

```
>>> from pygame import mixer # load the required library
Hello from the pygame community. https://www.pygame.org/
contribute.html
>>> mixer.init()
>>> mixer.music.load("../I_Will_Remember_You.mp3")
>>> mixer.music.play()
```

To learn more about how to use the mixer and mixer.music module, you can run the following commands in Python interactive mode as shown below, after the module has been imported:

```
>>> help(mixer)
```

You can then see the functions defined within the module, as further detailed below.

CHANNEL(ID)

This is used to create and return a Channel object for controlling playback.

FADEOUT(TIME)

This sets the time to fade out the volume on all sounds before stopping.

https://doi.org/10.15215/remix/9781998944088.01

FIND_CHANNEL(FORCE = FALSE)

This finds and returns an unused channel.

GET_BUSY()

This tests if any sound is being mixed and returns a Boolean value.

GET_INIT()

This tests if the mixer is initialized and returns a tuple (frequency, format, channels) representing the channel.

GET_NUM_CHANNELS()

This can be used to check and return the total number of playback channels.

INIT(FREQUENCY = 22050, SIZE = −16, CHANNELS = 2, BUFFER = 4096, DEVICENAME = NONE, ALLOWEDCHANGES = AUDIO_ALLOW_FREQUENCY_CHANGE | AUDIO_ALLOW_CHANNELS_CHANGE)

This can be used to initialize the mixer module.

PAUSE() -> NONE, TEMPORARILY STOP PLAYBACK OF ALL SOUND CHANNELS
PRE_INIT(FREQUENCY=22050, SIZE = −16, CHANNELS = 2, BUFFERSIZE = 4096, DEVICENAME = NONE)

These can be used to preset the mixer init arguments.

QUIT()

This can be used to uninitialize the mixer.

SET_NUM_CHANNELS(COUNT)

This can be used to set the total number of playback channels.

SET_RESERVED(COUNT)

This can be used to keep channels from being automatically used.

STOP()

This can be used to stop playback on all sound channels.

UNPAUSE()

This can be used to resume playback on sound channels after it has been paused.

https://doi.org/10.15215/remix/9781998944088.01

The mixer module has a submodule named music. To learn what functions are available in the mixer.music submodule module, run the following statement:

```
>>> help(mixer.music)
```

You will then see the following information about the related functions.

FADEOUT(TIME)

This can be used to stop music playback after fading out.

GET_BUSY()

This can be used to check if the music stream is playing. It will return True or False.

GET_ENDEVENT()

This can be used to get the event a channel sends when playback stops.

GET_POS()

This can be used to get the music playtime.

GET_VOLUME() -> VALUE

This can be used to get the music volume.

LOAD(FILENAME) -> NONE, OR LOAD(OBJECT) -> NONE,

This can be used to load a music file/object for playback.

PAUSE() -> NONE

This can be used to temporarily stop music playback.

PLAY(LOOPS = 0, START = 0.0) -> NONE

This can be used to start the music stream playback.

QUEUE(FILENAME) -> NONE

This can be used to queue a music file to follow the currently playing file.

REWIND() -> NONE

This can be used to restart the music.

https://doi.org/10.15215/remix/9781998944088.01

SET_ENDEVENT() -> NONE
SET_ENDEVENT(TYPE) -> NONE

These can be used to have the mixer send events when playback stops.

SET_POS(POS) -> NONE

This can be used to set the position in the music file when starting playback.

SET_VOLUME(VALUE) -> NONE

This can be used to set the music volume.

STOP() -> NONE

This can be used to stop the music playback.

UNPAUSE() -> NONE

This can be used to resume paused music.

These functions in mixer.music are the ones used directly to handle music files. These functions are sufficient for you to develop a high-quality music player with what you will learn in Chapter 9 on developing GUI-based applications in Python.

Please note that the mixer module from the PyGame package can also play other types of music files including WAV, as shown below:

```
>>> from pygame import mixer   # import mixer module from
PyGame
>>> mixer.init()   # initialize the mixer
>>> mixer.music.load("../I_will_Remember_you.mp3")   #
load the mp3 file
>>> mixer.music.play(loops = 2)   # play the most recent
loaded file twice
```

The functions listed above are needed if you are developing a music player with PyGame. For details on these functions, please refer to the official documentation on the PyGame mixer at https://www.pygame.org/docs/ref/mixer.html.

8.11 Modules for Creating and Manipulating Graphics and Images

In computing and information processing, graphics are an important part of data and information. In this section, we learn how to create and manipulate graphics and images with Python.

https://doi.org/10.15215/remix/9781998944088.01

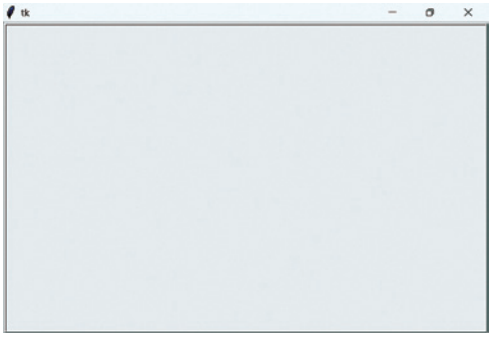


Figure 8-2: An example of TK Canvas

Create Graphics with Tkinter

The module built into the standard Python distribution for creating graphics is the Tkinter module, which is commonly used to develop graphical user interface (GUI) applications. However, Tkinter also provides a widget called Canvas for graphics and images. The following statements in Python interactive mode will produce a window containing a Canvas ready for drawing graphic objects—Canvas items:

```
>>> from tkinter import *
>>> d_board = Canvas()
```

Table 8-4 is a list of Canvas items we can draw on a Canvas.

Table 8-4: A list of functions that can be used to draw on a Canvas

Graphic object	Canvas method to create	Code sample

arc or arc-shaped region (such as a chord or pie slice)	create_arc(bbox, **options)	`>>> d_board = Canvas()`
		`>>> d_board.pack()`
		`>>> d_board.create_arc(30, 50, 100, 200)`
		`1`

bitmap (built-in or read from an XBM file)	create_bitmap(position, **options)	`>>> d_board.create_bitmap(30, 50)`
image (a BitmapImage or PhotoImage instance)	create_image(position, **options)	`>>> d_board.create_image(30, 50)`
line	create_line(coords, **options)	`>>> d_board.create_line(30, 50, 100, 200)` `2`

(continued on next page)

https://doi.org/10.15215/remix/9781998944088.01

Table 8-4: A list of functions that can be used to draw on a Canvas *(continued)*

Graphic object	Canvas method to create	Code sample
oval (a circle or an ellipse)	create_oval(bbox, **options)	`>>> d_board.create_oval(30, 50, 80, 80, fill = "RED")`
polygon	create_polygon(coords, **options)	`>>> d_board.create_polygon(30, 50, 80, 80, 70, 90, fill = "RED")`
rectangle	create_rectangle(bbox, **options)	`>>> d_board.create_rectangle(30, 50, 80, 80, fill = "RED")`
text	create_text(position, **options)	`>>> d_board.create_text(130, 150, text = "Hello!")`
window	create_window(position, **options)	

Every method listed in Table 8-4 returns a unique ID for the created graphic object, which can be used later to manipulate the object.

Note that graphic objects created by the above methods will be stacked on the Canvas and will remain until being moved, lifted, lowered, or deleted, with the methods in Table 8-5.

Table 8-5: Other methods of Canvas within Tkinter

Method	Operation	Code sample
dchars(item, from, to = None)	Deletes text from an editable graphic item such as text: *from* is where to start deleting text, *to* is where to stop deleting text. If *to* is omitted, only a single character is deleted.	`>>> d_board.create_text(130, 150, text = "Hello Python!")` `>>> d_board.dchars('text', 1,2)`

delete(item)	Deletes all matching items.	`>>> h3 = d_board.create_text(230, 150, text="Hello!")`
		`>>> d_board.delete(h3)`
		`>>> d_board.delete(3)`
		`>>> d_board.delete(10)`
		`>>> d_board.delete(11)`
		`>>> d_board.delete(9)`

https://doi.org/10.15215/remix/9781998944088.01

Table 8-5: Other methods of Canvas within Tkinter *(continued)*

Method	Operation	Code sample
coords(item, *coords)	Returns or sets the coordinates of matching items.	`>>> d_board.coords(o1, 30, 150, 80, 250)`
move(item, dx, dy)	Moves matching items by an offset.	`>>> d_board.move(o1, 10, 15)`

Canvas has many other methods for accessing and manipulating graphic objects. Running the following statements in Python interactive mode will give you a list of names defined within the Canvas class.

```
>>> from tkinter import *
>>> dir(Canvas)
['_Misc__winfo_getint', '_Misc__winfo_parseitem', '__
class__', '__delattr__', '__dict__', '__dir__', '__doc__',
'__eq__', '__format__', '__ge__', '__getattribute__',
'__getitem__', '__gt__', '__hash__', '__init__', '__init_
subclass__', '__le__', '__lt__', '__module__', '__ne__',
'__new__', '__reduce__', '__reduce_ex__', '__repr__',
'__setattr__', '__setitem__', '__sizeof__', '__str__',
'__subclasshook__', '__weakref__', '_bind', '_configure',
'_create', '_displayof', '_do', '_getboolean', '_
getconfigure', '_getconfigure1', '_getdoubles', '_getints',
'_grid_configure', '_gridconvvalue', '_last_child_ids',
'_nametowidget', '_noarg_', '_options', '_register', '_
report_exception', '_root', '_setup', '_subst_format',
'_subst_format_str', '_substitute', '_tclCommands', '_
windowingsystem', 'addtag', 'addtag_above', 'addtag_all',
'addtag_below', 'addtag_closest', 'addtag_enclosed',
'addtag_overlapping', 'addtag_withtag', 'after',
'after_cancel', 'after_idle', 'anchor', 'bbox', 'bell',
'bind', 'bind_all', 'bind_class', 'bindtags', 'canvasx',
'canvasy', 'cget', 'clipboard_append', 'clipboard_clear',
'clipboard_get', 'columnconfigure', 'config', 'configure',
'coords', 'create_arc', 'create_bitmap', 'create_image',
'create_line', 'create_oval', 'create_polygon', 'create_
rectangle', 'create_text', 'create_window', 'dchars',
'delete', 'deletecommand', 'destroy', 'dtag', 'event_add',
'event_delete', 'event_generate', 'event_info', 'find',
'find_above', 'find_all', 'find_below', 'find_closest',
'find_enclosed', 'find_overlapping', 'find_withtag',
```

https://doi.org/10.15215/remix/9781998944088.01

```
'focus', 'focus_displayof', 'focus_force', 'focus_get',
'focus_lastfor', 'focus_set', 'forget', 'getboolean',
'getdouble', 'getint', 'gettags', 'getvar', 'grab_
current', 'grab_release', 'grab_set', 'grab_set_global',
'grab_status', 'grid', 'grid_anchor', 'grid_bbox',
'grid_columnconfigure', 'grid_configure', 'grid_forget',
'grid_info', 'grid_location', 'grid_propagate', 'grid_
remove', 'grid_rowconfigure', 'grid_size', 'grid_slaves',
'icursor', 'image_names', 'image_types', 'index', 'info',
'insert', 'itemcget', 'itemconfig', 'itemconfigure',
'keys', 'lift', 'location', 'lower', 'mainloop', 'move',
'nametowidget', 'option_add', 'option_clear', 'option_get',
'option_readfile', 'pack', 'pack_configure', 'pack_forget',
'pack_info', 'pack_propagate', 'pack_slaves', 'place',
'place_configure', 'place_forget', 'place_info', 'place_
slaves', 'postscript', 'propagate', 'quit', 'register',
'rowconfigure', 'scale', 'scan_dragto', 'scan_mark',
'select_adjust', 'select_clear', 'select_from', 'select_
item', 'select_to', 'selection_clear', 'selection_get',
'selection_handle', 'selection_own', 'selection_own_
get', 'send', 'setvar', 'size', 'slaves', 'tag_bind',
'tag_lower', 'tag_raise', 'tag_unbind', 'tk_bisque',
'tk_focusFollowsMouse', 'tk_focusNext', 'tk_focusPrev', 'tk_
setPalette', 'tk_strictMotif', 'tkraise', 'type', 'unbind',
'unbind_all', 'unbind_class', 'update', 'update_idletasks',
'wait_variable', 'wait_visibility', 'wait_window',
'waitvar', 'winfo_atom', 'winfo_atomname', 'winfo_cells',
'winfo_children', 'winfo_class', 'winfo_colormapfull',
'winfo_containing', 'winfo_depth', 'winfo_exists', 'winfo_
fpixels', 'winfo_geometry', 'winfo_height', 'winfo_id',
'winfo_interps', 'winfo_ismapped', 'winfo_manager', 'winfo_
name', 'winfo_parent', 'winfo_pathname', 'winfo_pixels',
'winfo_pointerx', 'winfo_pointerxy', 'winfo_pointery',
'winfo_reqheight', 'winfo_reqwidth', 'winfo_rgb', 'winfo_
rootx', 'winfo_rooty', 'winfo_screen', 'winfo_screencells',
'winfo_screendepth', 'winfo_screenheight', 'winfo_
screenmmheight', 'winfo_screenmmwidth', 'winfo_screenvisual',
'winfo_screenwidth', 'winfo_server', 'winfo_toplevel',
'winfo_viewable', 'winfo_visual', 'winfo_visualid', 'winfo_
visualsavailable', 'winfo_vrootheight', 'winfo_vrootwidth',
'winfo_vrootx', 'winfo_vrooty', 'winfo_width', 'winfo_x',
'winfo_y', 'xview', 'xview_moveto', 'xview_scroll', 'yview',
'yview_moveto', 'yview_scroll']
```

https://doi.org/10.15215/remix/9781998944088.01

You can then call help on each of the names in the list to learn more about the name defined. The following are just two examples:

```
>>> help(Canvas.addtag)
```

Running help on the function addtag in module tkinter outputs the following:

```
addtag(self, *args)
    Internal function.
```

```
>>> help(Canvas.after)
```

Running help on the function after in module tkinter outputs the following:

```
after(self, ms, func=None, *args)
    Call function once after given time.
    MS specifies the time in milliseconds. FUNC gives the
    function, which shall be called. Additional parameters
    are given as parameters to the function call. Returns
    identifier to cancel scheduling with after_cancel.
```

```
>>> help(Canvas.create_image)
```

Running help on the function create_image in module tkinter outputs the following:

```
create_image(self, *args, **kw)
    Create image item with coordinates x1, y1.
```

The following coding example will draw a line on a Canvas:

```
import tkinter
from tkinter.constants import *
tk = tkinter.Tk()
canvas = tkinter.Canvas(tk, relief = RIDGE, borderwidth = 2)
canvas.pack(fill = BOTH, expand=1)
ln1 = canvas.create_line(100, 100, 300, 300, width = 6)
tk.mainloop()
```

https://doi.org/10.15215/remix/9781998944088.01

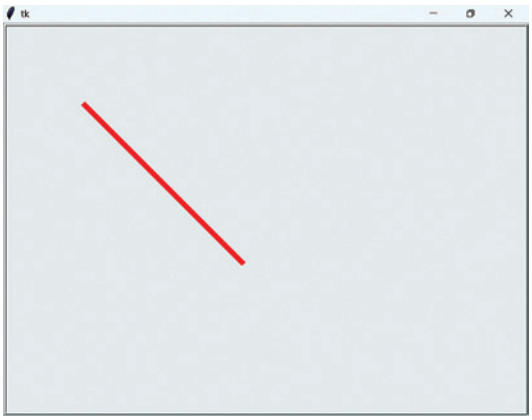


Figure 8-3: Drawing on TK Canvas

Manipulate Images with Pillow

Another way you can work with visual objects in Python is to manipulate images stored in files. These manipulations include the following:

- rotating
- converting from colour to grey-scale
- applying colour filtering
- highlighting a specific area of an image
- blurring an image or part of it
- sharpening an image or part of it
- changing the brightness of an image
- detecting the edge on an image
- scaling an image
- applying colour inversion to an image
- morphing one image into another image

How can all these manipulations be done within your computer? First, an image is made of pixels, which can be stored in an m × n matrix, or two-dimensional array, mapped to a rectangular area of the computer screen. The value of each cell of the matrix represents a pixel and contains all the information about it. All manipulations to the image can be done by manipulating the matrix or its values.

To manipulate an image with Python, you can use a package called Pillow (available from https://pypi.org/project/Pillow/2.2.1/ or https://github.com/python-pillow/Pillow). Because it is not a standard part of the Python library, you will need to install it with the following statement:

https://doi.org/10.15215/remix/9781998944088.01

```
pip install Pillow
Collecting Pillow
Downloading
wordhttps://files.pythonhosted.org/packages/70/21/04723e78916eff8e0
  9901dbb7dc9705f4de8a0dfe7882a9ed56982bd128e/Pillow-6.0.0
  -cp37-cp37m-win32.whl (1.7MB)
|████████████████████████████████| 1.7MB
  1.3MB/s

Installing collected packages: Pillow
Successfully installed Pillow-6.0.0
Once this is done, you can import and then use the following two
  modules:
Image,
ImageFilter
```

The following is a coding sample:

```
from PIL import Image, ImageFilter
# read the image
im = Image.open('./resort.jpg')
im_sharp = im.filter( ImageFilter.SHARPEN )
# save the filtered image to a new file
im_sharp.save( './resort-sharp.jpg', 'JPEG' )

# display the image
im_sharp.show()
```

The sharpened image is shown in Figure 8-4.

Note that the Image module has a class with the same name as the module, Image, although it is more convenient to construct an object of Image with the open statement rather than the constructor of Image class.

Once an image object has been generated with the open statement, we can check its format, size, and mode by looking at the format, size, and mode attributes and using the following methods of the Image class to manipulate the image object:

- Image.convert(self, mode = None, matrix = None, dither = None, palette = 0, colors = 256) makes various conversions to the image object.

https://doi.org/10.15215/remix/9781998944088.01

Figure 8-4: Picture sharpened with Pillow (©Harris Wang, Athabasca University)

- Image.copy(self) makes a copy and retains the original image object.
- Image.crop(self, box=None) returns a rectangular region of the image, defined by box.
- Image.draft(self, mode, size) returns a draft version of the image, such as a grey-scale version.
- Image.effect_spread(self, distance) returns an image with pixels randomly spread throughout the image.
- Image.filter(self, filter) filters this image using the given filter specified in the ImageFilter module.
- Image.paste(self, im, box=None, mask=None) pastes another image (im) into this image.
- Image.putalpha(self, alpha) adds or replaces the alpha layer in this image.
- Image.putdata(self, data, scale = 1.0, offset = 0.0) copies a sequence of pixel data to this image.
- Image.putpalette(self, data, rawmode = 'RGB') attaches a palette to this image.
- Image.putpixel(self, xy, value) modifies the pixel at the given position.
- Image.quantize(self, colors = 256, method = None, kmeans = 0, palette = None, dither = 1) converts the image to P mode with the specified number of colours.

https://doi.org/10.15215/remix/9781998944088.01

- Image.emap_palette(self, dest_map, source_palette = None) rewrites the image to reorder the palette.
- Image.resize(self, size, resample = 0, box = None) returns a resized copy of this image.
- Image.rotate(self, angle, resample = 0, expand = 0, center = None, translate = None, fillcolor = None) returns a rotated copy of this image.
- Image.split(self), splits the image into individual bands, such as R, G, B.
- Image.tobitmap(self, name='image') converts the image to an X11 bitmap.
- Image.tobytes(self, encoder_name = 'raw', *args) returns the image as a bytes-object.
- Image.toqimage(self) returns a QImage copy of this image.
- Image.toqpixmap(self) returns a QPixmap copy of this image.
- Image.transform(self, size, method, data=None, resample = 0, fill = 1, fillcolor = None) transforms this image to a given size but in the same mode as the original.
- Image.transpose(self, method) transposes the image (flips or rotates in 90-degree steps).

There are other methods defined within the Image class for other purposes. You can find out more info about the Image class by running the following statement in Python interactive mode:

```
>>> from PIL import Image, ImageFilter
>>> help(Image.Image)
```

As we have seen from the above list, the Image class has provided a good set of methods to manipulate an image.

The ImageFilter module provides some filtering operations on images, as the name implies. These filtering operations include blurring, box blurring, contouring, colour transformation, detailing, edge enhancing, embossing, sharpening, smoothing, and more.

8.12 Modules for Data Analytics

The modules often used for data analytics include pandas, NumPy, SciPy, and matplotlib. Among the four modules, pandas is mostly used to prepare

https://doi.org/10.15215/remix/9781998944088.01

data; NumPy and SciPy are used for numeric analysis and science computing, respectively; and matplotlib is for visualization.

Since data analytics is a big topic, we will only give some examples of how the modules can be used.

The first example is to show how to use matplotlib to visualise the square root function in math.

```
import math
import matplotlib.pyplot as mpl
sqrt_data = {i+1 : math.sqrt(i+1) for i in range(100)}
x1 = list(sqrt_data.keys())
y1 = list(sqrt_data.values())
mpl.plot(x1, y1)
mpl.title("visualization of square root")
mpl.xlabel('x')
mpl.ylabel('y')
mpl.show()
```

The plot produced by the program is shown in Figure 8-5.

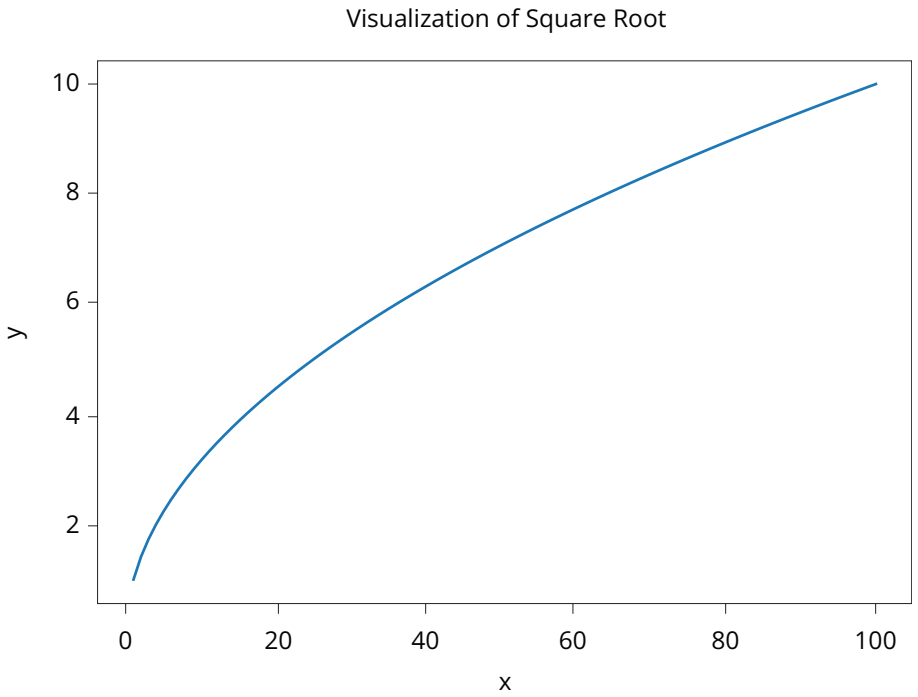


Figure 8-5: Visualization of square root

https://doi.org/10.15215/remix/9781998944088.01

Our next example is to visualize the world population changes in some regions as well as world total since 1960. The program code is as follows:

```
import pandas as pd
import matplotlib.pyplot as mplt
content = pd.read_excel("world-population.xls")
years = [1960, 1970, 1980, 1990, 2000, 2010, 2017]
mplt.plot(years, content.iloc[0, 1:])
mplt.plot(years, content.iloc[1, 1:])
mplt.plot(years, content.iloc[7, 1:])
mplt.plot(years, content.iloc[11, 1:])
mplt.plot(years, content.iloc[12, 1:])
mplt.plot(years, content.iloc[17, 1:])
mplt.plot(years, content.iloc[22, 1:])
mplt.title("Population - World Total and Region Total")
mplt.xlabel("years")
mplt.ylabel("Populations (in millions)")
mplt.legend(["World Total", "Africa", "Latin America",
"North America", "Asia", "Europe", "Oceana"])
mplt.show()
```

In the program, the pandas module is used to read and prepare the data. For details on how it works, please read the complete documentation at https://pandas.pydata.org/docs/—the user guide at https://pandas.pydata.org/docs/user_guide/index.html#user-guide in particular.

The rendered result of the program is shown in Figure 8-6.

NumPy allows you to do math manipulations on a data set, most often manipulations on a matrix. Here is an example:

```
import numpy as np # import numpy

a = np.array(range(21,25)) # create an array from a
python list
print(a, a.shape) # print the array and its shape

# create a 2-D array from nested lists
b = np.array([[1, 2, 3, 4], [5, 6, 7, 8],
        [11, 12, 13, 14], [15, 16, 17, 18]])

print(b, b.shape)  # print the array and its shape
```

https://doi.org/10.15215/remix/9781998944088.01

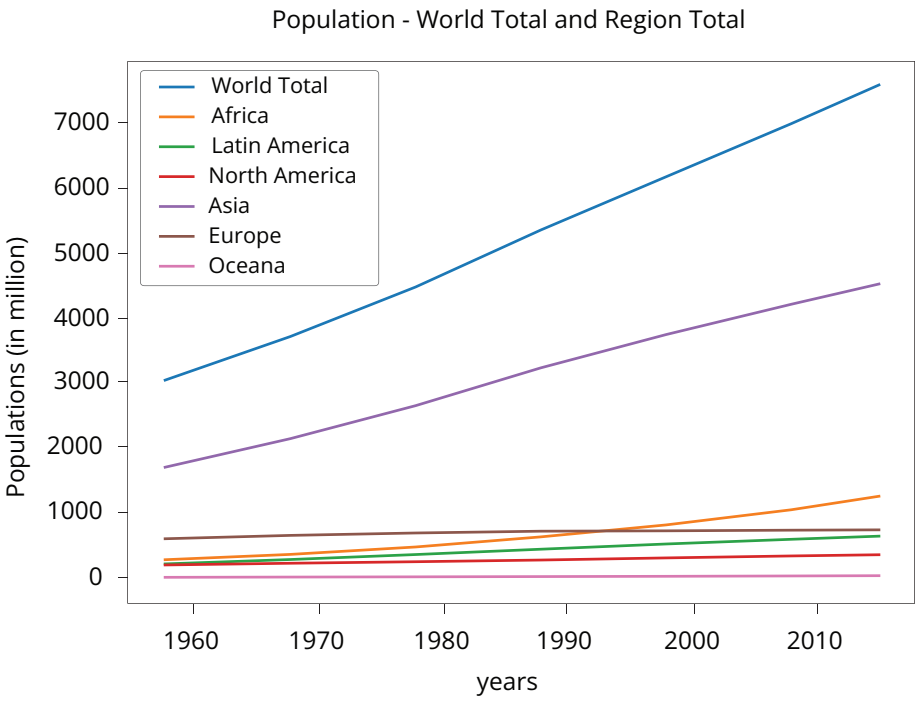


Figure 8-6: Visualization of world population changes

```
c = a + b # perform element-wise addition on two arrays

print(c, c.shape)  # print the result and the shape

d = np.dot(a, b)  # perform matrix multiplication

print(d)  # print the result

e = np.cos(a) # apply a mathematical function to an array

print(e)  # print the result

f = np.sort(b, axis=0) # sort an array along a given axis

print(f)  # print the result
```

The output from the program is as follows:

```
[21 22 23 24] (4,)
[[ 1 2 3 4]
```

https://doi.org/10.15215/remix/9781998944088.01

```
 [ 5 6 7 8]
 [11 12 13 14]
 [15 16 17 18]] (4, 4)
[[22 24 26 28]
 [26 28 30 32]
 [32 34 36 38]
 [36 38 40 42]] (4, 4)
[ 744 834 924 1014]
[-0.54772926 -0.99996083 -0.53283302 0.42417901]
[[ 1 2 3 4]
 [ 5 6 7 8]
 [11 12 13 14]
 [15 16 17 18]]
```

More details on NumPy can be found at https://numpy.org/doc/stable/.

SciPy is built upon NumPy, including various functions often needed for scientific computing. The following example shows how integration function in SciPy is used to integrate a function.

```
import numpy as np # import numpy and scipy.integrate
from scipy.integrate import quad  # import quad integral
function

# define a function to be integrated
def f(x):
  return np.exp(-x**2)  # use exp function from NumPy

# integrate the function from 0 to 1 using quad
result, _ = quad(f, 0, 1)

# print the result and the estimated error
print(result)
```

The result is 0.7468241328124271.

Chapter Summary

- In addition to functions and classes, modules and packages are important constructs that can be used to do modular programming in Python.

https://doi.org/10.15215/remix/9781998944088.01

- A module can be any file with a py extension that has legitimate Python program codes or scripts in it.
- A package is a file directory containing several Python modules.
- To identify the directories in which the Python packages reside, each such directory must contain a file named __init__.py.
- The __init__.py file defines what can be imported from the package. These may include variables, functions, and classes.
- Packages can be nested to any depth, just like file directories are nested. However, each directory for a package must have its own __init __.py file.
- Dotted notation is used to refer to a package or module within a deep package hierarchy.
- To use variables, functions, and classes defined in a module, you have to import the module or the specific variables, functions, and/or classes, using the ***import*** statement or ***from ... import*** statement.
- A large number of Python modules and packages have been developed and made available on the internet for Python programmers. Many of these modules and packages have been already installed with the standard Python distribution, such as the Anaconda package.
- A good Python programmer or software developer should have knowledge of existing modules and packages, including what they are developed for and what they do.
- Programmers can develop their own modules for their own applications and development.

Exercises

1. Open VS Code, create or open a Jupyter Notebook file (.ipynb), and select a Python virtual environment for the notebook file. Open a terminal and run the pip list command to see what library modules have been installed in the virtual environment.

   In the output of your VS Code shell terminal, identify some interesting packages and modules, and write a summary of those packages and modules in a Word document, including what each is developed for, where it may be used, and so on.
2. In VS Code, open the Jupyter Notebook file named chapter-8.ipynb. Create a new cell and ***import*** the math module, run the help command to study each of the functions defined in the math module, and do some hands-on coding with the function.

https://doi.org/10.15215/remix/9781998944088.01

3. Search the internet for tutorials or other web documents related to web scraping with Python and choose some to watch or read. Take some coding samples to run in your Jupyter Notebook within VS Code, then develop your own application based on the code samples.

Projects

1. Rational numbers are those real numbers that can be represented as a quotient of two integers such as a/b, which can then be represented by the pair of integers a and b. For this project, define a module that contains the definition of a class named Rational, within which dunder methods for print, addition, subtraction, division, multiplication, and various comparisons are defined.
2. Develop an application using the math module to calculate and display a table of squares for integers from 0 to 99. The layout is illustrated in Table 8-6.

Table 8-6: The layout of the table to be produced

	0	1	2	3	4	5	6	7	8	9
0	0	1	4	9	26	25	36	49	8	9
1	100	121	144	169	196	225	256	289	324	361
2	…	…	…							
3										
4										
5										
6										
7										
8										
9										

3. Develop an application to calculate and display a table showing the square roots of integers from 0 to 99 (similar to what you did for Project 2).

https://doi.org/10.15215/remix/9781998944088.01

This page intentionally left blank

Chapter 9

Develop GUI-Based Applications

In terms of user interface, there are two types of computer systems or applications: terminal-based, which can be run from a terminal, and those with a graphical user interface (GUI), or GUI-based. So far, the applications we have programmed have all been terminal-based. In this chapter you will learn how to develop GUI-based applications in Python.

Learning Objectives

After completing this chapter, you should be able to

- explain terminal-based applications and GUI-based applications as well as their differences.
- explain which Python libraries are available for developing GUI-based applications.
- discuss the widgets, functions, other classes, and methods provided in the Tkinter module.
- use widgets, functions, classes, and methods provided in the Tkinter module to design and implement a graphical user interface for a given application.
- develop GUI-based applications using the Tkinter module.
- discuss the themed Tkinter(Ttk) module and how it differs from the Tkinter module.

https://doi.org/10.15215/remix/9781998944088.01

9.1 Terminal-Based Applications Versus GUI-Based Applications

A computer terminal is a device, often referring to a utility program in today's Windows environment, that can receive and deliver data or commands. The following shows what a terminal on Windows 11 looks like:

```
(base) PS C:\Users\james>
```

A terminal-based application is a program that runs from the command line within a terminal. For that reason, terminal-based applications are also called applications with a command-line interface. The following shows a Python program with a command-line interface running within a terminal.

```
(base) PS C:\workshop\comp218\samples\project2> python
'.\perfect number.py'
tell me the upper bound:33
6 = [1, 2, 3]    28 = [1, 2, 4, 7, 14]
(base) PS C:\workshop\comp218\samples\project2> python
'.\perfect number.py'
tell me the upper bound:53
6 = [1, 2, 3]    28 = [1, 2, 4, 7, 14]
(base) PS C:\workshop\comp218\samples\project2>
```

The terminal-based program shown above takes an integer as upper bound from the user and then finds all perfect numbers between 1 and the upper bound. With a terminal-based application, only a keyboard can be used by users to interact with the application.

A GUI-based application provides users with a graphical interface so that users can interact with the application with keyboard and mouse. VS Code IDE is an example of a GUI-based application we have been using throughout this text.

The difference between terminal-based and GUI-based applications is obvious. Although GUI-based applications are more user-friendly and more common today, terminal-based applications still exist and are used by some IT professionals, such as system administrators.

In the next section, we will learn how to develop GUI-based applications with Python.

https://doi.org/10.15215/remix/9781998944088.01

9.2 Designing and Developing GUI-Based Applications in Python

The key component of a GUI-based application is a graphical user interface (GUI) on which both mouse and keyboard can be used to interact with the application on a two-dimensional graphic interface. As mentioned above, GUI-based applications are more user-friendly than terminal-based applications. This is why most computer applications used today are GUI-based.

In general, you need to ask yourself the following questions when designing a GUI app:

- What will be shown to users?
- What input will be taken from users?
- What actions will we allow users to take?
- What actions will be taken in response to the users' input?
- Where do we display the results of the actions?

Several Python modules are available for developing GUI-based applications.

- Tkinter, a de facto GUI library for Python applications, comes with the standard Python distribution so that you can import and use it right away. Documentation on tkinter can be found at https://docs.python.org/3/library/tk.html. You learned how to use Python modules and packages, including tkinter, in Chapter 8.
- PyGObject is a Python implementation of GObject-based libraries such as GTK, GStreamer, WebKitGTK, GLib, GIO, and some others. If you have played with Linux, you will probably know the GTK- and GNOME-based graphical user interfaces. The project is hosted at https://pygobject.readthedocs.io/en/latest/.
- PyQt5 is another GUI library for Python GUI-based application development that implements the well-known Qt graphic framework. Information about PyQt5 can be easily found on the internet, and its home is at https://pypi.org/project/PyQt5/.

We chose to use a module called tkinter for its popularity among Python application developers and because of its light weight. The tkinter module usually comes with the standard Python distribution. To test if you have it installed, run ***import tkinter*** as shown below to see if you will get the same output:

https://doi.org/10.15215/remix/9781998944088.01

```
>>> import tkinter
>>> dir(tkinter)
['ACTIVE', 'ALL', 'ANCHOR', 'ARC', 'BASELINE', 'BEVEL',
'BOTH', 'BOTTOM', 'BROWSE', 'BUTT', 'BaseWidget',
'BitmapImage', 'BooleanVar', 'Button', 'CASCADE',
'CENTER', 'CHAR', 'CHECKBUTTON', 'CHORD', 'COMMAND',
'CURRENT', 'CallWrapper', 'Canvas', 'Checkbutton',
'DISABLED', 'DOTBOX', 'DoubleVar', 'E', 'END', 'EW',
'EXCEPTION', 'EXTENDED', 'Entry', 'Event', 'EventType',
'FALSE', 'FIRST', 'FLAT', 'Frame', 'GROOVE', 'Grid',
'HIDDEN', 'HORIZONTAL', 'INSERT', 'INSIDE', 'Image',
'IntVar', 'LAST', 'LEFT', 'Label', 'LabelFrame',
'Listbox', 'MITER', 'MOVETO', 'MULTIPLE', 'Menu',
'Menubutton', 'Message', 'Misc', 'N', 'NE', 'NO',
'NONE', 'NORMAL', 'NS', 'NSEW', 'NUMERIC', 'NW',
'NoDefaultRoot', 'OFF', 'ON', 'OUTSIDE', 'OptionMenu',
'PAGES', 'PIESLICE', 'PROJECTING', 'Pack', 'PanedWindow',
'PhotoImage', 'Place', 'RADIOBUTTON', 'RAISED',
'READABLE', 'RIDGE', 'RIGHT', 'ROUND', 'Radiobutton',
'S', 'SCROLL', 'SE', 'SEL', 'SEL_FIRST', 'SEL_LAST',
'SEPARATOR', 'SINGLE', 'SOLID', 'SUNKEN', 'SW',
'Scale', 'Scrollbar', 'Spinbox', 'StringVar', 'TOP',
'TRUE', 'Tcl', 'TclError', 'TclVersion', 'Text',
'Tk', 'TkVersion', 'Toplevel', 'UNDERLINE', 'UNITS',
'VERTICAL', 'Variable', 'W', 'WORD', 'WRITABLE',
'Widget', 'Wm', 'X', 'XView', 'Y', 'YES', 'YView',
'__builtins__', '__cached__', '__doc__', '__file__',
'__loader__', '__name__', '__package__', '__path__',
'__spec__', '_cnfmerge', '_default_root', '_exit', '_
flatten', '_join', '_magic_re', '_setit', '_space_re',
'_splitdict', '_stringify', '_support_default_root',
'_test', '_tkerror', '_tkinter', '_varnum', 'constants',
'enum', 'getboolean', 'getdouble', 'getint', 'image_
names', 'image_types', 'mainloop', 're', 'sys',
'wantobjects']
```

A GUI application starts with a frame or window of a given size in terms of pixels, which is divided into rows and columns. Each of the grids is numbered from top to bottom, left to right, as shown in the following diagram:

https://doi.org/10.15215/remix/9781998944088.01

(0,0)	(0,1)	(0,2)	(0,3)	(0,4)	(0,5)
(1,0)	(1,1)	(1,2)	(1,3)	(1,4)	(1,5)
(2,0)	(2,1)	(2,0)	(2,3)	(2,4)	(2,5)
(3,0)	(3,1)	(3,2)	(3,3)	(3,4)	(3,5)
(4,0)	(4,1)	(4,2)	(4,3)	(4,4)	(4,5)
(5,0)	(5,1)	(5,2)	(5,3)	(5,4)	(5,5)

Please note that although the frame size is measured in pixels, the coordinate of each grid above is only a relative location within the grid and has nothing to do with pixels. The size of each grid is dynamically determined by the object or widget placed in it.

Tkinter Module

The following code running from Python interactive mode renders a window/frame (as shown in Figure 9-1):

```
>>> from tkinter import *
>>> f = Tk()
```

The following statements can be used to set the title and the size of the window:

```
>>> f.title("My First GUI App")
>>> f.geometry("300 x 200")
```

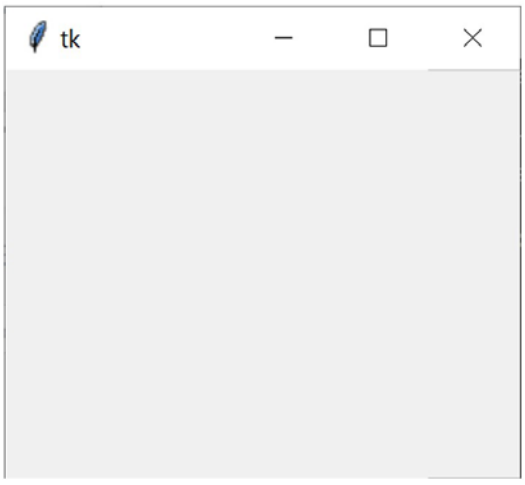


Figure 9-1: A window rendered by Python with Tkinter

https://doi.org/10.15215/remix/9781998944088.01

Within this frame/window, we can place specific objects, called widgets, to serve their designated purposes, as detailed below:

- Label displays text and bitmaps.
- Entry displays simple text.
- Button displays a button and associates a command with the button that can be invoked.
- Checkbutton displays a button with only an on or off state. It is often used in groups to provide a number of options for users to select.
- Menu displays menu bars, pull-down menus, and pop-up menus.
- OptionMenu allows the user to select a value from a menu.
- PanedWindow is a container that may contain a number of panes.
- Radiobutton displays a radio button with an on/off state. It is used in groups, from which the user can select only one option at a time.
- Frame is a container to organize other widgets.
- LabelFrame is a spacer or container for complex window layouts.
- Text displays text in multiple lines.
- Canvas provides an area on which you can draw shapes such as lines, ovals, polygons, and rectangles in your application.
- Scale provides a slider showing numerical scale.
- Scrollbar adds scrolling capability to various widgets, such as text and list boxes.
- Spinbox is a variant of the standard Tkinter Entry widget and selects from a fixed number of values.
- Toplevel provides a separate window container.

The following is a code sample in Jupyter Notebook:

In [ ]:
```
f = Tk() # make an instance of Tk window
f.title("My First GUI App") # add a title to the window
f.geometry("300 x 200") # set the size
# add a label widget with text
lb = tkinter.Label(text = "This is my first label") #
  label
lb.grid(row = 0, column = 0) # place the window
f.mainloop() # render it up to show
```

Out [ ]: The window will be rendered as shown in Figure 9-2:

In the above, the first statement creates a label object with the text “This is my first label” on it, whereas the second statement places the object in a specific grid of the window. If no arguments are specified for the grid, the

https://doi.org/10.15215/remix/9781998944088.01

Figure 9-2: A GUI example with more details

default value is used, which is the next available grid in the window in order from top to bottom, left to right.

Similarly, we can add an Entry widget to the window using the following statements:

```
In [ ]:    f = Tk()
           f.title("My first GUI app")
           f.geometry("300x200")
           lb = tkinter.Label(text = "Please input here")
           lb.grid(row = 0, column = 0)
           ent = tkinter.Entry()
           ent.grid(row = 0, column = 1)
           f.mainloop()
```

By now the window should look like the one shown in Figure 9-3.

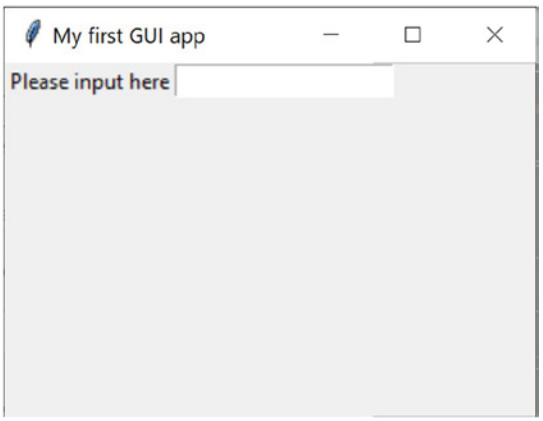


Figure 9-3: A GUI example with added entry widget

https://doi.org/10.15215/remix/9781998944088.01

In addition to the Entry widget for taking single-line text input, the Button and Text widgets take mouse click and multiple-line text input, respectively, from users. We can add buttons and text boxes to the window with the following statements:

```
In [ ]:  f = Tk()
         f.title("My first GUI app")
         f.geometry("300 x 200")
          lb = tkinter.Label(text = "Please input here")
          lb.grid(row = 0, column = 0)
          ent = tkinter.Entry()
          ent.grid(row = 0, column = 1)
          btn = Button(text = "Click me")
          btn.grid(column = 0, row = 1)
          txt = Text(width = 20, height = 10)
          txt.grid(column = 1, row = 2)
          f.mainloop()
```

The rendered window is shown in Figure 9-4.

Within the window, you can type in the entry field, type more in the text area, and click the button, but the app does not do anything in response, as shown in Figure 9-5.

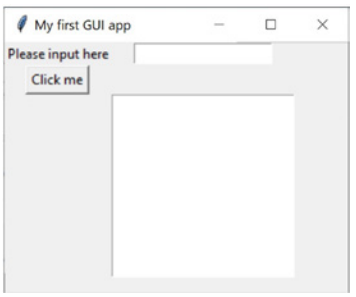


Figure 9-4: A GUI example with more widgets added

Figure 9-5: A GUI example taking user input

https://doi.org/10.15215/remix/9781998944088.01

How can we make the app respond to the user's input and do what we want it to do? We need to attach an even handler to the intended object, such as a button, as shown below:

```
In [ ]:    f = Tk()
           f.title("My first GUI app")
           f.geometry("300 x 200")
           lb = tkinter.Label(text = "Please input here")
           lb.grid(row = 0, column = 0)
           ent = tkinter.Entry()
           ent.grid(row = 0, column = 1)
           btn = Button(text = "Click me")
           btn.grid(column = 0, row = 1)
           txt = Text(width = 20, height = 10)
           txt.grid(column = 1, row = 2)

           def hdl():
           btn.config(text = "You clicked me!")

           btn.config(command = hdl) # add handler to act

           f.mainloop()
```

In the above, we first define a function as a handler, which simply changes the text on the button to "You clicked me," then attaches the handler to the button using the config() method built into the button object. As expected, the text on the button changes after a click on the button, as shown in Figure 9-6.

To learn more details about each of the widgets listed above, we can call for help in Python interactive mode or Jupyter Notebook. The following example shows how to get help on the Frame widget in Jupyter Notebook:

Figure 9-6: Now the GUI is able to respond to user input

https://doi.org/10.15215/remix/9781998944088.01

```
In [ ]:  from tkinter import *
         help(Frame)
```

As always, you can also search the internet for any widget in the tkinter module to get help, including very detailed discussions on its use.

With what we have learned so far, we should be ready to develop a real GUI-based application, which is the GUI version of a grade conversion program. The conversion is based on the data in Table 9-1, and the case study is in Table 9-2.

Table 9-1: Grade conversion table

Letter grade	%	Letter grade	%
A+	90–100	C+	67–69
A	85–89	C	64–66
A-	80-84	C-	60-63
B+	76–79	D+	55–59
B	73–75	D	50–54
B–	70–72	F	0–49

Table 9-2: Case study: How to convert grades in a GUI program

The problem	To develop a GUI-based application that can take the a numerical grade from a user and convert it to a letter grade.
The analysis and design	The application is meant to convert numerical grades to letter grades based on the conversion table shown as Table 9-1. The information flow is: Get a numerical grade from user → convert to a letter grade → display the letter grade In previous chapters, we showed how to develop a terminal-based application to do this conversion, where the input and output were all done on a standard terminal. In a GUI-based application, the input and output/display need to be done on a graphic interface. Consider the widget in the tkinter module. We need an Entry widget for input and a Label widget to display the converted letter grade, a Button to start a conversion, and another Button to exit and close the application. Of course, we also need a function to do the actual conversion.

https://doi.org/10.15215/remix/9781998944088.01

The code

```
"""
A GUI-based program using the Tkinter module
to convert numeric grades to alpha letter/grades.
"""


from tkinter import *


def n_to_l(n_g):
  n_g = round(float(n_g))
  if n_g in range(90, 101):
    lg = f"Letter grade of {n_g} is A+"
  elif n_g in range(85, 90):
    lg = f"Letter grade of {n_g} is A"
  elif n_g in range(80, 85):
    lg = f"Letter grade of {n_g} is A-"
  elif n_g in range(76, 80):
    lg = f"Letter grade of {n_g} is B+"
  elif n_g in range(73, 76):
  lg = f"Letter grade of {n_g} is B"
  elif n_g in range(70, 73):
    lg = f"Letter grade of {n_g} is B-"
  elif n_g in range(67, 70):
    lg = f"Letter grade of {n_g} is C+"
  elif n_g in range(64, 67):
    lg = f"Letter grade of {n_g} is C"
  elif n_g in range(60, 64):
    lg = f"Letter grade of {n_g} is C-"
  elif n_g in range(55, 60):
    lg = f"Letter grade of {n_g} is D+"
  elif n_g in range(50, 54):
    lg = f"Letter grade of {n_g} is D"
  elif n_g in range(0, 50):
    lg = f"Letter grade of {n_g} is F"
  else:
    lg = "invalid mark!"
  return lg


def hdl():
  n = int(ent1.get())
  lb2.config(text = f"{n_to_l(n)}")
  return -1


w = Tk()
w.title("My GUI-based grade converter")
w.geometry("500 x 200")
lb1 = Label(text = "Please input percentage grade:")
lb1.grid(column = 2, row = 1, rowspan = 3, pady = 30)
ent1 = Entry()
ent1.grid(column = 3, row = 1, rowspan = 3, pady = 30)
btn = Button(text = "Convert", background = "#00FF00")
btn.grid(column = 2, row = 5, rowspan = 3, pady = 30)
btn.config(command = hdl)
btn_quit = Button(text = "Quit", background = "#FF0000")
btn_quit.grid(column = 6, row = 5, rowspan = 3, pady =
  30)
btn_quit.config(command=quit)
lb2 = Label(text = "Letter grade will be displayed here")
lb2.grid(column = 2, row = 8, rowspan = 3)
mainloop()
# this statement must be placed here to keep the window
  alive
```

https://doi.org/10.15215/remix/9781998944088.01

In the coding example, the .grid() method positions a widget in the parent widget in a grid or cell. The options that can be used with the .grid() method are as follows:

column = number—uses cell identified with given column (starting with 0)
columnspan = number—spans several columns
in = master—uses master to contain this widget
in_ = master—uses master to contain this widget
ipadx = amount—adds internal padding in x direction
ipady = amount—adds internal padding in y direction
padx = amount—adds padding in x direction
pady = amount—adds padding in y direction
row = number—uses a cell identified with a given row (starting with 0)
rowspan = number—span several rows
sticky = NSEW—tells on which sides this widget will stick to the cell boundary if the cell is larger

Note that we placed a method of the imported tkinter module named .mainloop() at the very end of the program. The call to the method is necessary to keep the window of the GUI-based app alive when running the code from the Python program file. Otherwise, the window will disappear as soon as the Python Virtual Machine (PVM) reaches the end of the program file.

You may wonder why the GUI window we created from the Python interactive shell stays alive without the .mainloop() statement. The reason is very simple: because PVM is still waiting for you to input a numeric grade, which means that the file (standard IO file) won't reach the end as long as you have not quit the Python interactive shell.

The GUI application can also be placed in a frame. The result and the code are shown in Figure 9-7 and below. As you can see, it has become nicer.

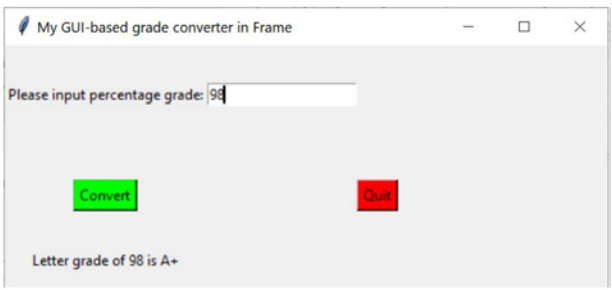


Figure 9-7: A GUI application doing grade conversion

https://doi.org/10.15215/remix/9781998944088.01

```
"""
You are required to design a GUI-based program using the
Tkinter
module to convert numeric grades to alpha letter/grades.
"""
from tkinter import *

def n_to_l(n_g):
  n_g = round(float(n_g))
  if n_g in range(90, 101):
    lg = f"Letter grade of {n_g} is A+"
  elif n_g in range(85, 90):
    lg = f"Letter grade of {n_g} is A"
  elif n_g in range(80, 85):
    lg = f"Letter grade of {n_g} is A-"
  elif n_g in range(76, 80):
    lg = f"Letter grade of {n_g} is B+"
  elif n_g in range(73, 76):
    lg = f"Letter grade of {n_g} is B"
  elif n_g in range(70, 73):
    lg = f"Letter grade of {n_g} is B-"
  elif n_g in range(67, 70):
    lg = f"Letter grade of {n_g} is C+"
  elif n_g in range(64, 67):
    lg = f"Letter grade of {n_g} is C"
  elif n_g in range(60, 64):
    lg = f"Letter grade of {n_g} is C-"
  elif n_g in range(55, 60):
    lg = f"Letter grade of {n_g} is D+"
  elif n_g in range(50, 54):
    lg = f"Letter grade of {n_g} is D"
  elif n_g in range(0, 50):
    lg = f"Letter grade of {n_g} is F"
  else:
    lg = "invalid mark!"
  return lg

def hdl():
  n = int(ent1.get())
  lb2.config(text = f"{n_to_l(n)}")
```

https://doi.org/10.15215/remix/9781998944088.01

```
  return -1

w = Frame()
# to add frame widget

w.master.title("My GUI-based grade converter in Frame")
w.master.geometry("500 x 200")
lb1 = Label(text = "Please input percentage grade:")
lb1.grid(column = 2, row = 1, rowspan = 3, pady = 30)
ent1 = Entry()
ent1.grid(column = 3, row = 1, rowspan = 3, pady = 30)
btn = Button(text = "Convert", background = "#FF0000")
btn.grid(column = 2, row = 5, rowspan = 3, pady = 30)
btn.config(command = hdl)
btn_quit = Button(text = "Quit", background = "#FF0000")
btn_quit.grid(column = 6, row = 5, rowspan = 3, pady =
30)
btn_quit.config(command = quit)
lb2 = Label(text = "Letter grade will be displayed here")

lb2.grid(column = 2, row = 8, rowspan = 3)
mainloop()
# this statement must be placed below all others to keep
the window alive
```

If you don't like placing widgets with the .geometry() method, you may use the .pack() method to let the PVM automatically place the widgets for you. The new GUI-based app and the revised Python code are shown in Figure 9-8 and below.

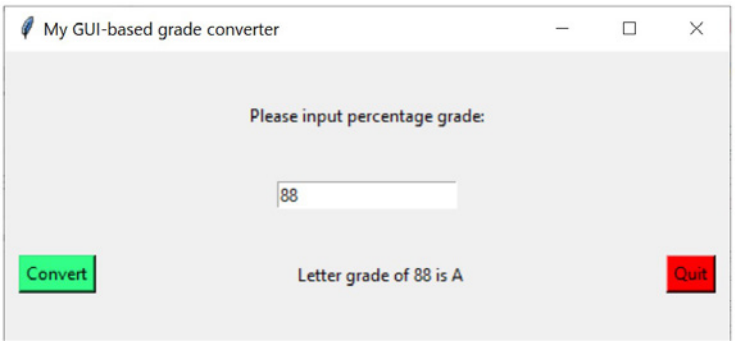


Figure 9-8: Frame widget is used in the GUI application

https://doi.org/10.15215/remix/9781998944088.01

```
"""Revised grade conversion: placing widgets with the
.geometry() method."""

from tkinter import *

def n_to_l(n_g):
  n_g = round(float(n_g))
  if n_g in range(90, 101):
    lg = f"Letter grade of {n_g} is A+"
  elif n_g in range(85, 90):
    lg = f"Letter grade of {n_g} is A"
  elif n_g in range(80, 85):
    lg = f"Letter grade of {n_g} is A-"
  elif n_g in range(76, 80):
    lg = f"Letter grade of {n_g} is B+"
  elif n_g in range(73, 76):
    lg = f"Letter grade of {n_g} is B"
  elif n_g in range(70, 73):
    lg = f"Letter grade of {n_g} is B-"
  elif n_g in range(67, 70):
    lg = f"Letter grade of {n_g} is C+"
  elif n_g in range(64, 67):
    lg = f"Letter grade of {n_g} is C"
  elif n_g in range(60, 64):
    lg = f"Letter grade of {n_g} is C-"
  elif n_g in range(55, 60):
    lg = f"Letter grade of {n_g} is D+"
  elif n_g in range(50, 54):
    lg = f"Letter grade of {n_g} is D"
  elif n_g in range(0, 50):
    lg = f"Letter grade of {n_g} is F"
  else:
    lg = "invalid mark!"
  return lg

def hdl():
  n = int(ent1.get())
  lb2.config(text = f"{n_to_l(n)}")
  return -1
```

https://doi.org/10.15215/remix/9781998944088.01

```
w = Tk()
w.title("My GUI-based grade converter")
w.geometry("500 x 200")
lb1 = Label(text = "Please input percentage grade:")
lb1.pack(fill = Y, expand = 1)
ent1 = Entry()
ent1.pack(expand = 0)
btn_convert = Button(text = "Convert", background =
"#33FF88")
btn_convert.pack(padx = 10, side = LEFT, expand = 0)
btn_convert.config(command = hdl)
btn_quit = Button(text = "Quit", background = "#FF0000")
btn_quit.pack(padx = 10, side = RIGHT, expand = 0)
btn_quit.config(command = quit)
lb2 = Label(text = "Letter grade will be displayed
here")
lb2.pack(fill = Y, expand = 1)
mainloop()
# this statement must be placed at the end to keep the
window alive
```

As you can see, the GUI-based app is rendered much more nicely with the .pack() method. The .pack() method is used to pack a widget in the parent widget. The options of the .pack() method and their meaning are as follows:

- after = widget—packs it after you have packed the widget
- anchor = NSEW (or subset)—positions widget according to a given direction
- before = widget—packs it before you pack the widget
- expand = bool—expands the widget if the parent size grows
- fill = NONE or X or Y or BOTH—fills the widget if the widget grows
- in = master—uses master to contain the widget
- in_= f—uses fx to contain the widget
- ipadx = amount—adds internal padding in x direction
- ipady = amount—adds internal padding in y direction
- padx = amount—adds padding in x direction
- pady = amount—adds padding in y direction
- side = TOP or BOTTOM or LEFT or RIGHT—indicates where to add the widget

https://doi.org/10.15215/remix/9781998944088.01

Because all the widgets inherit behaviours from the Widget class, it is worth learning about the general methods available in the Widget class for manipulating widgets.

tkinter.ttk—Tk-Themed Widgets

In the past, people have complained about the look of GUI and the widgets of tkinter, which led to the development of themed widgets of tkinter, or Ttk for short. Now the Ttk module is part of the standard Python distribution. To use it, all you need to do is to import the modules in the sequence shown below to make sure Ttk overrides definitions of classes as intended.

```
>>> from tkinter import *
>>> import tkinter.ttk
```

Because themed widgets is based on the tkinter module, it overrides definitions of classes, widgets, and methods.

Again, once imported, you can use help(tkinter.ttk) to get a rather detailed documentation of the module, as summarized below:

As for all GUI libraries, the core of the module are the widgets and methods associated with the widgets. All widget classes inherit from a generic class called Widget. The following is a list of widgets defined in the themed Tk module.

BUTTON(WIDGET)

Ttk Button displays a textual label and/or image and evaluates a command when pressed. In addition to the methods inherited from Widget, it has method specific for its purpose, invoke(self), which invokes the command associated with the button.

CHECKBUTTON

Ttk Checkbutton will be either in an on or off state. It has a specific method called invoke(self) in addition to those inherited from Widget. The invoke(self) will switch the Checkbutton between on and off states, further invoke the associated command, and return the result of executing the command associated with Checkbutton.

ENTRY(WIDGET, TKINTER.ENTRY)

Ttk Entry displays a one-line text string that allows that string to be edited by the user. It inherits from Widget and tkinter.Entry.

https://doi.org/10.15215/remix/9781998944088.01

COMBOBOX(ENTRY)

Ttk Combobox widget combines a text field with a pop-down list of values. It inherits from Ttk Entry, described above. It has two methods of its own:

1. current(self, newindex = None)—if newindex is supplied, it sets the combobox value to the element at position newindex in the list of values. Otherwise, it returns the index of the current value in the list of values or −1 if the current value does not appear in the list.
2. set(self, value)—sets the value of the combobox to "value."

SPINBOX(ENTRY)

Ttk Spinbox is an Entry with increment and decrement arrows. It is commonly used for number entry or to select from a list of string values.

FRAME(WIDGET)

Ttk Frame is a container used to group other widgets together.

LABELEDSCALE(FRAME)

Ttk Scale is used with Ttk Label, indicating its current value. Ttk Scale can be accessed through instance.scale, and Ttk Label can be accessed through instance.label.

LABEL(WIDGET)

Ttk Label displays a textual label and/or image.

LABELFRAME(WIDGET)

Ttk Labelframe is a container used to group other widgets together. It has an optional label, which may be a plaintext string or another widget.

MENUBUTTON(WIDGET)

Ttk Menubutton displays a textual label and/or image. It will display a menu when pressed.

OPTIONMENU(MENUBUTTON)

Ttk OptionMenu allows the user to select a value from a menu.

NOTEBOOK(WIDGET)

Ttk Notebook manages a collection of windows and displays a single one at a time. Each child window is associated with a tab that the user may select to

https://doi.org/10.15215/remix/9781998944088.01

make the associated window show up. This gives us a way to implement tabs like those in web browsers.

PANEDWINDOW(WIDGET, TKINTER.PANEDWINDOW)

Ttk Panedwindow displays a number of subwindows stacked either vertically or horizontally. It has the following specific methods:

1. remove(self, child)
2. insert(self, pos, child, **kw)—inserts a pane at the specified positions.
3. pane(self, pane, option = None, **kw)—queries or modifies the options of the specified pane.
4. sashpos(self, index, newpos = None)—if newpos is specified, sets the position of sash number index and returns the new position of sash number index.

PROGRESSBAR(WIDGET)

Ttk Progressbar shows the status of a long-running operation. It can operate in two modes. Determinate mode shows the amount completed relative to the total amount of work to be done, and indeterminate mode provides an animated display to let the user know that something is happening.

RADIOBUTTON

Ttk Radiobuttons are used in groups to show or change a set of mutually exclusive options. The specific method invoke(self) sets the option variable to the option value, selects the widget, and invokes the associated command. It returns the result of the command or an empty string if no command is specified.

SCALE(WIDGET, TKINTER.SCALE)

Ttk Scale is typically used to control the numeric value of a linked variable that varies uniformly over some range.

SCROLLBAR(WIDGET, TKINTER.SCROLLBAR)

Ttk Scrollbar controls the viewport of a scrollable widget.

SEPARATOR(WIDGET)

Ttk Separator displays a horizontal or vertical separator bar.

SIZEGRIP(WIDGET)

Ttk Sizegrip allows the user to resize the containing top-level window by pressing and dragging the grip.

https://doi.org/10.15215/remix/9781998944088.01

TREEVIEW(WIDGET, TKINTER.XVIEW, TKINTER.YVIEW)

Ttk Treeview displays a hierarchical collection of items. Each item has a textual label, an optional image, and an optional list of data values. The data values are displayed in successive columns after the tree label. It has the following specific methods:

- ***bbox(self, item, column = None)***
  It returns the bounding box (relative to the Treeview widget's window) of the specified item in the form of x y width height. If a column is specified, it returns the bounding box of that cell. If the item is not visible (i.e., if it is a descendant of a closed item or is scrolled off-screen), it returns an empty string.
- ***column(self, column, option = None, **kw)***
  It queries or modifies the options for the specified column. If kw is not given, it returns a dictionary of the column option values. If option is specified, then the value for that option is returned. Otherwise, it sets the options to the corresponding values.
- ***delete(self, *items)***
  It deletes all specified items and all their descendants. The root item may not be deleted.
- ***detach(self, *items)***
  It unlinks all of the specified items from the tree. The items and all of their descendants are still present and may be reinserted at another point in the tree but will not be displayed. The root item may not be detached.
- ***exists(self, item)***
  It returns True if the specified item is present in the tree; returns False otherwise.
- ***focus(self, item = None)***
  if an item is specified, it sets the focus item to item. Otherwise, it returns the current focus item, or '' if there is none.
- ***get_children(self, item = None)***
  It returns a tuple of children belonging to item. If the item is not specified, it returns root children.
- ***heading(self, column, option = None, **kw)***
  It queries or modifies the heading options for the specified column. If kw is not given, it returns a dict of the heading option values. If option is specified, then the value for that option is returned. Otherwise, it sets the options to the corresponding values. Valid options/values include text, image, anchor, and a callback command.

https://doi.org/10.15215/remix/9781998944088.01

- ***identify(self, component, x, y)***
  It returns a description of the specified component under the point given by x and y or the empty string if no such component is present at that position.
- ***identify_column(self, x)***
  It returns the data column identifier of the cell at position x. The tree column has ID #0.
- ***identify_element(self, x, y)***
  It returns the element at position x, y.
- ***identify_region(self, x, y)***
  For the given coordinator (x, y) of a point related to the widget, it returns a string indicating one of the following: nothing (not within any functional part of the widget), heading (a column heading), separator (a separator between columns), tree (the icon column), or cell (a data cell within an item row).
- ***identify_row(self, y)***
  It returns the item ID of the item at position y.
- ***index(self, item)***
  It returns the integer index of an item within its parent's list of children.
- ***insert(self, parent, index, iid = None, **kw)***
  It creates a new item and returns the item identifier of the newly created item, where parent is the item ID of the parent item or the empty string to create a new top-level item and index is an integer or the value end, specifying where in the list of parent's children to insert the new item. If index is less than or equal to 0, the new node is inserted at the beginning; if index is greater than or equal to the current number of children, it is inserted at the end. If iid is specified, it is used as the item identifier, but iid must not already exist in the tree. Otherwise, a new unique identifier is generated.
- ***item(self, item, option = None, **kw)***
  It queries or modifies the options for a specified item. If no options are given, a dict with options/values for the item is returned. If an option is specified, then the value for that option is returned. Otherwise, it sets the options to the corresponding values, as given by kw.
- ***move(self, item, parent, index)***
  It moves item to position index in parent's list of children. It is illegal to move an item under one of its descendants. If an index is less than or equal to 0, the item is moved to the beginning; if greater than or

https://doi.org/10.15215/remix/9781998944088.01

equal to the number of children, it is moved to the end. If the item was detached, it is reattached.

- ***next(self, item)***
  It returns the identifier of an item's next sibling, or '' if the item is the last child of its parent.
- ***parent(self, item)***
  It returns the ID of the parent of an item, or '' if the item is at the top level of the hierarchy.
- ***prev(self, item)***
  It returns the identifier of an item's previous sibling, or '' if the item is the first child of its parent.
- ***see(self, item)***
  It ensures that the item is visible and sets all of the item's ancestors' open options to True and scrolls the widget if necessary so that the item is within the visible portion of the tree.
- ***selection(self, selop = <object object at 0x000002122111D7F0>, items=None)***
  It returns the tuple of the selected items.
- ***selection_add(self, *items)***
  It adds all of the specified items to the selection.
- ***selection_remove(self, *items)***
  It removes all of the specified items from the selection.
- ***selection_set(self, *items)***
  It makes the specified items become the new selection.
- ***selection_toggle(self, *items)***
  It toggles the selection state of each specified item.
- ***set(self, item, column = None, value = None)***
  It queries or sets the value of a given item.
- ***set_children(self, item, *newchildren)***
  It replaces an item's child with newchildren.
- ***tag_bind(self, tagname, sequence = None, callback = None)***
  It binds a callback for the given event sequence to the tag tagname. When an event is delivered to an item, the callbacks for each tag option of the item are called.
- ***tag_configure(self, tagname, option = None, **kw)***
  It queries or modifies the options for the specified tagname.
- ***tag_has(self, tagname, item = None)***
  If an item is specified, it returns 1 or 0 depending on whether the specified item has the given tagname. Otherwise, it returns a list of all items that have the specified tag.

https://doi.org/10.15215/remix/9781998944088.01

The Treeview widget provides a way to use tree-like structures to visualize and manipulate data and information. The methods available for manipulating trees are versatile.

Among all the widgets provided in themed tkinter, the newly added Treeview, Progressbar, and Notebook are very powerful and can be very useful in developing today's GUI applications, such as those in the exercises, and projects below.

The content above Ttk and Treeview in particular are mostly taken from the official documentation of Tkinter, for the purpose to prepare readers for the projects at the end of this chapter.

For details on how Ttk can be used to develop GUI applications, please read the Ttk documentation at https://docs.python.org/3/library/tkinter.ttk.html and tutorials online such as the one at https://www.pythontutorial.net/tkinter/tkinter-ttk/.

Instead of giving code samples for each widget like we previously did, we conclude this chapter with the following sample program to show how Ttk and Treeview can be used in GUI application development:

```
import tkinter as tk
from tkinter import ttk

class FileManager:
    def __init__(self):
        self.root = tk.Tk()
        self.tree = ttk.Treeview(self.root)
        self.tree.pack()
        self.tree["columns"] = ("one", "two", "three",
"four")
        self.tree.heading("one", text="Path")
        self.tree.heading("two", text="File name")
        self.tree.heading("three", text="File size")
        self.tree.heading("four", text="Last modified")
        self.tree.insert("", "end", text="1", values=("/
home/james/", "a.txt","213","June 3, 2023" ))
        self.tree.insert("", "end", text="2", values=("/
home/james/", "b.txt","215","June 5, 2023" ))
        self.tree.insert("", "end", text="3", values=("/
home/james/", "c.txt","217","June 7, 2023" ))
        self.root.mainloop()

FileManager()
```

https://doi.org/10.15215/remix/9781998944088.01

Chapter Summary

- Terminal-based applications are those started and rendered within a terminal.
- A terminal is a system command-line-based application in which you can interact with a computer.
- On the Windows platform, Windows terminal, Windows PowerShell, and command prompt are examples of terminals.
- Within a terminal-based application, pointing devices such as a mouse cannot be used to interact with the application.
- A graphical user interface is a two-dimensional graphic area in which graphic objects or widgets can be created, placed, located, and accessed with mouse and keyboard.
- GUI-based applications look nicer and are much more user-friendly.
- Python has some very well-developed library modules for the development of applications with graphical user interface.
- The Tk and Themed Tk (Ttk) are the modules covered in this text; both come with the Tkinter package.
- The Tk module provides the fundamental widgets and operations needed for a graphic interface.
- To use the Tk module, use ***import tkinter***, ***import tkinter as tk***, or ***from tkinter import. ****.
- The Themed Tk (Ttk) module provides programmers with more styled widgets to develop applications with a nicer graphical user interface.
- To ensure you are using the Ttk module instead of Tk module, override Tk by running statements ***from tkinter import ** and ***from tkinter.ttk import ** in order.
- To build a graphical user interface, create a main frame or window first.
- Other widgets can be created and added to the main frame/window as needed.
- Each widget has certain properties/attributes such as size, colour, and more.
- The properties of a widget can be set when created and changed later.
- A widget can be placed at particular location within the main frame or subframe.
- A function/method can be bound to widgets, such as a button, to help users interact with the graphical user interface.
- A graphical user interface needs to be rendered by calling the mainloop() method of main frame object.

https://doi.org/10.15215/remix/9781998944088.01

Exercises

1. Explore the relationship between Tk and Ttk, and explain what the following code does:
   from tkinter import *
   from tkinter.ttk import *
2. Write a script that will render a window as shown in Figure 9-9.

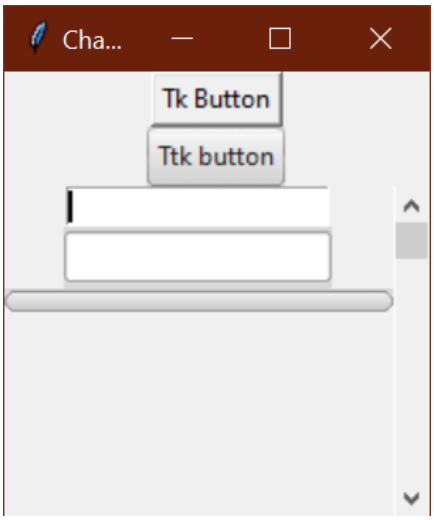


Figure 9-9: Required GUI interface for the exercise

3. Write a script that will render a page for login or registration, as shown in Figure 9-10.

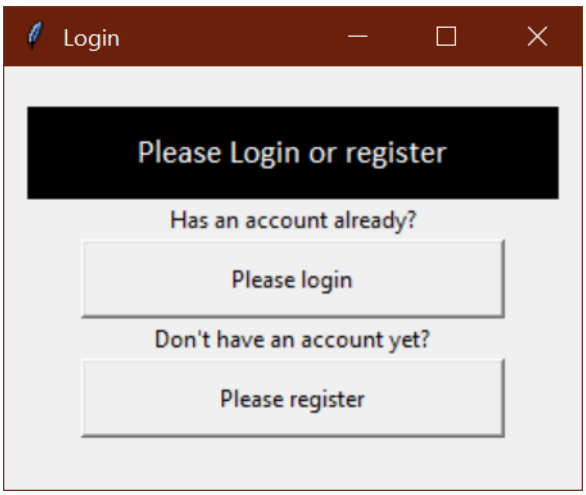


Figure 9-10: Required GUI interface for the project

https://doi.org/10.15215/remix/9781998944088.01

Projects

1. For this project, develop a GUI-based music player with the module mixer.music as well as the os and os.path modules for file navigation. The player should have a panel for the navigation and selection of music files, as well as buttons to control play, stop, and pause.
2. For this project, develop a course management system with a graphical user interface that meets the following requirements:
   a. Define a student class modelling a student, including their name, student id, and start date in the class, as well as the name of the tutor assigned to them and a list of their assessment records. Each assessment record will be a tuple containing a number to identify the assessment, the weight of the assessment, and the mark, which should be 0 for a newly added instance to a course.
   b. Define a course class modelling a course, including its course number (such as comp218), title, and revision number, as well as the date of its initial offering and a list of its students, professor(s). Additionally, include an assessment schedule as a list of assessment items where each assessment item is represented as a tuple of (id, name, weight) in which the id is the assessment item id, the name is the assessment name such as assignment 1 or final exam, and the weight is the percentage of the assessment that will go toward the final grade.
   c. The GUI program should include and do all of the following:
      - There should be a button for adding a new course to the system, which will open a form for input and save the basic course info (previously mentioned), with the list of students as empty. Note that when saving the basic course info, the system should be able to check whether the weight of all the assessment items makes up 100%.
      - When a new course is added to the system, a unique binary file will be created as permanent storage of the course data.
      - At the start of the system, it should automatically load all courses from their respective binary files to restore their internal object representation.
      - There should be a button to get a list of courses being offered to students.
      - A user should be able to select a course from the list.
      - A user should be able to add students to the selected course.
      - A user should be able to see a list of students in the course.

https://doi.org/10.15215/remix/9781998944088.01

- A user should be able to select a student from the list.
- A user should be able to record an assessment for the selected student.
- The system should be able to automatically calculate and display the final grade of the student for the course.
- A user should be able to see a list of assessments, including the calculated final grade for the selected student.
- There should be a button to shut down the system, but before shutting down the application, the system must save/pickle each piece of course data back to its binary file.

d. Your analysis and design of the system should be well documented in your assignment report.

e. Within each of your program files, there should be a docstring at the beginning stating the name and purpose of the file, as well as its ownership and revision history. One docstring is required for each class and function/method class. An end-of-line comment is desired when deemed necessary.

3. In Project 1 of Chapter 7, you developed a terminal-based quiz system. For this project, you are required to develop a GUI-based quiz system also using the Quiz_item and Quiz classes you wrote in Chapter 7's Exercises 7 and 8. The system should have controls/widgets on the GUI that will allow a user to do the following:
   a. Create a quiz.
   b. Select a quiz and add new quiz items.
   c. Select a quiz and preview all the quiz items.
   d. Select a quiz and execute the quiz by presenting the quiz items one by one.
4. Modify the quiz system developed for Project 3 so that after the quiz, it will present to the user the score as a percentage and the correct answers to incorrectly answered questions. The quiz items should be displayed to the user one by one during the quiz. Add a timer to the quiz so that the entire quiz must be done within a given time. To make sense of the timer function, you will need to show the total number of quiz items, the number of items left to be answered, the total amount of time allocated for the entire quiz, and the amount of time left for the remaining quiz items.
5. Further modify the quiz system developed for Projects 3 and 4, so that not only is the quiz timed, but each quiz item is also timed too, meaning that the user must answer each quiz question within a given time. For simplicity, we can assume that the time allocated to each

https://doi.org/10.15215/remix/9781998944088.01

quiz item is the same and equal to the total time allocated to the quiz divided by the total number of questions in the quiz. To make better sense of both timers now, the time (number of seconds) left for a quiz question also needs to be displayed, and the quiz will move to the next question as soon as the question is answered or the timer for the question has run out.

https://doi.org/10.15215/remix/9781998944088.01