ESSENTIAL PYTHON

SUCCINCTLY®

BY ED FREITAS

SUCCINCTLY® EBOOK SERIES

Syncfusion®

Essential Python Succinctly

Ed Frietas

Foreword by Daniel Jebaraj

2501 Aerial Center Parkway
Suite 111
Morrisville, NC 27560
USA

ISBN: 978-1-64200-249-2

Technical Reviewer: James McCaffrey
Copy Editor: Courtney Wright
Acquisitions Coordinator: Graham High, content team lead, Syncfusion, Inc.
Proofreader: Naomi Spicer, content producer, Syncfusion, Inc.

Table of Contents

The *Succinctly* Series of Books

Daniel Jebaraj
CEO of Syncfusion, Inc.

When we published our first *Succinctly* series book in 2012, *jQuery Succinctly*, our goal was to produce a series of concise technical books targeted at software developers working primarily on the Microsoft platform. We firmly believed then, as we do now, that most topics of interest can be translated into books that are about 100 pages in length.

We have since published over 200 books that have been downloaded millions of times. Reaching more than 2.7 million readers around the world, we have more than 70 authors who now cover a wider range of topics, such as Blazor, machine learning, and big data.

Each author is carefully chosen from a pool of talented experts who share our vision. The book before you and the others in this series are the result of our authors' tireless work. Within these pages, you will find original content that is guaranteed to get you up and running in about the time it takes to drink a few cups of coffee.

We are absolutely thrilled with the enthusiastic reception of our books. We believe the *Succinctly* series is the largest library of free technical books being actively published today. Truly exciting!

Our goal is to keep the information free and easily available so that anyone with a computing device and internet access can obtain concise information and benefit from it. The books will always be free. Any updates we publish will also be free.

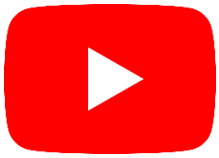
Let us know what you think

If you have any topics of interest, thoughts, or feedback, please feel free to send them to us at succinctlyseries@syncfusion.com.

We sincerely hope you enjoy reading this book and that it helps you better understand the topic of study. Thank you for reading.

Please follow us on social media and help us spread the word about the *Succinctly* series!



About the Author

Ed Freitas is a business process automation consultant and a software developer focused on customer success.

He likes technology and enjoys learning, playing soccer, running, traveling, and being around his family.

Ed is available at https://edfreitas.me.

Acknowledgments

Thank you to the fantastic Syncfusion team that helped this book become a reality—especially Jacqueline Bieringer, Tres Watkins, and Graham High.

The Manuscript Managers and Technical Editor thoroughly reviewed the book's organization, code quality, and overall accuracy: Jacqueline Bieringer, Graham High from Syncfusion, and James McCaffrey from Microsoft Research. Thank you all.

I dedicate this book to my beloved Chelin, Dudu, and Yego. May your journeys be blessed.

Introduction

Welcome to this comprehensive guide to mastering one of the world's most versatile and powerful programming languages.

Python's significance in the modern tech landscape cannot be overstated. From its humble beginnings in the late 1980s, Python has evolved into an essential tool for developers, data scientists, and tech enthusiasts, revolutionizing fields such as web development, artificial intelligence, and data analysis.

Python's popularity stems from its simplicity and readability, making it an ideal choice for beginners and experienced programmers. Its elegant syntax allows for rapid development and reduces the learning curve, enabling you to focus on solving problems rather than grappling with complex code.

The language's philosophy emphasizes code readability and simplicity, translating into fewer lines of code to accomplish tasks, making development faster and maintenance easier. Moreover, Python's extensive standard library and vibrant ecosystem of third-party packages offer robust solutions for virtually any task you can imagine.

Whether you're dealing with web frameworks like Django or Flask, data analysis libraries such as Pandas or NumPy, or machine learning tools like TensorFlow and scikit-learn, Python has you covered.

This rich ecosystem makes Python a go-to language for professionals across various industries, from finance and healthcare to entertainment and education. Python's role is indispensable in a world increasingly driven by data and automation.

Python is the backbone of many data science and machine learning frameworks, empowering organizations to get insights from vast datasets and build intelligent applications. Companies like Google, Facebook, and Netflix rely on Python for its efficiency and scalability, enabling them to innovate and stay ahead of the competition.

Furthermore, Python's capabilities in scripting and automation allow developers to streamline workflows, reduce errors, and increase productivity, making it a valuable asset in any workplace.

For researchers and academics, Python provides powerful tools for simulation and analysis, enabling groundbreaking discoveries in fields ranging from biology to astrophysics. In education, Python's intuitive features make it an excellent choice for teaching programming concepts to students of all ages, fostering the next generation of tech-savvy individuals.

Additionally, Python's interoperability with other programming languages and platforms ensures that it can be seamlessly integrated into diverse tech stacks, making it a versatile tool in any developer's toolkit.

Whether you're working on a small personal project or a large-scale enterprise application, Python's cross-platform capabilities and robust community support make it invaluable.

Furthermore, whether you aim to build dynamic websites, automate mundane tasks, analyze data, or venture into artificial intelligence, Python provides the tools and flexibility to bring your ideas to life.

This book's goal is to equip you with the essential knowledge and skills to harness the potential of this remarkable programming language, setting you on a path to success in the ever-evolving world of technology.

You'll notice that the book is written more like a cheat sheet than a full-blown application-building tutorial—this is intentional, given the breadth of Python.

The idea is to guide you in learning the essential aspects of Python to get you up and running quickly. For a topic so broad as Python, a cheat sheet-style approach is probably better suited to help you in that journey.

Welcome to your brief journey into Python! Let's get started.

Note: You can download this book's code samples from this GitHub repository.

Chapter 1 Getting Started

Installation

Before we can use Python, we must install it. We can get the latest installer from the Python downloads website.

> ***Note: An alternative to installing Python by itself is to install the Anaconda distribution, which contains a Python execution engine and approximately 500 compatible libraries, such as scikit-learn. Installing and using Anaconda is complicated and outside the scope of this book.***

Figure 1-a: Python Downloads Website

I usually go for the latest version of Python for Windows, which is the operating system I use. Select **Download Python** to proceed, and ensure you choose the version corresponding to your operating system.

Then, once the download has been completed, execute the installer. Because I already have a previous version of Python installed, I get this message to upgrade.

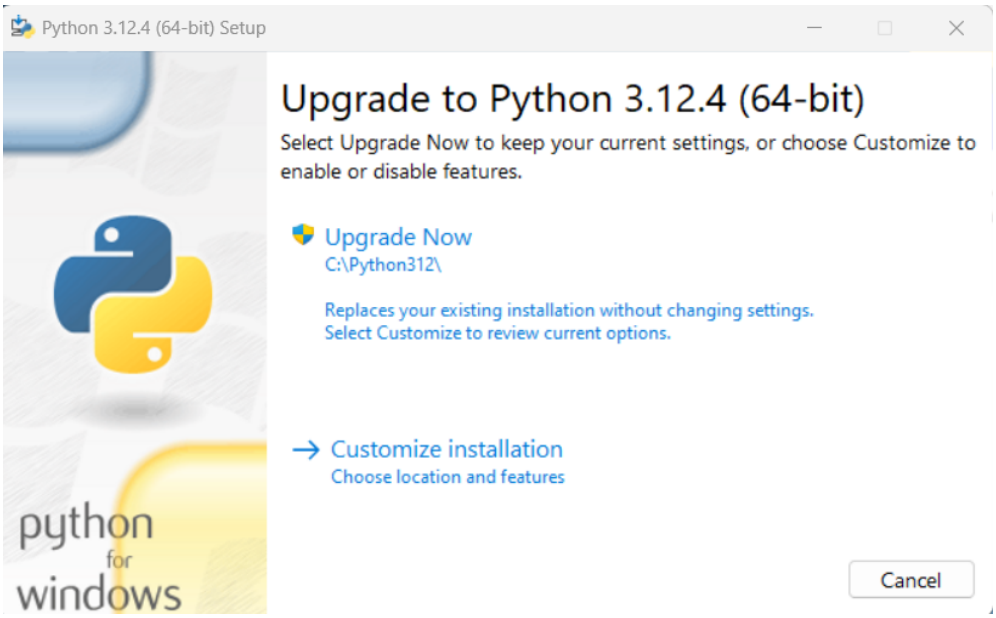


Figure 1-b: Python Installer

In some cases, the **Customize installation** option is an excellent choice because it allows you to adapt the Python installation to your environment settings and features.

In any case, whichever option you choose, the process is intuitive and straightforward, and the installation wizard will guide you.

For simplicity, and because I already have a previous version of Python installed, I'll go with the **Upgrade Now** option without changing settings.

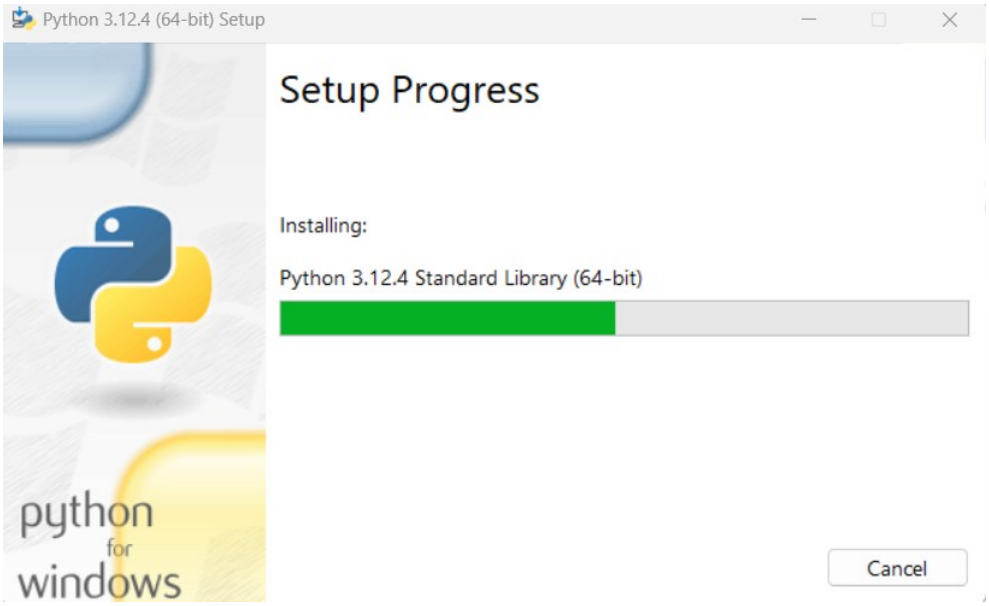


Figure 1-c: Installing (Upgrading) Python

Once the installation process has been finalized, you'll see a screen that looks similar to this one, and then you can click **Close**.

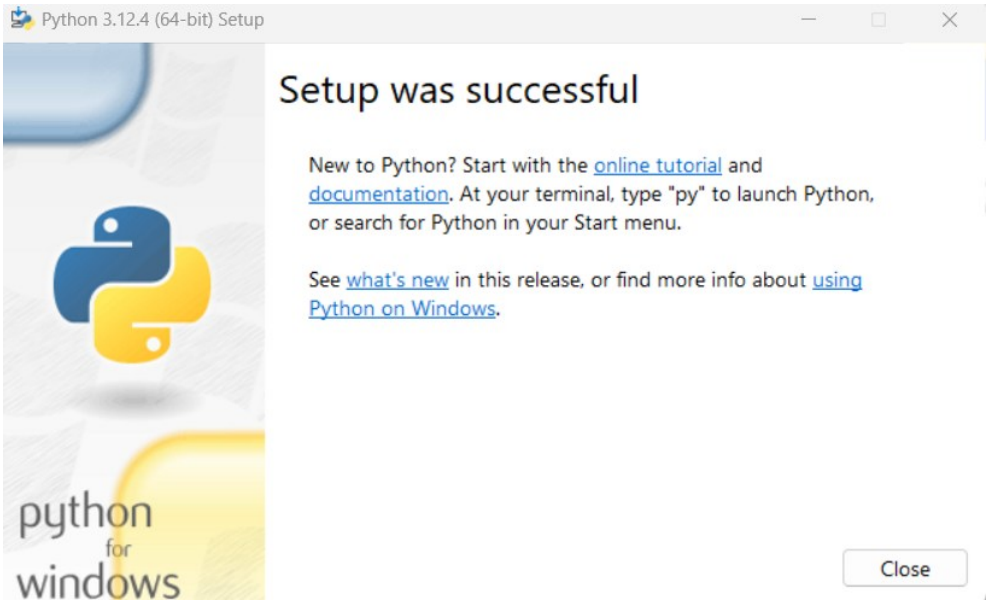


Figure 1-d: Python Installation Finalized

The other thing you'll want to do is to install an IDE or a programming editor. In my experience, the two best options available for Python development are Visual Studio Code (VS Code) and PyCharm.

I mostly use Visual Studio Code, so feel free to install it to follow along easily.

Using print

So, now that everything has been set up, let's start with the basics. I will create a new file called **basics.py** within the Visual Studio Code **Explorer**.

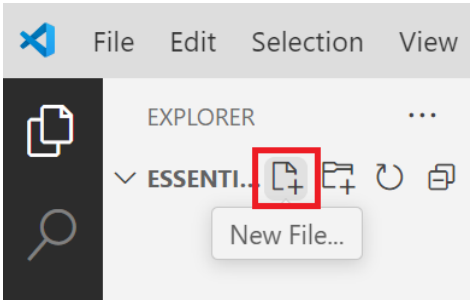


Figure 1-e: Create a New File (VS Code Explorer)

Once the **basics.py** file has been created, let's add the following code to print a message in the console.

Listing 1-a: The basics.py File (Printing a Message – Single Quotes)

```
print('Welcome to Python programming!')
```

As its name implies, the **`print`** statement outputs a message in Python. The text between single quotes (`''`) is the message that will be printed: **`Welcome to Python programming!`**

It is also possible to use double quotes (`""`) to print messages.

Listing 1-b: The basics.py File (Printing a Message Using Double Quotes)

```
print("Welcome to Python programming!")
```

Regardless of which quotes are used, the output will be the same. So, let's combine these instructions.

Listing 1-c: The basics.py File (Printing Messages)

```
print('Welcome to Python programming!')
print("Welcome to Python programming!")
```

To see this in action, let's execute this code. Go to the **Terminal** menu in Visual Studio Code and select **New Terminal**—this will open the console within VS Code.

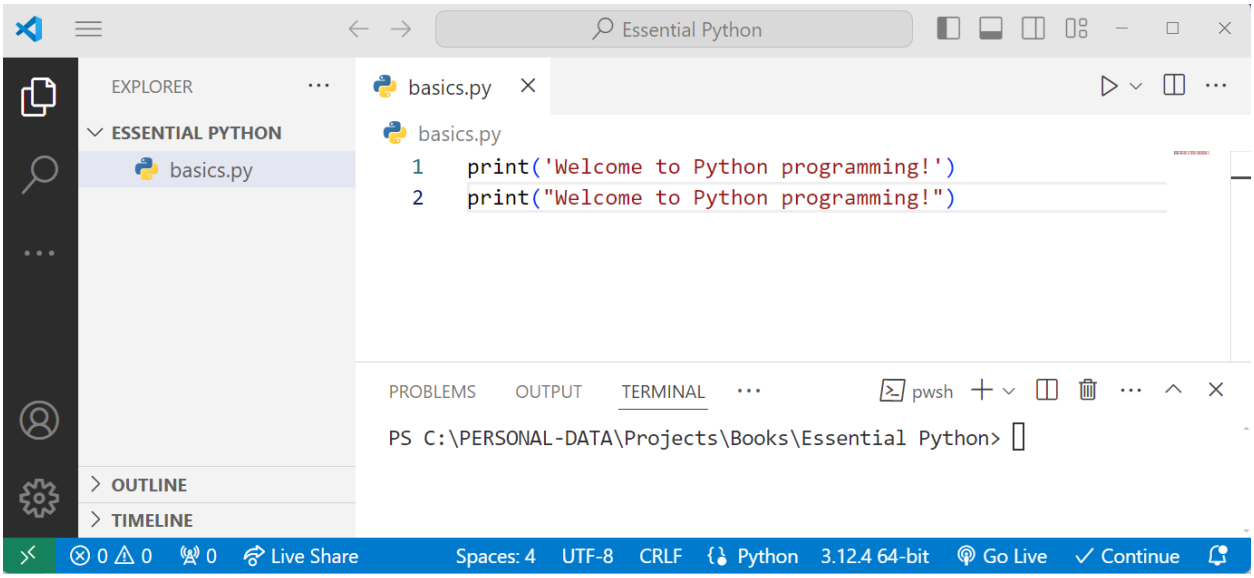


Figure 1-f: VS Code (TERMINAL Open)

Within the terminal, type the following command to run the code and press **Enter**.

Listing 1-d: Command to Execute the Python Script

```
python basics.py
```

Once done, you'll see the messages in the terminal. As you can see, the output was the same regardless of the quotes used.

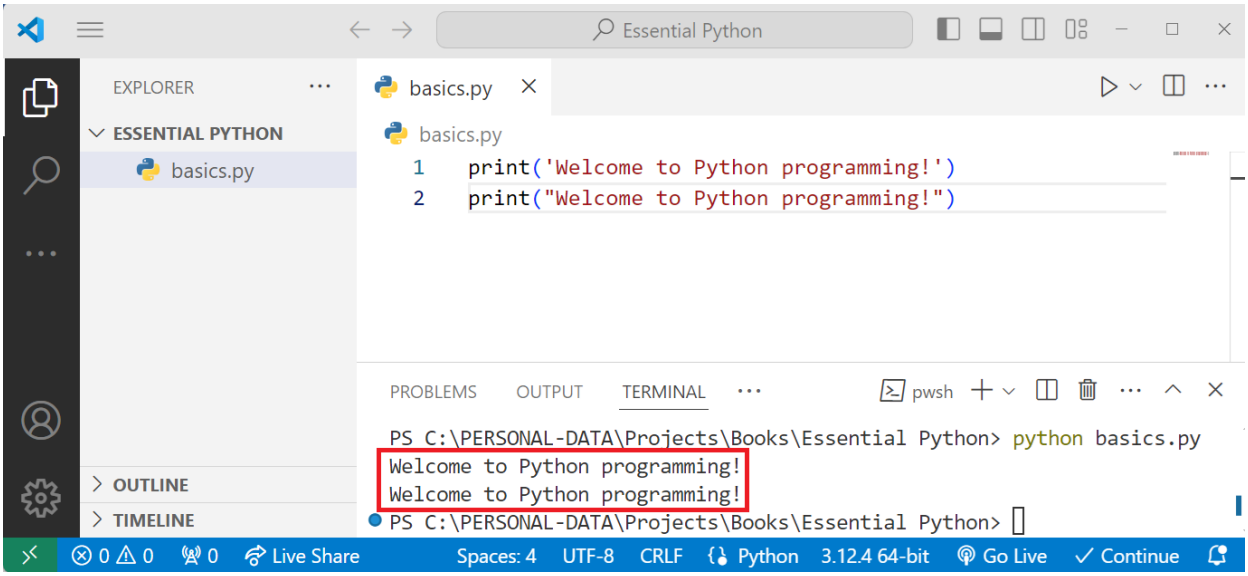


Figure 1-g: VS Code (Terminal Open with Messages)

Variables

A fundamental aspect of programming with Python is working with variables. A variable stores data that can be used later within a Python program.

Let's examine different variable types in Python. Delete all the previous code in the **basics.py** file and add the following code.

> ***Note: Comments in Python start with the # character. Python is a dynamically typed language, meaning you don't declare a variable's type. Python has an add-on library named typing that allows you to supply "type hints."***

> ***Note: You can run the following code examples by opening the built-in terminal within VS Code and executing the command:*** *python <script>.py*. ***Replace*** *<script>* ***with the name of the respective Python file to execute.***

Listing 1-e: Variable Types in Python (basics.py)

```
name = "Alice"  # String variable
```

```
anotherName = 'Alice'  # String variable
age = 30  # Integer variable
height = 5.5  # Float variable
is_student = True  # Boolean variable
```

First, we have a string variable called **name**, to which we assign **Alice** as a value using a string with double quotes (").

Then, we have a string variable called **anotherName**, to which we assign **Alice** as a value using a string with single quotes (').

Following that, we have an integer (number) variable called **age**, to which we assign **30** as a value. On the other hand, **height** is a float variable to which we assign **5.5** as a value.

Finally, we have the **is_student** variable, to which we assign a **True** value. The other value a boolean variable can have is **False**. Now, let's print those variables.

Listing 1-f: Printing the Variable Types in Python (basics.py)

```
print("Name:", name)
print("Another Name:", anotherName)
print("Age:", age)
print("Height:", height)
print("Is Student:", is_student)
```

We get the following output if we run the code by invoking the **python basics.py** command.

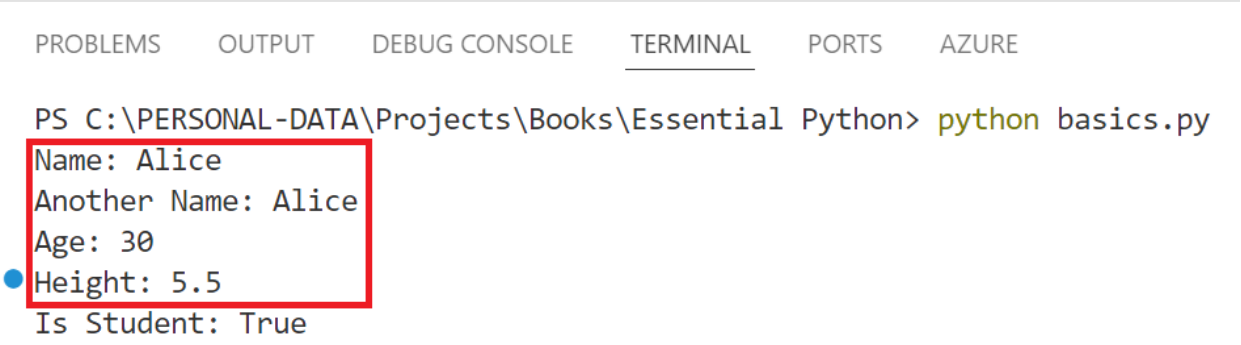


Figure 1-h: VS Code (Terminal Open with Printed Results)

As you can see, we can store values and later use them by using variables. In our case, we printed them, but as you'll see later, we can do other operations with variables.

Lists

Lists in Python are used to store multiple items in a single variable. Lists are ordered (meaning the items have a defined order), mutable (meaning you can change their content), and can hold different data types. Let's go ahead and create our first list.

Listing 1-g: Creating a List (basics.py)

```
fruits = ["apple", "banana", "cherry"]
```

This code creates a list named **fruits** with three string elements: **apple**, **banana**, and **cherry**.

Now, let's add some additional code on how to access an element within the list, and how to add an element to the list.

Listing 1-h: Lists - Accessing and Adding Elements (basics.py)

```
# Accessing elements in a list
print("First fruit:", fruits[0])

# Adding elements to a list
fruits.append("mango")

print("Fruits after adding date:", fruits)
```

Lists are zero-indexed, meaning the first element has an index of **0**, the second element has an index of **1**, and so on. In the previous code listing, **fruits[0]** accesses the first element in the list, **apple**.

The **fruits.append()** method adds a new item to the end of the list. In this case, **fruits.append("mango")** adds **mango** to the list. After this operation, **fruits** contain **["apple", "banana", "cherry", "mango"]**.

So, lists are a great way to store multiple items in a single variable, and they are instrumental in Python programming.

Loops

The **for** loop in Python iterates over each item in a collection (like a list) and executes the indented code block for each item. Let's begin by looking at the following code.

Listing 1-i: For Loop – Printing a List (basics.py)

```
# For loop
for fruit in fruits:
```

```
    print("Fruit:", fruit)
```

This code essentially prints the list of **fruits** we previously created. The **for fruit in fruits** statement iterates over the list of **fruits**. On each iteration, the variable **fruit** holds the current item's value in the list.

The **print("Fruit:", fruit)** statement, inside the **for** loop, prints the current value of **fruit**.

Let's explore the **while** loop by looking at the following code.

Listing 1-j: While Loop (basics.py)

```
count = 0
while count < 3:
    print("Count:", count)
    count += 1
```

The **while** loop continues to execute the indented block of code as long as the condition specified is **True**. Let's break this code down to understand it better.

- The statement **count = 0** initializes a variable count with **0**.
- **while count < 3** checks if **count** is less than **3**. If the condition is **True**, it executes the code block inside the loop.
- **print("Count:", count)** prints the current **count** value.
- **count += 1** increments count by **1**. The loop will stop executing once **count** reaches **3**.

Conditionals

Conditionals allow you to run different code blocks depending on whether a particular condition is **True** or **False**. Let's have a look at the following code.

Listing 1-k: Conditionals (basics.py)

```
if age < 18:
    print("Minor")
elif age < 65:
    print("Adult")
else:
    print("Senior")
```

Explanation:

- The **if age < 18** statement checks if **age** is less than **18**. If **True**, the code block inside the **if** statement is executed, printing **Minor**.

- The **elif age < 65** statement checks another condition if the previous **if** statement was **False**. If **age** is less than **65**, it prints **Adult**.
- The **else** statement catches all other cases. If none of the previous conditions were **True**, it prints **Senior**.

Functions

Functions are blocks of reusable code that perform a specific task. They help to modularize your code, making it easier to manage, test, and reuse. Let's look at a basic function.

Listing 1-l: Basic Function (basics.py)

```
def greet(name):
    """Greet a person by their name."""
    return f"Hello, {name}!"

print(greet("Bob"))
```

Explanation:

- **def greet(name)** defines a function named **greet** that takes a parameter **name**.
- **"""Greet a person by their name."""** is a docstring, a special kind of comment used to describe the purpose of the function.
- **return f"Hello, {name}!"** returns a string that includes the value of **name**.
- **print(greet("Bob"))** calls the greet function with **"Bob"** as the argument and prints the returned string **"Hello, Bob!"**.

Let's look at a function with multiple parameters and a default parameter value.

Listing 1-m: Function with Parameters (basics.py)

```
def add(a, b=5):
    """Add two numbers."""
    return a + b

print("Add 3 + 4:", add(3, 4))
print("Add 3 + default 5:", add(3))
```

This function takes multiple parameters, and you can also set default values for these parameters. Let's break this function down to understand what each part does.

- **def add(a, b=5)** defines a function **add** with two parameters, **a** and **b**. The parameter **b** has a default value of **5**. If no argument is provided for **b**, it will automatically be set to **5**.
- **return a + b** returns the sum of **a** and **b**.

- **`print("Add 3 + 4:", add(3, 4))`** calls the **`add`** function with **`3`** and **`4`** as arguments, so **`a=3`** and **`b=4`**. It prints **`7`**.
- **`print("Add 3 + default 5:", add(3))`** calls the **`add`** function with only **`3`** as an argument. Here, **`a=3`**, and **`b`** takes the default value **`5`**. It prints **`8`**.

Dictionaries

Dictionaries in Python store data in key-value pairs. They are unordered, mutable, and indexed by keys. Let's look at the following code.

Listing 1-n: Dictionaries (basics.py)

```
person = {
    "name": "Charlie",
    "age": 28,
    "city": "New York"
}

# Accessing elements in a dictionary
print("Person's name:", person["name"])

# Adding elements to a dictionary
person["email"] = "charlie@example.com"
print("Person dictionary after adding email:", person)
```

Explanation:

- **`person = {"name": "Charlie", "age": 28, "city": "New York"}`** creates a dictionary named **`person`** with keys **`name`**, **`age`**, and **`city`** corresponding to the values **`Charlie`**, **`28`**, and **`New York`**, respectively.
- You can access a dictionary value using its key, so **`person["email"]`** retrieves the value associated with the key name, **`Charlie`**.
- You can add new key-value pairs to a dictionary by assigning a value to a new key. **`person["email"] = "charlie@example.com"`** adds the key **`email`** with the value **`"charlie@example.com"`** to the person dictionary.

Tuples

A tuple is an ordered, immutable collection of items. Tuples are similar to lists but immutable, meaning that once you create a tuple, you cannot change its elements. Tuples are often used for data that should remain the same throughout the program, and to return multiple values from a function.

Listing 1-o: Tuples (basics.py)

```
coordinates = (10.0, 20.0)

# Accessing elements in a tuple
print("X coordinate:", coordinates[0])
print("Y coordinate:", coordinates[1])
```

Explanation:

- **`coordinates = (10.0, 20.0)`** creates a tuple named coordinates with two float elements: **`10.0`** and **`20.0`**.
- Like lists, tuples are zero-indexed; thus, **`coordinates[0]`** accesses the first element, which is **`10.0`**, and **`coordinates[1]`** accesses the second element, which is **`20.0`**.

Classes

Classes are blueprints for creating objects (instances). An object is an instance of a class, containing both data (attributes) and functions (methods) that operate on the data.

Listing 1-p: Classes (basics.py)

```
class Dog:
    """A simple class representing a dog."""

    def __init__(self, name, age):
        """Initialize the dog with a name and age."""
        self.name = name
        self.age = age

    def bark(self):
        """Make the dog bark."""
        return f"{self.name} says woof!"

# Creating an instance of the Dog class
my_dog = Dog("Rex", 5)

# Accessing attributes and methods
print("Dog's name:", my_dog.name)
print("Dog's age:", my_dog.age)
print(my_dog.bark())
```

Explanation:

- **`class Dog`** defines a new class named **`Dog`**.

- `"""A simple class representing a dog."""` this docstring describes what the class represents.
- `def __init__(self, name, age)`; the `__init__` method is a special method called a constructor. It is automatically invoked when a new instance of the class is created. It initializes the object's attributes.
- `self.name = name` assigns the value of `name` to the object's `name` attribute.
- `self.age = age` assigns `age` value to the object's `age` attribute.
- `def bark(self)` defines a method named `bark` associated with instances of the `Dog` class.
- `return f"{self.name} says woof!"` returns a string where the dog's `name` is included, followed by `" says woof!"`.
- `my_dog = Dog("Rex", 5)` creates an instance of the `Dog` class with the `name` `"Rex"` and `age` `5`. The `__init__` method is called, setting `my_dog.name` to `"Rex"` and `my_dog.age` to `5`.
- `my_dog.name` accesses the `name` attribute of `my_dog`, which is `"Rex"`.
- `my_dog.age` accesses the `age` attribute of `my_dog`, which is `5`.
- `my_dog.bark()` calls the `bark` method of `my_dog`, which returns `"Rex says woof!"`.

List of objects

You can create lists containing objects, just like any other data type. Let's consider the following code.

Listing 1-q: List of Objects (basics.py)

```
dogs = [Dog("Buddy", 3), Dog("Bella", 2), Dog("Max", 1)]

for dog in dogs:
    print(dog.name, "is", dog.age, "years old.")
```

So, `dogs = [Dog("Buddy", 3), Dog("Bella", 2), Dog("Max", 1)]` creates a list of `Dog` objects. Each `Dog` is instantiated with a `name` and `age`.

On the other hand, `for dog in dogs` iterates over each `Dog` object in the `dogs` list. Then, for each `Dog` object, it prints the dog's `name` and `age`.

The finished basics.py code

Based on all the previous code snippets covered, here is what the finished **basics.py** code file looks like.

Listing 1-r: Recap of Previous Examples (basics.py)

```
# Variables
#
```

```
name = "Alice"  # String variable
anotherName = 'Alice'  # String variable
age = 30  # Integer variable
height = 5.5  # Float variable
is_student = True  # Boolean variable

print("Name:", name)
print("Another Name:", anotherName)
print("Age:", age)
print("Height:", height)
print("Is Student:", is_student)

# Lists
#
# A list is an ordered collection of items.
fruits = ["apple", "banana", "cherry"]

# Accessing elements in a list
print("First fruit:", fruits[0])

# Adding elements to a list
fruits.append("mango")
print("Fruits after adding date:", fruits)

# For loop
for fruit in fruits:
    print("Fruit:", fruit)

# While loop
count = 0
while count < 3:
    print("Count:", count)
    count += 1

# Conditionals
# Conditionals allow you to execute different code based on conditions.

if age < 18:
    print("Minor")
elif age < 65:
    print("Adult")
else:
    print("Senior")
```

```python
# Functions
# A function is a reusable block of code that performs a specific task.

def greet(name):
    """Greet a person by their name."""
    return f"Hello, {name}!"

print(greet("Bob"))

# Function with multiple parameters and default parameter value
def add(a, b=5):
    """Add two numbers."""
    return a + b

print("Add 3 + 4:", add(3, 4))
print("Add 3 + default 5:", add(3))

# Dictionaries
# A dictionary is a collection of key-value pairs.

person = {
    "name": "Charlie",
    "age": 28,
    "city": "New York"
}

# Accessing elements in a dictionary
print("Person's name:", person["name"])

# Adding elements to a dictionary
person["email"] = "charlie@example.com"
print("Person dictionary after adding email:", person)

# Tuples
# A tuple is an ordered, immutable collection of items.

coordinates = (10.0, 20.0)

# Accessing elements in a tuple
print("X coordinate:", coordinates[0])
print("Y coordinate:", coordinates[1])

# Classes
# A class defines a blueprint for creating objects.
```

```
class Dog:
    """A simple class representing a dog."""

    def __init__(self, name, age):
        """Initialize the dog with a name and age."""
        self.name = name
        self.age = age

    def bark(self):
        """Make the dog bark."""
        return f"{self.name} says woof!"

# Creating an instance of the Dog class
my_dog = Dog("Rex", 5)

# Accessing attributes and methods
print("Dog's name:", my_dog.name)
print("Dog's age:", my_dog.age)
print(my_dog.bark())

dogs = [Dog("Buddy", 3), Dog("Bella", 2), Dog("Max", 1)]

for dog in dogs:
    print(dog.name, "is", dog.age, "years old.")
```

Combining dictionaries and classes

Let's look at a comprehensive example that combines dictionaries and classes to manage more complex data.

Listing 1-s: Dictionaries and Classes (students.py)

```
# Comprehensive example combining dictionaries and classes

class Student:
    """A class representing a student."""

    def __init__(self, name, grades):
        """Initialize the student with a name and grades."""
        self.name = name
        self.grades = grades  # Dictionary of subject: grade
```

```
    def average_grade(self):
        """Calculate the average grade of the student."""
        total = sum(self.grades.values())
        count = len(self.grades)
        return total / count if count > 0 else 0

students = [
    Student("Alice", {"math": 90, "science": 85, "english": 88}),
    Student("Bob", {"math": 75, "science": 80, "english": 70}),
    Student("Charlie", {"math": 95, "science": 100, "english": 90}),
]

for student in students:
    print(f"{student.name}'s average grade is {student.average_grade()}")
```

Explanation:

- **`class Student`** defines a new class named **`Student`**.
- **`def __init__(self, name, grades)`** The constructor initializes the **`name`** and **`grades`** attributes.
- **`self.grades = grades`:** The **`grades`** attribute is a dictionary where the keys are subjects (like **`"math"`** or **`"science"`**), and the values are the corresponding grades.
- **Calculating average grade:**
    - **`def average_grade(self)`** defines a method to calculate the average student's grade.
    - **`total = sum(self.grades.values())`** sums all the values (grades) in the grades dictionary.
    - **`count = len(self.grades)`** counts the number of subjects.
    - **`return total / count if count > 0 else 0`** calculates and returns the average grade. If there are no **`grades`**, it returns **`0`** to avoid division by zero.
- **Creating instances and calculating averages:**
    - **`students = []`** creates a list of **`Student`** objects, each initialized with a **`name`** and a dictionary of **`grades`**.
    - **`for student in students`** iterates over each **`Student`** object in the **`students`** list.
    - **`print(f"{student.name}'s average grade is {student.average_grade()}")`** prints the student's name and their average grade, calculated by the average_grade() method.

If we execute this script by running the **`python students.py`** command, we get the following results from the built-in terminal within VS Code.

```
PS C:\PERSONAL-DATA\Projects\Books\Essential Python> python students.py
Alice's average grade is 87.66666666666667
Bob's average grade is 75.0
Charlie's average grade is 95.0
PS C:\PERSONAL-DATA\Projects\Books\Essential Python>
```

Figure 1-i: VS Code (Terminal Open with Printed Results)

Recap

In conclusion, each of these snippets introduces you to the core concepts of Python, offering a solid foundation for programming in the language. Variables help you understand how to store different types of data, while lists allow you to work with ordered collections of items.

Loops enable automation by iterating over collections, and conditionals help you make decisions in your code based on specific conditions. Functions encapsulate reusable logic, making your code more efficient and modular.

Moving further, dictionaries let you manage key-value pairs, providing a flexible way to store and access data. Unlike lists but immutable, tuples offer a way to work with fixed collections. Classes introduce the concept of object-oriented programming by creating blueprints for objects, combining both data and behavior.

By incorporating dictionaries within classes, you can manage more complex data structures and operations. Together, these concepts form a comprehensive base that will support further exploration and practical application of Python in various programming tasks.

Chapter 2 Essential Libraries

Quick intro

Python's extensive ecosystem of libraries makes it a versatile and powerful language for developers across various domains.

From essential data handling libraries like NumPy and Pandas to web development frameworks like Flask and Django, these libraries streamline everything from simple scripting to advanced machine learning.

Mastering essential libraries such as Requests, TensorFlow, and SQLAlchemy empowers developers to efficiently build, analyze, and deploy robust applications in data science to web development.

The recommended way to install Python libraries is to use the pip program, included by default with all Python binary installers starting with version 3.4. If you get an error message when using **pip**, ensure its location is included in your system **PATH** environment variable.

Standard library (built-in)

The Python standard library is an extensive suite of modules that includes Python, providing functionalities for file handling, math, networking, threading, and much more. Some essential modules include:

- **`os`:** Interact with the operating system, e.g., file and directory management.
- **`sys`:** Access system-specific parameters and functions.
- **`math`:** Provides mathematical functions.
- **`datetime`:** Work with dates and times.
- **`json`:** Handle JavaScript Object Notation (JSON) data (serialization and deserialization).
- **`re`:** Perform regular expression operations.

NumPy

- **Purpose:** Numerical computations.
- **Why it's important:** NumPy supports arrays, matrices, and many mathematical functions to operate on these data structures. It's a foundational library in Python for scientific computing.
- **Use case:** Efficient array manipulation, complex mathematical operations, and integration with other data analysis libraries.

Listing 2-a: Install NumPy Command

```
pip install numpy
```

Pandas

- **Purpose:** Data manipulation and analysis.
- **Why it's important:** Pandas introduces two primary data structures, **`Series`** and **`DataFrame`**, making it easier to work with structured data (such as CSV files and SQL databases).
- **Use case:** Data cleaning, transformation, and analysis in tasks ranging from finance to scientific research.

Listing 2-b: Install Pandas Command

```
pip install pandas
```

Matplotlib and Seaborn

- **Purpose:** Data visualization.
- **Why they're important:**
  - **Matplotlib** is a comprehensive library for creating static, animated, and interactive visualizations in Python.
  - **Seaborn** is built on top of Matplotlib, which makes creating more advanced statistical visualizations easier. The name *Seaborn* has no significant meaning; it's a nod to the Samuel Norman Seaborn character in the TV series *The West Wing*.
- **Use case:** Plotting charts, graphs, histograms, and heat maps, crucial for data analysis and reporting.

Listing 2-c: Install Matplotlib Command

```
pip install matplotlib
```

Listing 2-d: Install Seaborn Command

```
pip install seaborn
```

Requests

- **Purpose:** HTTP requests.
- **Why it's essential:** It simplifies making HTTP requests in Python (**`GET`**, **`POST`**, **`PUT`**, etc.), handling headers, form data, and more, making it an essential library for web scraping, RESTful API calls, and communication between web applications.
- **Use case:** Interacting with web APIs or scraping websites.

Listing 2-e: Install Requests Command

```
pip install requests
```

Flask and Django

- **Purpose:** Web development.
- **Why they're important:**
    - **Flask** is a micro-framework that provides flexibility and simplicity for building web applications.
    - **Django** is a high-level framework designed for rapid development, offering a complete toolkit for building robust, scalable applications.
- **Use case:** Developing web applications, RESTful APIs, and microservices.

Listing 2-f: Install Flask Command

```
pip install Flask
```

Listing 2-g: Install Django Command

```
pip install Django
```

SQLAlchemy

- **Purpose:** Database interaction.
- **Why it's important:** SQLAlchemy is an object relational mapper (ORM) that allows developers to interact with databases using Python instead of writing raw SQL queries.
- **Use case:** Database integration in Python applications, supporting multiple database backends like SQLite, PostgreSQL, and MySQL.

Listing 2-h: Install SQLAlchemy Command

```
pip install SQLAlchemy
```

BeautifulSoup

- **Purpose:** Web scraping.
- **Why it's essential:** BeautifulSoup allows developers to parse HTML and XML documents quickly, making it essential for web scraping tasks.
- **Use case:** Extracting data from webpages and scraping website content.

Listing 2-i: Install BeautifulSoup Command

```
pip install beautifulsoup4
```

TensorFlow and PyTorch

- **Purpose:** Machine learning and deep learning.
- **Why they're important:**

    - **TensorFlow** is an open-source machine learning library developed by Google, widely used for neural networks and deep learning.
    - **PyTorch** is an alternative developed by Facebook, known for its flexibility and ease of use, especially in research settings.
- **Use case:** Building machine learning models, deep neural networks, computer vision, and natural language processing.

Listing 2-j: Install TensorFlow Command

```
pip install tensorflow
```

Listing 2-k: Install PyTorch Command

```
pip install torch
```

scikit-learn

- **Purpose:** Machine learning.
- **Why it's essential:** scikit-learn is a general-purpose machine learning library that offers tools for classification, regression, clustering, and model evaluation.
- **Use case:** Building machine learning models like decision trees, SVMs, and linear regression, as well as performing feature extraction and model evaluation.

Listing 2-l: Install scikit-learn Command

```
pip install scikit-learn
```

Jupyter Notebook

- **Purpose:** Interactive computing environment.
- **Why it's essential:** Jupyter Notebook provides an interactive, web-based environment where developers can write and execute Python code. It's beneficial for data analysis, visualization, and exploratory programming.
- **Use case:** Data science, teaching, and quick prototyping of Python code with inline visualizations.

Listing 2-m: Install Jupyter Notebook Command

```
pip install notebook
```

Pillow

- **Purpose:** Image processing.
- **Why it's essential:** Pillow is a Python Imaging Library fork that supports opening, manipulating, and saving image files in various formats.
- **Use case:** Image transformation, resizing, cropping, and filtering in applications like web development and computer vision.

Listing 2-n: Install Pillow Command

```
pip install Pillow
```

Pytest

- **Purpose:** Testing framework.
- **Why it's important:** Pytest is a simple yet powerful testing framework for Python. It supports simple unit testing as well as complex functional testing for applications.
- **Use case:** Writing and running test cases to ensure code quality and reliability.

Listing 2-o: Install Pytest Command

```
pip install pytest
```

OpenCV

- **Purpose:** Computer vision.
- **Why it's essential:** OpenCV provides tools for image and video processing, allowing for the creation of computer vision applications.
- **Use case:** Real-time image processing, facial recognition, object detection, and more.

Listing 2-p: Install OpenCV Command

```
pip install opencv-python
```

asyncio

- **Purpose:** Asynchronous programming. It's built in for Python 3.4 onward.
- **Why it's essential:** asyncio allows Python developers to write concurrent code using the async/await syntax. It is helpful for I/O-bound tasks such as network and file operations.
- **Use case:** Handling asynchronous tasks like multiple HTTP requests or background jobs.
- **Installation:** Installing it is unnecessary; it is part of Python's standard library, included by default from Python 3.4 onward.

Recap

Installing all these libraries by executing the following command is also possible.

Listing 2-q: Installing All Previous Libraries Command (All in the same line)

```
pip install numpy pandas matplotlib seaborn requests Flask Django
SQLAlchemy beautifulsoup4 tensorflow torch scikit-learn notebook Pillow
pytest opencv-python
```

These libraries cover many use cases—from basic scripting and web development to advanced machine learning and data analysis.

Mastering them can significantly enhance your productivity as a Python developer, allowing you to tackle a diverse range of projects and domains efficiently.

Chapter 3 Object-Oriented Programming

Quick intro

Object-oriented programming (OOP) is a programming paradigm that structures code using classes and objects to model real-world entities. OOP emphasizes modularity, encapsulation, inheritance, and polymorphism, making it easier to create organized, reusable, and scalable code.

This chapter will explore these OOP principles using Python with a detailed code example to illustrate each concept.

> ***Note: You can run the code examples below by opening the built-in terminal within VS Code and executing the command:*** *`python <script>.py`*. ***Replace*** *`<script>`* ***with the name of the respective Python file to execute.***

Classes and objects

In OOP, a class is a blueprint for creating objects (instances). A class defines attributes and methods (functions defined within the class) that describe an object's properties and behaviors. An object is an instance of a class containing data (attributes) and behaviors (methods).

Let's create a simple class representing a **`BankAccount`**, modeling real-world properties such as **`account_balance`**, and actions like **`deposit`** and **`withdrawal`**.

Listing 3-a: Creating a BankAccount Class (oop.py)

```
class BankAccount:
    # Constructor method to initialize the BankAccount object
    def __init__(self, account_holder, balance=0.0):
        self.account_holder = account_holder  # Public attribute
        self._balance = balance               # Protected attribute

    # Method to deposit money
    def deposit(self, amount):
        if amount > 0:
            self._balance += amount
            print(f"{amount} deposited successfully.")
        else:
            print("Invalid deposit amount.")

    # Method to withdraw money
```

```
    def withdraw(self, amount):
        if 0 < amount <= self._balance:
            self._balance -= amount
            print(f"{amount} withdrawn successfully.")
        else:
            print("Insufficient funds or invalid amount.")

    # Method to get the current balance
    def get_balance(self):
        return self._balance
```

Explanation:

So, to understand what is going on, let's break the code into parts. Let's begin with the class definition parts.

- **Class definition:** `class BankAccount` defines the class **`BankAccount`**.
- **Constructor (`__init__`):** The constructor (**`__init__`** method) is a particular method that initializes the object's attributes.
- **`account_holder`:** A public attribute representing the account holder's name
- **`_balance`:** A private attribute (indicated by the underscore) representing the account balance. The underscore indicates that the variable is meant for private or internal use, but the variable does have a public scope. There are no private variables in Python classes.

Now, let's look at the methods.

- **`deposit`:** Increases the balance if the deposit amount is valid (greater than zero).
- **`withdraw`:** Decreases the balance if there are sufficient funds.
- **`get_balance`:** Returns the current balance of the account.

Now, let's look at some code to create and use a **`BankAccount`** object.

Listing 3-b: Creating and Using a BankAccount Object (oop.py - Continued)

```
# Create an instance of BankAccount
account = BankAccount("John Doe", 1000.0)

# Deposit money
account.deposit(500.0)

# Withdraw money
account.withdraw(200.0)

# Check balance
print("Current Balance:", account.get_balance())
```

This code snippet creates a **`BankAccount`** object, makes a deposit, withdraws funds, and displays the balance.

Inheritance

Inheritance allows a new class to inherit attributes and methods from an existing class, promoting code reuse. Let's create a new class, **`SavingsAccount`**, which inherits from **`BankAccount`**.

Listing 3-c: Implementing Inheritance (oop.py - Continued)

```
class SavingsAccount(BankAccount):
    # Constructor method for the SavingsAccount class
    def __init__(self, account_holder, balance=0.0, interest_rate=0.02):
        # Initialize the parent BankAccount class
        super().__init__(account_holder, balance)
        self.interest_rate = interest_rate  # attribute for interest rate

    # Method to calculate interest
    def add_interest(self):
        interest = self._balance * self.interest_rate
        self.deposit(interest)
        print(f"Interest of {interest} added to balance.")
```

Explanation:

- **`class SavingsAccount(BankAccount):`** This defines a new class **`SavingsAccount`** that inherits from **`BankAccount`**.
- **Calling the parent constructor:** The **`super()`** function calls the parent (**`BankAccount`**) class's constructor to initialize inherited attributes.
- **New attribute: `interest_rate`** is specific to **`SavingsAccount`** and represents the interest rate.
- **New method (`add_interest`):** Calculates and adds interest to the balance.

Now, let's use the **`SavingsAccount`** class.

Listing 3-d: Using Inheritance (oop.py - Continued)

```
# Create an instance of SavingsAccount
savings = SavingsAccount("Jane Doe", 2000.0, 0.03)

# Add interest
savings.add_interest()

# Check balance after interest is added
```

```
print("Balance with Interest:", savings.get_balance())
```

Polymorphism

Polymorphism enables methods to be used differently based on the object invoking them. This allows flexibility when calling methods, even if they behave differently across various classes.

Listing 3-e: Implementing Polymorphism (oop.py - Continued)

```
class CurrentAccount(BankAccount):
    def withdraw(self, amount):
        if amount > self._balance:
            print("Overdraft: Withdrawal exceeds current balance.")
        else:
            super().withdraw(amount)
```

The **`withdraw`** method in the class **`CurrentAccount`** behaves differently than in **`BankAccount`**. In **`CurrentAccount`**, an overdraft message is displayed if the withdrawal exceeds the balance. Now, let's see how to use polymorphism.

Listing 3-f: Using Polymorphism (oop.py - Continued)

```
# Create instances of different account types
accounts = [BankAccount("John Smith", 1000), SavingsAccount("Jane Doe",
2000, 0.03), CurrentAccount("Paul Johnson", 500)]

# Demonstrate polymorphism
for account in accounts:
    account.withdraw(600)
    # This will call each class's unique withdraw method
```

Encapsulation

Encapsulation restricts access to specific attributes, protecting them from unintended modification.

Encapsulation is achieved in Python by making attributes "private" (using double underscores __). Let's look at an example.

Listing 3-g: Implementing Encapsulation (oop.py - Continued)

```
class EncapsulatedAccount:
```

```
    def __init__(self, balance=0.0):
        self.__balance = balance  # Private attribute

    def deposit(self, amount):
        if amount > 0:
            self.__balance += amount
        else:
            print("Invalid deposit amount.")

    def get_balance(self):
        return self.__balance
        # Accessing private attribute through a method
```

Explanation:

- **Private attribute (`__balance`):** Prefixing balance with double underscores becomes private and inaccessible outside the class.
- **Accessing private attributes:** We use a public method (**`get_balance`**) to access the balance, maintaining controlled access to this data.

Now, let's use encapsulation.

Listing 3-h: Using Encapsulation (oop.py - Continued)

```
# Create an instance of EncapsulatedAccount
encap_account = EncapsulatedAccount(1000)

# Deposit money
encap_account.deposit(300)

# Attempt to access private attribute directly (will raise an error)
# print(encap_account.__balance)
# Uncommenting this line above would cause an AttributeError

# Accessing private attribute through a method
print("Encapsulated Account Balance:", encap_account.get_balance())
```

Now, notice that uncommenting **`print(encap_account.__balance)`** would cause an attribute error. This is because **`__balance`** is private and inaccessible outside the class.

All together

Here's the complete combined code that demonstrates all the core concepts of object-oriented programming (OOP) in Python, including classes and objects, inheritance, polymorphism, and encapsulation.

Listing 3-i: Full OOP Code (oop.py)

```
# Base class for a bank account
class BankAccount:
    # Constructor method to initialize a BankAccount object
    def __init__(self, account_holder, balance=0.0):
        self.account_holder = account_holder  # Public attribute
        self._balance = balance               # Protected attribute

    # Method to deposit money
    def deposit(self, amount):
        if amount > 0:
            self._balance += amount
            print(f"{amount} deposited successfully.")
        else:
            print("Invalid deposit amount.")

    # Method to withdraw money
    def withdraw(self, amount):
        if 0 < amount <= self._balance:
            self._balance -= amount
            print(f"{amount} withdrawn successfully.")
        else:
            print("Insufficient funds or invalid amount.")

    # Method to get the current balance
    def get_balance(self):
        return self._balance


# Derived class for a savings account with interest
class SavingsAccount(BankAccount):
    # Constructor for SavingsAccount, including an interest rate
    def __init__(self, account_holder, balance=0.0, interest_rate=0.02):
        super().__init__(account_holder, balance)
        # Initialize parent class attributes
        self.interest_rate = interest_rate
        # Additional attribute for interest rate
```

```
    # Method to add interest to the account balance
    def add_interest(self):
        interest = self._balance * self.interest_rate
        self.deposit(interest)
        # Add interest to the balance using deposit method
        print(f"Interest of {interest} added to balance.")

# Derived class for a current account with an overdraft notification
class CurrentAccount(BankAccount):
    # Overriding the withdraw method for overdraft notification
    def withdraw(self, amount):
        if amount > self._balance:
            print("Overdraft: Withdrawal exceeds current balance.")
        else:
            super().withdraw(amount)
            # Call the parent class's withdraw method

# Encapsulated account class to demonstrate encapsulation
class EncapsulatedAccount:
    def __init__(self, balance=0.0):
        self.__balance = balance
        # Private attribute to restrict direct access

    # Method to deposit money into the account
    def deposit(self, amount):
        if amount > 0:
            self.__balance += amount
        else:
            print("Invalid deposit amount.")

    # Method to access the private balance attribute safely
    def get_balance(self):
        return self.__balance

# Testing the combined functionality of all classes

# Create a general bank account
account = BankAccount("John Doe", 1000.0)
account.deposit(500)
account.withdraw(300)
print("Bank Account Balance:", account.get_balance())
print("-" * 40)

# Create a savings account and add interest
```

```
savings = SavingsAccount("Jane Doe", 2000.0, 0.03)
savings.add_interest()
print("Savings Account Balance after interest:", savings.get_balance())
print("-" * 40)

# Create a current account and attempt an overdraft
current = CurrentAccount("Paul Johnson", 500)
current.withdraw(600)  # Should show an overdraft warning
current.withdraw(400)  # Valid withdrawal
print("Current Account Balance:", current.get_balance())
print("-" * 40)

# Demonstrate encapsulation with a protected account
encap_account = EncapsulatedAccount(1000)
encap_account.deposit(300)
# Attempting to access private attribute directly would raise an error
# print(encap_account.__balance)
# Uncommenting this line would cause an AttributeError

print("Encapsulated Account Balance:", encap_account.get_balance())
```

Recap

To wrap up this chapter, let's look at what we learned:

- **Classes and objects:** Classes define templates, while objects are instances of these templates.
- **Inheritance:** A class can inherit properties from another, promoting code reuse.
- **Polymorphism:** Allows objects to use methods differently, enabling flexibility.
- **Encapsulation:** Restricts access to specific data, protecting the integrity of an object's attributes.

These principles lay the foundation for building scalable and modular applications, transforming Python into a powerful tool for creating complex software.

Chapter 4 Error Handling and Debugging

Quick intro

Error handling and debugging are essential parts of writing robust and reliable software. By anticipating and managing errors effectively, you can ensure that your code behaves predictably, even when unexpected events occur.

Python offers powerful error-handling tools, such as **`try-except`** blocks, and comprehensive debugging support through libraries like logging and tools like **`pdb`**.

In this chapter, we'll dive into these techniques with examples to illustrate how they work.

> ***Note: You can run the following code examples by opening the built-in terminal within VS Code and executing the command:*** *`python <script>.py`*. ***Replace*** *`<script>`* ***with the name of the respective Python file to execute.***

Try-except blocks

A **`try-except`** block allows you to handle exceptions (errors) that might occur in your code. By surrounding potentially problematic code with **`try`**, you can "catch" exceptions in the **`except`** block and handle them gracefully without stopping the program.

Here's a basic example of attempting to divide two numbers and handling potential exceptions, such as division by zero or invalid input.

Listing 4-a: Try-Except (try.py)

```
def divide_numbers(a, b):
    try:
        # Attempt to divide a by b
        result = a / b
        print(f"The result is {result}")
    except ZeroDivisionError:
        # Handle division by zero error
        print("Error: Cannot divide by zero.")
    except TypeError:
        # Handle incorrect data types error
        print("Error: Both inputs must be numbers.")
    else:
        # This block runs if no exceptions were raised
        print("Division successful.")
```

```
    finally:
        # This block always runs, even if an exception occurred
        print("End of division operation.")

# Testing divide_numbers function
divide_numbers(10, 2)   # Normal division
divide_numbers(10, 0)   # Division by zero
divide_numbers(10, 'a') # Type error
```

Explanation:

- **`try` block:** `try` initiates a block where an exception might occur. Here, **`result = a / b`** could raise exceptions if **`b`** is zero or if **`a`** or **`b`** are not numbers.
- We have two **`except`** blocks here:
    - **`except ZeroDivisionError`** catches an attempt to divide by zero, printing an error message.
    - **`except TypeError`** catches an error if a non-numeric value is used, printing a different error message.
- **`else`:** Runs if no exceptions occur, confirming that the division was successful.
- **`finally`:** Executes regardless of whether an exception occurred, allowing cleanup or final actions (e.g., closing resources, such as closing open file handles).

Logging

The **`logging`** module in Python provides a powerful way to log messages in your code. Logging is more flexible and systematic than **`print`** statements, especially for larger applications, as it supports different logging levels, formatted messages, and output to files.

Let's use logging to record information and error messages when performing division. Instead of just printing, this lets us keep a record of important events and errors.

Listing 4-b: Logging (log.py)

```
import logging

# Set up basic logging configuration
logging.basicConfig(
    level=logging.INFO,      # Minimum level to log
    format='%(asctime)s - %(levelname)s - %(message)s',  # Log format
    filename='app.log',      # Log output file
    filemode='w'             # Write mode (overwrites each run)
)

def divide_numbers(a, b):
```

```
    try:
        result = a / b
        logging.info(f"Division successful: {a} / {b} = {result}")
        # Info log for successful division
        return result
    except ZeroDivisionError as e:
        logging.error("Attempted to divide by zero.")
        # Error log for division by zero
    except TypeError as e:
        logging.error("Invalid types provided for division.")
        # Error log for type error
    except Exception as e:
        logging.exception("An unexpected error occurred.")
        # Logs exception with traceback

# Testing the logging function
divide_numbers(10, 2)    # Successful division
divide_numbers(10, 0)    # Division by zero
divide_numbers(10, 'a')  # Type error
```

Explanation:

- **Basic logging configuration:**
  - **`level=logging.INFO`** sets the logging level to **`INFO`**, capturing all events at **`INFO`** level or higher (e.g., **`ERROR`**).
  - **`format`** defines the format for each log entry, including timestamp, level, and **`message`**.
  - **`filename='app.log'`** saves logs to **`app.log`**, a file created in write mode (**`w`**) each time the program runs.
- **Info logs:**
  - **`logging.info()`** is used to log a successful division.
- **Error logs:**
  - **`logging.error()`** logs specific error messages for **`ZeroDivisionError`** and **`TypeError`**.
  - **`logging.exception()`** captures any unexpected error with full traceback, aiding debugging.
- **File output:**
  - Logs are written to **`app.log`**, where you can view a history of all events and errors.

If you execute the **log.py** script from the built-in terminal within VS Code using the **`python log.py`** command, you'll see an **app.log** file created with content similar to the following.

Listing 4-c: app.log

```
2024-10-29 16:21:37,541 - INFO - Division successful: 10 / 2 = 5.0
```

```
2024-10-29 16:21:37,542 - ERROR - Attempted to divide by zero.
2024-10-29 16:21:37,542 - ERROR - Invalid types provided for division.
```

Debugging with pdb

Python's built-in debugger **pdb** allows you to step through code, inspect variables, and better understand how your program is running. This is particularly useful when tracking down hard-to-find bugs.

Let's use **pdb** to inspect the division process step-by-step and see variable values at each point.

Listing 4-d: debug.py

```
import pdb  # Import the pdb module

def divide_numbers(a, b):
    pdb.set_trace()  # Set a breakpoint
    result = a / b
    return result

# Run the function
divide_numbers(10, 2)  # Try with valid input to see how pdb works
```

pdb.set_trace() sets a breakpoint in the code, pausing execution so you can inspect values and step through the code.

Run the code from the built-in terminal in VS Code by executing the **python debug.py**, and when it pauses, you can use the following common **pdb** commands:

- **n** (next): Move to the following line.
- **s** (step): Step into a function call to inspect it.
- **p** <*variable*>: Print the value of a variable.
- **c** (continue): Continue execution until the next breakpoint or the end of the program.

Running **divide_numbers(10, 2)** will pause at **pdb.set_trace()**, allowing you to inspect variables **a**, **b**, and **result**.

Full example

Combining **try-except**, **logging**, and **pdb** gives you a solid foundation for handling errors and debugging effectively. Let's look at the following example.

Listing 4-e: fulldebug.py

```
import logging
import pdb

# Set up logging
logging.basicConfig(level=logging.INFO, format='%(asctime)s - %(levelname)s
- %(message)s', filename='app.log')

def divide_numbers(a, b):
    pdb.set_trace()  # Set breakpoint for debugging

    try:
        result = a / b
        logging.info(f"Division successful: {a} / {b} = {result}")
        return result
    except ZeroDivisionError:
        logging.error("Attempted to divide by zero.")
        print("Error: Cannot divide by zero.")
    except TypeError:
        logging.error("Invalid types provided for division.")
        print("Error: Please enter numeric values.")
    except Exception as e:
        logging.exception("An unexpected error occurred.")
        print("An unexpected error occurred.")

# Testing different scenarios
divide_numbers(10, 2)    # Successful division
divide_numbers(10, 0)    # Division by zero
divide_numbers(10, 'a')  # Type error
```

Explanation:

- **Setting a breakpoint:**
    - The **`pdb.set_trace()`** in **`divide_numbers`** lets you observe the function's execution and variable values.
- **Logging setup:**
    - **`logging.basicConfig`** records successful operations and errors, storing details in **`app.log`**.
- **Try-except block:**
    - Handles known exceptions (**`ZeroDivisionError`**, **`TypeError`**) with specific error messages and logs.
    - Uses **`logging.exception()`** for any unexpected error, which logs the complete traceback.
- **Testing different cases:**
    - Calls **`divide_numbers`** with different inputs to verify that errors are handled appropriately.

Recap

Using **`try-except`** blocks, **`logging`**, and **`pdb`**, you can create robust Python applications that handle unexpected errors gracefully, keep a detailed error log, and offer a structured approach to debugging.

These tools help keep code maintainable, user-friendly, and easier to troubleshoot.

Chapter 5 File Handling

Quick intro

File handling in Python is a fundamental skill for many applications, allowing you to read, write, and manipulate files. This section will cover the basics of reading and writing different file formats, like text, CSV, and JSON, and more advanced file operations using the OS and shutil modules.

> ***Note: You can run the code following examples by opening the built-in terminal within VS Code and executing the command:*** *`python <script>.py`*. ***Replace*** *`<script>`* ***with the name of the respective Python file to execute.***

Reading and writing text files

Python provides built-in functions to open, read, and write files, especially with text-based data. Let's start by understanding how to handle essential text files, and then we'll dive into CSV and JSON files.

To open a text file, you can use the **`open()`** function, which accepts the file path and a mode (**`'r'`** for read, **`'w'`** for write, **`'a'`** for append).

Listing 5-a: text.py

```
# Writing to a text file
with open("example.txt", "w") as file:
    file.write("Hello, this is a sample text file.\n")
    file.write("File handling is essential in Python.\n")

# Reading from a text file
with open("example.txt", "r") as file:
    content = file.read()  # Read the entire file content
    print(content)
```

We open **`example.txt`** in **`"w"`** (write) mode. The **`with`** statement ensures that the file is correctly closed after completing operations. Then **`file.write()`** writes strings to the file. Each **`write`** call adds text as a new line in the file.

The file is opened in **`"r"`** (read) mode, and **`file.read()`** reads the entire file content. This content is then printed to the console.

Furthermore, the Python **`with`** statement will automatically close open files when the block finishes execution. Although it is possible to open and close files explicitly, this technique is not recommended.

Reading and writing CSV files

The CSV module in Python makes it easy to work with CSV files, which are commonly used for data storage and exchange.

Listing 5-b: csvfiles.py

```
import csv

# Writing to a CSV file
with open("data.csv", "w", newline="") as file:
    writer = csv.writer(file)
    writer.writerow(["Name", "Age", "Country"])
    writer.writerow(["Alice", 30, "USA"])
    writer.writerow(["Bob", 25, "UK"])

# Reading from a CSV file
with open("data.csv", "r") as file:
    reader = csv.reader(file)
    for row in reader:
        print(row)
```

Explanation:

- **Writing to a CSV file:**
  - We use **`csv.writer()`** to create a **`writer`** object to write rows to the CSV file.
  - **`writer.writerow()`** writes each row to the CSV, where each element in the list becomes a cell in the row.
- **Reading from a CSV file:**
  - **`csv.reader()`** creates a **`reader`** object, allowing us to loop through each row.
  - Each row read is a list of strings, making processing data line by line easy.

Reading and writing JSON files

The **`json`** module is invaluable for working with JSON, a format widely used in web APIs and data storage. Let's look at the following code.

Listing 5-c: jsonfiles.py

```
import json
```

```
# Writing to a JSON file
data = {
    "name": "Alice",
    "age": 30,
    "is_employee": True,
    "skills": ["Python", "Data Analysis", "Machine Learning"]
}
with open("data.json", "w") as file:
    json.dump(data, file, indent=4)

# Reading from a JSON file
with open("data.json", "r") as file:
    data_loaded = json.load(file)
    print(data_loaded)
```

Explanation:

- **Writing JSON data:**
  - **`json.dump()`** writes a Python dictionary (or list) to a JSON file.
  - The **`indent=4`** argument makes the JSON readable by formatting it with four spaces per indentation level.
- **Reading JSON data:**
  - **`json.load()`** reads JSON data from a file and converts it to a Python dictionary or list, making it easy to work with structured data.

File ops

The **`os`** and **`shutil`** modules provide powerful tools for file and directory manipulation. You can create, move, copy, and delete files and directories, making managing your file system programmatically more accessible.

Let's look at an example demonstrating basic operations such as creating directories, moving files, and deleting them.

Listing 5-d: fileops.py

```
import os
import shutil

# Create a new directory
os.makedirs("test_dir/sub_dir", exist_ok=True)

# Create a new file in the directory
```

```
with open("test_dir/sample.txt", "w") as file:
    file.write("This is a sample file.")

# Move the file to a new location
shutil.move("test_dir/sample.txt", "test_dir/sub_dir/sample.txt")

# Copy the file to a new location
shutil.copy("test_dir/sub_dir/sample.txt", "test_dir/sample_copy.txt")

# Delete the file and directory
os.remove("test_dir/sample_copy.txt")
shutil.rmtree("test_dir")  # Deletes test_dir and all its contents
```

Explanation:

- **Creating directories:**
  - **`os.makedirs()`** creates directories, including any necessary parent directories. The **`exist_ok=True`** argument ensures that no error occurs if the directory already exists.
- **Moving files:**
  - **`shutil.move()`** moves a file from one location to another.
- **Copying files:**
  - **`shutil.copy()`** creates a copy of the file at the specified location.
- **Deleting files and directories:**
  - **`os.remove()`** deletes a single file.
  - **`shutil.rmtree()`** removes an entire directory tree, deleting the specified directory and all its contents.

Recap

You can manage and process data stored in various file formats by mastering file handling in Python.

Whether working with text, CSV, or JSON files or performing file operations with the **`os`** and **`shutil`** modules, these techniques are invaluable for developing applications that interact with the file system.

Chapter 6 Async Programming

Quick intro

Asynchronous programming is a powerful technique for handling tasks that may take time to complete, such as network requests, file I/O, and other I/O-bound operations.

By running these tasks concurrently, you can significantly improve the efficiency of your code and reduce wait times.

Python provides a few tools for asynchronous programming, including asyncio, threading, and multiprocessing, each with their strengths.

This chapter will cover the following:

- **Concurrency with `asyncio`:** Using **`async`**, **`await`**, and the **`asyncio`** module to handle I/O-bound tasks.
- **Threading and multiprocessing:** Using threading for lightweight concurrency and multiprocessing for CPU-bound tasks.

Note: You can run the following code examples by opening the built-in terminal within VS Code and executing the command: *`python <script>.py`*. ***Replace*** *`<script>`* ***with the name of the respective Python file to execute.***

Concurrency with asyncio

The **`asyncio`** library in Python is designed to handle I/O-bound tasks by enabling you to write asynchronous code using **`async`**, **`await`**, and **`coroutines`**.

Let's look at an asynchronous function with **`asyncio`**. We'll create two tasks that simulate long-running operations, such as fetching data from a server. Using **`async`** and **`await`**, we can run these tasks concurrently.

Listing 6-a: async.py

```
import asyncio

async def fetch_data(delay, name):
    print(f"Starting task {name}...")
    await asyncio.sleep(delay)  # Simulates a long-running I/O-bound task
    print(f"Finished task {name} after {delay} seconds.")
    return f"Data from {name}"
```

```
async def main():
    # Schedule both tasks to run concurrently
    task1 = asyncio.create_task(fetch_data(2, "Task 1"))
    task2 = asyncio.create_task(fetch_data(3, "Task 2"))

    print("Waiting for tasks to complete...")
    results = await asyncio.gather(task1, task2)
    print("All tasks completed:", results)

# Run the asyncio event loop
asyncio.run(main())
```

Explanation:

- **Async function (`fetch_data`):**
  - Defined using **`async def`**, making it a coroutine that can be paused.
  - Inside, **`await asyncio.sleep(delay)`** simulates an I/O-bound delay, where the coroutine "sleeps" without blocking other tasks.
  - The function completes after the specified delay and returns a result.
- **Main function (`main`):**
  - **`asyncio.create_task()`** schedules **`fetch_data`** tasks to run concurrently.
  - **`await asyncio.gather(task1, task2)`** waits for both tasks to complete, gathering their results.
- **Running the event loop:**
  - **`asyncio.run(main())`** starts the primary function and manages the **`async`** tasks, ensuring the code runs to completion.

Threading and multiprocessing

Asynchronous programming is less effective for heavy computation (CPU-bound tasks) due to Python's Global Interpreter Lock (GIL). In such cases, you can use threading or multiprocessing.

Let's look at an example of concurrent execution using threading. The **`threading`** module allows for simultaneous execution and is suitable for I/O-bound tasks where the GIL isn't a bottleneck.

Listing 6-b: multi.py

```
import threading
import time

def perform_task(name, delay):
    print(f"Starting task {name}...")
    time.sleep(delay)  # Simulates a blocking operation
    print(f"Finished task {name} after {delay} seconds.")
```

```
# Creating threads
thread1 = threading.Thread(target=perform_task, args=("Thread 1", 2))
thread2 = threading.Thread(target=perform_task, args=("Thread 2", 3))

# Starting threads
thread1.start()
thread2.start()

# Waiting for threads to complete
thread1.join()
thread2.join()

print("Both threads have completed.")
```

Explanation:

- **Defining the task:**
  - **`perform_task`** is a function that simulates a blocking operation using **`time.sleep`**.
- **Creating threads:**
  - **`threading.Thread()`** initializes a new thread for each task, with **`target=perform_task`** and arguments for name and delay.
- **Starting and joining threads:**
  - **`start()`** initiates each thread, allowing them to run concurrently.
  - **`join()`** waits for both threads to finish before continuing with the main program.

Using multiprocessing for CPU-bound tasks

The **`multiprocessing`** module creates separate processes, bypassing the GIL and allowing true parallelism, which is ideal for CPU-bound tasks.

Listing 6-c: processing.py

```
from multiprocessing import Process
import time

def compute_square(numbers):
    print(f"Calculating squares...")
    for n in numbers:
        time.sleep(1)  # Simulates a heavy computation delay
        print(f"Square of {n}: {n * n}")

if __name__ == "__main__":
    numbers = [1, 2, 3, 4, 5]
```

```
    # Creating a separate process for compute_square
    process = Process(target=compute_square, args=(numbers,))

    process.start()  # Start the process
    process.join()   # Wait for the process to complete

    print("All computations are completed.")
```

Explanation:

- **Defining a CPU-intensive task:**
  - **`compute_square`** operates for each number, simulating CPU-intensive work with **`time.sleep`**.
- **Creating and running a process:**
  - **`Process(target=compute_square, args=(numbers,))`** creates a new process with the function **`compute_square`** and the **`numbers`** list as arguments.
  - **`process.start()`** begins the process, allowing it to run independently.
  - **`process.join()`** blocks the main program until the process completes.

Recap

Using **`asyncio`**, **`threading`**, and **`multiprocessing`** provides flexible and efficient ways to handle both I/O-bound and CPU-bound tasks in Python.

Mastering these tools allows you to create scalable applications that take full advantage of concurrency and parallelism.

Chapter 7 Regular Expressions

Quick intro

Regular expressions (regex) are a powerful tool in Python for working with strings. They allow you to define patterns for matching specific character combinations within strings, making regex invaluable for tasks like validation, parsing, and text manipulation. Python's built-in **`re`** module provides methods to use regular expressions for these purposes.

Let's explore regex basics with examples demonstrating pattern matching, validation, and text manipulation.

> ***Note: You can run the following code examples by opening the built-in terminal within VS Code and executing the command:*** *python <script>.py*. ***Replace*** `<script>` ***with the name of the respective Python file to execute.***

Basic pattern matching

The **`re`** module in Python offers several functions for working with regular expressions, including **`re.search()`**, **`re.match()`**, **`re.findall()`**, and **`re.sub()`**.

We'll review these and show how to apply them to real-world examples.

Simple pattern matching with re.search()

Suppose you want to find if an email address is contained within a string. The **`re.search()`** method is suitable when locating a match within a larger string.

Listing 7-a: regex1.py

```
import re

# Sample text
text = "Contact us at support@example.com for more information."

# Define a pattern for an email
email_pattern = r"\b[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Z|a-z]{2,7}\b"

# Search for the pattern in the text
match = re.search(email_pattern, text)
```

```
if match:
    print("Found email:", match.group())
else:
    print("No email found.")
```

Explanation:

- **Defining the pattern:**
  - `r"\b[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Z|a-z]{2,7}\b"` represents a regular expression pattern that matches a typical email format.
  - `\b` marks a word boundary, ensuring that we don't match email addresses that are part of more significant words.
  - `[A-Za-z0-9._%+-]+` matches the username part of an email, allowing alphanumeric characters and familiar special characters.
  - `[A-Za-z0-9.-]+` matches the domain name, while `\.[A-Z|a-z]{2,7}` matches the domain extension.
- **Using `re.search()`:**
  - `re.search(email_pattern, text)` searches the text for the email pattern.
  - If a match is found, `match.group()` retrieves the matched substring.

Extracting all matches with re.findall()

To extract multiple matches within a string, `re.findall()` is helpful. For instance, finding all phone numbers in a document. Let's have a look at the code.

Listing 7-b: regex2.py

```
import re

text = "Call us at +1-800-555-1234 or +1-800-555-5678."

# Define a pattern for phone numbers
phone_pattern = r"\+1-\d{3}-\d{3}-\d{4}"

# Find all phone numbers in the text
matches = re.findall(phone_pattern, text)

print("Found phone numbers:", matches)
```

Explanation:

- **Phone number pattern:**
  - `r"\+1-\d{3}-\d{3}-\d{4}"` matches US phone numbers in the format `+1-800-555-1234`.
  - `\+1` matches the country code.
  - `\d{3}` matches three digits, and `-` matches the hyphen separators.

- **Using `re.findall()`:**
  - **`re.findall(phone_pattern, text)`** searches for all instances of the pattern in text.
  - It returns a list of all **`matches`**, making it easy to retrieve multiple occurrences.

Validating input with re.match()

Let's say we want to validate that a string is in the correct date format **YYYY-MM-DD**. For this, **`re.match()`** is helpful since it checks if the string starts with the specified pattern.

Listing 7-c: regex3.py

```
import re

# Define a date pattern
date_pattern = r"^\d{4}-\d{2}-\d{2}$"

# Sample date inputs
dates = ["2024-10-29", "29-10-2024", "2024/10/29"]

for date in dates:
    if re.match(date_pattern, date):
        print(f"{date} is a valid date format.")
    else:
        print(f"{date} is an invalid date format.")
```

Explanation:

- **Date pattern:**
  - **`r"^\d{4}-\d{2}-\d{2}$"`** matches dates in the format **YYYY-MM-DD**.
  - **`^`** and **`$`** ensure that the entire string matches the pattern.
  - **`\d{4}`** matches a four-digit year, and **`\d{2}-\d{2}`** matches two-digit month and day parts.
- **Using `re.match()`:**
  - **`re.match(date_pattern, date)`** checks if each date string matches the pattern.
  - This approach is helpful for validation, confirming that the entire string conforms to the format.

Text substitution with re.sub()

You can use **`re.sub()`** to replace string parts that match a pattern, such as sensitive information like email addresses, with a placeholder. Let's have a look at the code.

Listing 7-d: regex4.py

```
import re

text = "Contact us at support@example.com or sales@example.com."

# Pattern for emails
email_pattern = r"\b[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Z|a-z]{2,7}\b"

# Substitute emails with a placeholder
masked_text = re.sub(email_pattern, "[hidden email]", text)

print("Text with emails masked:", masked_text)
```

Explanation:

- **Defining the pattern:**
  - We use the same email pattern as before to locate email addresses in the text.
- **Using `re.sub()`:**
  - **`re.sub(email_pattern, "[hidden email]", text)`** replaces all email matches in text with the string **`[hidden email]`**.
  - This is useful for masking sensitive information in text outputs.

Recap

Regular expressions in Python provide a versatile and powerful way to work with strings. You can handle complex string-processing tasks efficiently by mastering pattern matching, validation, and substitution with the **`re`** module.

Chapter 8 Data Structures

Quick intro

In Python, efficient data handling is essential for performance and scalability, especially when working with large datasets or complex data processing tasks.

This is where Python's advanced data structures come into play, particularly those available in the **`collections`** module, such as **`deque`**, **`defaultdict`**, and **`namedtuple`**.

Structures like sets and heaps are also invaluable for fast lookups and sorting data. Understanding and using these data structures will help you write optimized, maintainable code.

> ***Note: You can run the following code examples by opening the built-in terminal within VS Code and executing the command:*** *`python <script>.py`*. ***Replace*** *`<script>`* ***with the name of the respective Python file to execute.***

Advanced collections

The **`collections`** module in Python offers several valuable data structures for managing data efficiently. Here's a look at three important ones: **`deque`**, **`defaultdict`**, and **`namedtuple`**.

Fast insertions and deletions

A double-ended queue, also known as a **`deque`**, is a data structure optimized for fast insertions and deletions from both ends.

Unlike lists optimized for random access, a **`deque`** is ideal when you frequently need to add or remove elements at the beginning or end. Let's have a look at an example.

Listing 8-a: deque.py

```
from collections import deque

# Initialize a deque with some elements
dq = deque([1, 2, 3, 4, 5])

# Append elements to the right and left ends
dq.append(6)
dq.appendleft(0)
```

```
# Pop elements from the right and left ends
dq.pop()
dq.popleft()

print("Final deque state:", dq)
```

Explanation:

- **Initialize a `deque`:** **`deque([1, 2, 3, 4, 5])`** creates a deque with initial elements.
- **Appending elements:** **`dq.append(6)`** adds `6` to the right end, while **`dq.appendleft(0)`** adds `0` to the left.
- **Popping elements:** **`dq.pop()`** removes the rightmost element, and **`dq.popleft()`** removes the leftmost element.
- **Efficiency:** Unlike lists, these operations are optimized in **`deque`**, making it a good choice when you need fast queue or stack operations.

Simplified dictionary management

A **`defaultdict`** is like a regular dictionary but provides a default value for a nonexistent key, which can prevent common errors and reduce code complexity. Let's look at the code.

Listing 8-b: defaultdict.py

```
from collections import defaultdict

# Initialize a defaultdict with default type of int (for counting)
word_count = defaultdict(int)

# Count occurrences of each word in a list
words = ["apple", "banana", "apple", "orange", "banana", "apple"]
for word in words:
    word_count[word] += 1

print("Word counts:", word_count)
```

Explanation:

- **Initialize a `defaultdict`:** **`defaultdict(int)`** creates a dictionary where each missing key automatically has a default value of `0` (since **`int()`** returns `0`).
- **Counting elements:** Each time a word appears in the list words, **`word_count[word] += 1`** increments its count.
- **Efficiency:** Using **`defaultdict`** prevents checking if a key exists, making code cleaner and less error-prone.

Lightweight object-like data structures

In Python, a **`namedtuple`** allows you to create a lightweight, immutable object type with named fields, providing a clean and memory-efficient alternative to classes when you need simple data storage. Let's look at some code.

Listing 8-c: namedtuple.py

```
from collections import namedtuple

# Define a namedtuple type for a point in 2D space
Point = namedtuple("Point", ["x", "y"])

# Create instances of Point
p1 = Point(3, 4)
p2 = Point(5, 9)

# Access the fields
print("Point 1:", p1)
print("Point 1 x coordinate:", p1.x)
print("Point 2 y coordinate:", p2.y)
```

Explanation:

- **Defining a `namedtuple` type:** **`Point = namedtuple("Point", ["x", "y"])`** defines a named tuple type called **`Point`** with fields **`x`** and **`y`**.
- **Creating instances:** **`Point(3, 4)`** and **`Point(5, 9)`** create immutable **`Point`** objects, which you can access by attribute name.
- **Efficiency:** Named tuples are more memory-efficient than dictionaries and classes, making them ideal for simple, structured data storage.

Sets for fast membership testing

Beyond the **`collections`** module, other data structures like sets and heaps provide efficient ways to handle extensive data.

Sets store unique elements and allow fast membership testing, making them useful for tasks like deduplication and filtering. Let's have a look.

Listing 8-d: sets.py

```
# Define two sets of numbers
set_a = {1, 2, 3, 4, 5}
set_b = {4, 5, 6, 7, 8}
```

```
# Perform set operations
union = set_a | set_b  # Union of sets
intersection = set_a & set_b  # Intersection of sets
difference = set_a - set_b  # Difference of sets

print("Union:", union)
print("Intersection:", intersection)
print("Difference:", difference)
```

Explanation:

- **Creating sets:** `{1, 2, 3, 4, 5}` creates a set with unique elements.
- **Set operations:** `set_a | set_b` finds the union, `set_a & set_b` finds the intersection, and `set_a - set_b` finds elements in `set_a` not in `set_b`.
- **Efficiency:** Sets provide constant time complexity for membership tests (in), making them efficient for filtering and deduplication.

Heaps for efficient data sorting

A heap is a binary tree-based data structure that maintains a sorted list where the smallest or largest element can be accessed immediately. Python's **heapq** module provides tools for working with heaps. Let's look at the following example.

Listing 8-e: heaps.py

```
import heapq

# Create a list of unsorted numbers
numbers = [20, 5, 15, 10, 30]

# Convert the list to a heap
heapq.heapify(numbers)
print("Heapified list:", numbers)

# Add a new number to the heap
heapq.heappush(numbers, 7)
print("Heap after adding 7:", numbers)

# Pop the smallest element
smallest = heapq.heappop(numbers)
print("Smallest element:", smallest)
print("Heap after popping smallest element:", numbers)
```

Explanation:

- **Heapify:** `heapq.heapify(numbers)` converts the list numbers into a heap, arranging elements so that the smallest element is at the root.
- **Adding to the heap:** `heapq.heappush(numbers, 7)` adds seven while maintaining heap order, ensuring the smallest element is immediately accessible.
- **Removing the smallest element:** `heapq.heappop(numbers)` removes the smallest element, reordering the heap to keep it in sorted order.

Recap

Understanding these data structures can significantly improve your code's efficiency. You can handle complex data organization effectively with **`deque`**, **`defaultdict`**, and **`namedtuple`**.

Sets allow fast filtering, while heaps provide efficient ways to sort and access data. Mastering these structures empowers you to handle more advanced and large-scale data tasks in Python.

Chapter 9 Working with APIs

Quick intro

Application programming interfaces (APIs) are essential for integrating external data sources and services into your applications.

They allow you to request external servers and receive responses, typically in the form of data you can manipulate and use within your program.

Python provides several libraries to work with APIs, with requests being one of the most popular for making HTTP requests. Additionally, parsing the responses, often in JSON or XML formats, is crucial for utilizing the data effectively.

> ***Note: You can run the following code examples by opening the built-in terminal within VS Code and executing the command:*** *`python <script>.py`*. ***Replace*** *`<script>`* ***with the name of the respective Python file to execute.***

Making a GET request to a REST API

The **`requests`** library simplifies making HTTP requests in Python. It supports various methods, such as **`GET`**, **`POST`**, **`PUT`**, and **`DELETE`**, enabling you to interact with RESTful APIs effortlessly.

We'll start by making a **`GET`** request to a public API that provides data about Pokémon.

Listing 9-a: getrequest.py

```
import requests

# Define the URL of the API endpoint
url = "https://pokeapi.co/api/v2/pokemon/ditto"

# Make the GET request
response = requests.get(url)

# Check if the request was successful
if response.status_code == 200:
    # Parse the JSON response
    pokemon_data = response.json()
    print("Name:", pokemon_data['name'])
    print("Base experience:", pokemon_data['base_experience'])
```

```
    print("Abilities:", [ability['ability']['name'] for ability in
pokemon_data['abilities']])
else:
    print("Failed to retrieve data:", response.status_code)
```

Explanation:

- **Importing requests:**
  - `import requests` imports the `requests` library to make HTTP requests.
- **Defining the URL:**
  - `url = "https://pokeapi.co/api/v2/pokemon/ditto"` sets the API endpoint URL for retrieving data about the Pokémon Ditto.
- **Making the request:**
  - `response = requests.get(url)` sends a `GET` request to the specified URL and stores the response.
- **Checking the response:**
  - `if response.status_code == 200` checks whether the request succeeded (status code `200`).
- **Parsing JSON data:**
  - `pokemon_data = response.json()` parses the JSON response into a Python dictionary.
  - `print("Name:", pokemon_data['name'])` prints the name of the Pokémon.
  - `print("Base experience:", pokemon_data['base_experience'])` prints the base experience value.
  - `print("Abilities:", [ability['ability']['name']...` prints a list of the Pokémon's abilities.

Parsing and using JSON data

APIs commonly return data in JSON format, which is lightweight and easy to parse in Python using the built-in `json` module or the requests library's built-in methods.

Here's how to parse and manipulate JSON data from an API response.

Listing 9-b: parsejson.py

```
import requests

# Define the URL of the API endpoint
url = "https://jsonplaceholder.typicode.com/posts/1"

# Make the GET request
response = requests.get(url)

# Check if the request was successful
if response.status_code == 200:
```

```
    # Parse the JSON response
    post_data = response.json()
    print("Post Title:", post_data['title'])
    print("Post Body:", post_data['body'])
else:
    print("Failed to retrieve data:", response.status_code)
```

Explanation:

- **Making the request:**
  - Similar to the previous code listing example, a **`GET`** request is made to the URL **`https://jsonplaceholder.typicode.com/posts/1`**, which returns a JSON object representing a post.
- **Parsing the response:**
  - **`post_data = response.json()`** converts the JSON response into a Python dictionary.
  - **`print("Post Title:", post_data['title'])`** and **`print("Post Body:", post_data['body'])`** extract and print the **`title`** and **`body`** of the post.

Making a POST request

`POST` requests are used to send data to an API. This example demonstrates sending JSON data to a Uniform Resource Identifier (URI) endpoint.

Listing 9-c: postrequest.py

```
import requests

# Define the URL of the API endpoint
url = "https://jsonplaceholder.typicode.com/posts"

# Define the data to be sent in the POST request
data = {
    "title": "foo",
    "body": "bar",
    "userId": 1
}

# Make the POST request
response = requests.post(url, json=data)

# Check if the request was successful
if response.status_code == 201:
    # Parse the JSON response
```

```
    post_data = response.json()
    print("Created Post ID:", post_data['id'])
else:
    print("Failed to create post:", response.status_code)
```

Explanation:

- **Defining the data:** `data` is a dictionary containing the data sent in the POST request.
- **Making the `POST` request:** `response = requests.post(url, json=data)` sends the data as a JSON payload to the specified URL.
- **Checking and parsing the response:**
  - A successful `POST` request returns a status code `201`.
  - The JSON response contains the newly created post, and `print("Created Post ID:", post_data['id'])` prints the ID of the created post.

Parsing XML responses

Although JSON is more common across APIs and applications, some APIs return XML data. Python's `xml.etree.ElementTree` module can parse XML data. Here's how to parse XML data from an API response.

Listing 9-d: parsexml.py

```
import requests
import xml.etree.ElementTree as ET

# Define the URL of the API endpoint
url = "https://www.w3schools.com/xml/note.xml"

# Make the GET request
response = requests.get(url)

# Check if the request was successful
if response.status_code == 200:
    # Parse the XML response
    root = ET.fromstring(response.content)
    print("To:", root.find('to').text)
    print("From:", root.find('from').text)
    print("Heading:", root.find('heading').text)
    print("Body:", root.find('body').text)
else:
    print("Failed to retrieve data:", response.status_code)
```

Explanation:

- **Importing modules:**
  - `import xml.etree.ElementTree as ET` imports the XML parsing module.
- **Making the request:**
  - Similar to previous code snippet examples, a `GET` request is made to the URL `https://www.w3schools.com/xml/note.xml`, which returns XML data.
- **Parsing the XML response:**
  - `root = ET.fromstring(response.content)` parses the XML content into an `ElementTree` object.
  - `root.find('to').text` extracts the text of the <to> element and similar methods extract other elements.

Recap

By mastering API integration, you can significantly enhance the functionality of your applications, allowing them to interact with external data sources and services.

You can easily make HTTP requests and handle responses using the requests library. Understanding how to parse and manipulate JSON and XML data ensures you can effectively utilize the data returned by APIs.

Chapter 10 Unit Testing and Test-Driven Development

Quick intro

Unit testing and test-driven development (TDD) are critical practices in software engineering that help ensure code reliability and maintainability. In TDD, tests are typically written before the actual code, guiding the development process to meet specified requirements.

Python has powerful testing libraries, including **`unittest`** and **`pytest`**, which enable the creation of automated tests for your code.

Additionally, understanding test coverage and the use of mocks for testing isolated functions and components is essential for building robust applications.

> ***Note: You can run the following code examples by opening the built-in terminal within VS Code and executing the command:*** *`python <script>.py`*. ***Replace*** *`<script>`* ***with the name of the respective Python file to execute.***

Writing unit tests (unittest)

Python's built-in **`unittest`** framework provides a straightforward way to write tests. With it, you can define test cases and check conditions using assertions and test suites.

Let's create a simple **`Calculator`** class with basic operations (addition, subtraction, multiplication, and division) and write unit tests for each function.

Listing 10-a: calc.py

```
class Calculator:
    """A simple calculator class for basic arithmetic operations."""

    def add(self, a, b):
        return a + b

    def subtract(self, a, b):
        return a - b

    def multiply(self, a, b):
        return a * b

    def divide(self, a, b):
```

```
        if b == 0:
            raise ValueError("Cannot divide by zero")
        return a / b
```

Next, let's write unit tests for each method in this **`Calculator`** class using **`unittest`**.

Listing 10-b: testcalc.py

```
import unittest
from calculator import Calculator

class TestCalculator(unittest.TestCase):

    def setUp(self):
        """Set up a Calculator instance before each test method."""
        self.calc = Calculator()

    def test_add(self):
        """Test the addition method."""
        self.assertEqual(self.calc.add(2, 3), 5)
        self.assertEqual(self.calc.add(-1, 1), 0)

    def test_subtract(self):
        """Test the subtraction method."""
        self.assertEqual(self.calc.subtract(5, 3), 2)
        self.assertEqual(self.calc.subtract(0, 5), -5)

    def test_multiply(self):
        """Test the multiplication method."""
        self.assertEqual(self.calc.multiply(3, 4), 12)
        self.assertEqual(self.calc.multiply(-2, 3), -6)

    def test_divide(self):
        """Test the division method."""
        self.assertEqual(self.calc.divide(10, 2), 5)
        self.assertRaises(ValueError, self.calc.divide, 10, 0)

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

Explanation:

- **`Calculator` class:**
  - This class includes methods for basic arithmetic operations.
  - **`divide`** includes a check for division by zero, raising a **`ValueError`** if **`b`** is **`0`**.

- **Importing and setting up `unittest`:**
  - `import unittest` and `from calculator import Calculator` import the testing framework and the class to be tested.
  - `setUp` initializes a `Calculator` instance before each test.
- **Writing test methods:**
  - Each method tests a specific operation (`test_add`, `test_subtract`, etc.).
  - `self.assertEqual` checks if the result matches the expected value.
  - `self.assertRaises` verifies that divide raises a `ValueError` when dividing by zero.
- **Running the tests:**
  - The code checks for correctness across various cases, making it easy to identify errors if any of the tests fail.

Testing with pytest

The **`pytest`** library is a popular testing framework that provides simpler syntax, additional functionalities, and plugins for extended capabilities—**`pytest`** can run **`unittest`** tests, but it has its own syntax that's often more concise.

The following is the same test for the calculator, but written using **`pytest`** syntax.

Listing 10-c: testcalcpy.py

```
import pytest
from calc import Calculator

@pytest.fixture
def calc():
    """Fixture for Calculator instance."""
    return Calculator()

def test_add(calc):
    assert calc.add(2, 3) == 5
    assert calc.add(-1, 1) == 0

def test_subtract(calc):
    assert calc.subtract(5, 3) == 2
    assert calc.subtract(0, 5) == -5

def test_multiply(calc):
    assert calc.multiply(3, 4) == 12
    assert calc.multiply(-2, 3) == -6

def test_divide(calc):
    assert calc.divide(10, 2) == 5
```

```
    with pytest.raises(ValueError):
        calc.divide(10, 0)
```

Explanation:

- **Using fixtures:**
  - **`@pytest.fixture`** is a fixture that initializes **`Calculator()`** before each test, similar to **`setUp`** in **`unittest`**.
- **Simplified assertions**:
  - **`assert`** statements replace **`self.assertEqual`**, making the tests concise.
  - **`pytest.raises(ValueError)`** checks that dividing by zero raises the expected error.
- **Running pytest:**
  - Simply run **`pytest`** from the terminal to see organized test results.

Test coverage

Test coverage measures how much of your code is covered by tests. Python's coverage library works well with both **`unittest`** and **`pytest`** for measuring test coverage.

Listing 10-d: Install and Run Test Coverage

```
# Install coverage
pip install coverage

# Run coverage with pytest
coverage run -m pytest

# Generate a coverage report
coverage report -m
```

Explanation:

- **Installation: `pip install coverage`** installs the coverage package.
- **Running coverage: `coverage run -m pytest`** executes tests with coverage tracking.
- **Generating reports: `coverage report -m`** shows the percentage of code covered by tests, allowing you to identify untested sections.

Mocking

Sometimes, you must test code interacting with external systems like APIs and databases. Mocking allows you to replace these parts with mock objects to isolate the code being tested.

Let's expand the **`Calculator`** class to include an API call and demonstrate how to mock it in tests.

Listing 10-e: calcapi.py

```
import requests

class Calculator:

    def add(self, a, b):
        return a + b

    def get_random_number(self):
        response = requests.get("https://randomapi.com/api/random")
        if response.status_code == 200:
            return response.json()['number']
        else:
            return None
```

In this example, **`get_random_number`** makes a **`GET`** request to an API. Here's how to mock this API call.

Listing 10-f: mock.py

```
import unittest
from unittest.mock import patch
from calcapi import Calculator

class TestCalculatorWithAPI(unittest.TestCase):

    @patch("calcapi.get")
    def test_get_random_number(self, mock_get):
        """Mock the API call and test get_random_number."""
        calc = Calculator()

        # Define mock response data
        mock_response = mock_get.return_value
        mock_response.status_code = 200
        mock_response.json.return_value = {'number': 42}

        # Test the method with the mocked response
        self.assertEqual(calc.get_random_number(), 42)

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

Explanation:

- **Mocking the API request:**
  - `@patch("calcapi.get")` replaces the `requests.get` call in `Calculator` with a mock.
- **Configuring the mock:**
  - `mock_get.return_value` simulates the response object.
  - `mock_response.json.return_value = {'number': 42}` sets the JSON response.
- **Testing with mocked data:**
  - The statement `self.assertEqual(calc.get_random_number(), 42)` checks that the `get_random_number` method returns the mocked value.

Recap

Unit testing and TDD play a vital role in building reliable software. Using `unittest` and `pytest`, you can create structured test cases, maintain test coverage, and isolate code behavior using mocks.

These tools and techniques allow you to develop high-quality applications that are easier to maintain and scale over time.

Chapter 11 Memory and Performance

Quick intro

Effective memory management and performance optimization are essential for developing high-quality, efficient Python applications. Memory profiling can help identify areas where memory usage can be reduced, which is especially useful for large-scale applications or those running on limited resources.

Tools like **`memory_profiler`** and **`tracemalloc`** allow you to track memory consumption, while optimization techniques—such as using list comprehensions, generator functions, and the **`timeit`** module—can speed up code execution.

> ***Note: You can run the following code examples by opening the built-in terminal within VS Code and executing the command:*** *`python <script>.py`*. ***Replace*** *`<script>`* ***with the name of the respective Python file to execute.***

Memory profiling

Python's **`memory_profiler`** tool helps track memory usage in your code, providing insight into where memory bottlenecks occur.

Listing 11-a: Installing Memory Profiler

```
pip install memory_profiler
```

Let's create a function that generates an extensive list and profiles its memory usage.

Listing 11-b: memprofile.py

```
from memory_profiler import profile

@profile
def generate_large_list(n):
    """Function that generates a large list of squares from 1 to n."""
    large_list = [i ** 2 for i in range(n)]
    return large_list

# Run the function with profiling
generate_large_list(1000000)
```

Explanation:

- **Decorator `@profile`:**
  - The **`@profile`** decorator from **`memory_profiler`** tracks memory usage each time **`generate_large_list`** is called.
- **Function logic:**
  - **`generate_large_list`** creates a list of squares for each integer up to **`n`**.
  - The list comprehension (**`[i ** 2 for i in range(n)]`**) is concise but can use a lot of memory for large values of **`n`**.
- **Profiling:**
  - Run this script to see memory usage details, which can reveal if memory optimizations are needed.

Tracking memory allocation

Python's **`tracemalloc`** is another powerful module that provides more detailed insights into memory allocation, showing where each byte is allocated in your code.

Here's how to use **`tracemalloc`** to track memory allocation in a program that manages a list of numbers.

Listing 11-c: tracemem.py

```
import tracemalloc

def memory_intensive_task(n):
    """Function that generates a large list of squares from 1 to n and
returns it."""
    return [i ** 2 for i in range(n)]

# Start tracking memory
tracemalloc.start()

# Execute the function and get a snapshot
memory_intensive_task(1000000)
snapshot = tracemalloc.take_snapshot()

# Display the top memory blocks
top_stats = snapshot.statistics('lineno')

print("[Top 5 Memory Consumers]")
for stat in top_stats[:5]:
    print(stat)
```

Explanation:

- **Starting tracemalloc:**
  - **`tracemalloc.start()`** initializes the memory tracker, recording memory allocations.
- **Taking a snapshot:**
  - After calling the **`memory_intensive_task`** method, we take a memory snapshot with **`take_snapshot()`** to record memory usage at that specific point.
- **Displaying memory consumption:**
  - **`snapshot.statistics('lineno')`** provides a breakdown of memory usage by line, which helps identify the most memory-intensive parts of the code.
  - The loop then prints out the top five memory consumers in the code.

Code optimization with list comprehensions and generators

List comprehensions provide a more efficient way of creating lists, while generators allow you to handle large datasets without consuming excessive memory.

Consider the following code, which generates a list of squares. To illustrate memory optimization, we'll use both list comprehensions and generators.

Listing 11-d: codeopt.py

```
import timeit

# Using list comprehension
def squares_list(n):
    """Generates a list of squares using list comprehension."""
    return [i ** 2 for i in range(n)]

# Using generator
def squares_generator(n):
    """Generates squares one at a time using a generator."""
    for i in range(n):
        yield i ** 2

# Measure time for list comprehension
list_time = timeit.timeit("squares_list(1000000)", setup="from __main__
import squares_list", number=1)
print("List comprehension time:", list_time)

# Measure time for generator
generator_time = timeit.timeit("list(squares_generator(1000000))",
setup="from __main__ import squares_generator", number=1)
print("Generator time:", generator_time)
```

Explanation:

- **List comprehension** (`squares_list`):
  - Generates an entire list of squares at once and returns it.
  - While fast, it consumes a significant amount of memory for large lists.
- **Generator (`squares_generator`):**
  - `yield` is used instead of `return` to generate squares one at a time, only when needed.
  - This approach saves memory, as it doesn't store all values in memory at once.
- **Timing the functions:**
  - The `timeit.timeit` method measures the execution time of each function, comparing list comprehension and generator performance.
  - The results show the potential speed advantage of generators for some instances.

Profiling code execution time

Python's `timeit` is a module designed to measure the execution time of small code snippets, which helps identify and optimize slow code sections.

Here's a simple example that compares the execution time of two approaches to summing squares: a loop and a generator expression.

Listing 11-e: codeopt.py

```
import timeit

# Sum squares with a for loop
def sum_squares_loop(n):
    total = 0
    for i in range(n):
        total += i ** 2
    return total

# Sum squares with a generator expression
def sum_squares_generator(n):
    return sum(i ** 2 for i in range(n))

# Measure time for loop
loop_time = timeit.timeit("sum_squares_loop(1000000)", setup="from __main__
import sum_squares_loop", number=1)
print("Loop time:", loop_time)

# Measure time for generator expression
generator_time = timeit.timeit("sum_squares_generator(1000000)",
setup="from __main__ import sum_squares_generator", number=1)
```

```
print("Generator expression time:", generator_time)
```

Explanation:

- **Loop-based summation** (`sum_squares_loop`):
  - Iterates through numbers from `0` to `n` using a loop to calculate each number's square and add it to the `total`.
  - This approach is straightforward, but potentially slower than other methods.
- **Generator expression (`sum_squares_generator`):**
  - Uses `sum(i ** 2 for i in range(n))`, a memory-efficient generator expression that calculates and adds squares on the fly, which can be faster.
- **Timing with `timeit`:**
  - `timeit.timeit` measures the execution time for each function.
  - Running this script shows how generator expressions can outperform traditional loops in specific scenarios, making them a powerful optimization tool.

Recap

Memory management and performance optimization are crucial for creating efficient, scalable Python applications. Profiling tools like `memory_profiler` and `tracemalloc` help you understand and optimize memory usage.

Code optimization techniques, such as list comprehensions, generator functions, and timing measurements with `timeit`, enable you to refine your code for better performance. By mastering these techniques, you'll be well-equipped to write faster, memory-efficient Python code.

Chapter 12 Data Science and ML

Quick intro

Python is a powerful language in data science and machine learning, owing to its versatility and the rich ecosystem of libraries designed for data manipulation, scientific computing, and advanced machine learning techniques.

Libraries like Pandas, NumPy, scikit-learn, TensorFlow, and PyTorch are instrumental in helping data scientists and machine learning engineers process, analyze, and model data effectively.

> ***Note: You can run the following code examples by opening the built-in terminal within VS Code and executing the command:*** *`python <script>.py`*. ***Replace*** *`<script>`* ***with the name of the respective Python file to execute.***

Working with data using Pandas

Pandas provides powerful data manipulation capabilities, making it an essential library for handling large datasets, performing data analysis, and cleaning data.

Let's look at a fundamental data manipulation example with Pandas.

Listing 12-a: sample_data.csv

```
column_name,category_column,value_column
45,Category A,150
67,Category B,200
89,Category A,300
34,Category C,400
78,Category B,250
56,Category A,350
23,Category C,500
49,Category B,180
66,Category A,270
92,Category C,330
```

Listing 12-b: basicpandas.py

```
# Importing Pandas
import pandas as pd

# Loading a CSV file
```

```
data = pd.read_csv('sample_data.csv')

# Displaying the first few rows of the dataset
print(data.head())

# Descriptive statistics
print(data.describe())

# Filtering data
filtered_data = data[data['column_name'] > 50]

# Aggregating data
grouped_data = data.groupby('category_column')['value_column'].mean()
print(grouped_data)
```

Explanation:

- **Loading data:** **`pd.read_csv`** loads data from a CSV file into a **`DataFrame`**, Pandas's primary data structure.
- **Viewing data:** **`data.head()`** shows the first few rows, giving you a quick preview.
- **Descriptive statistics:** **`data.describe()`** provides statistical summaries, like mean and standard deviation, for numerical columns.
- **Filtering data:** Filters rows where **`column_name`** values are greater than **`50`**.
- **Aggregation:** Groups data by **`category_column`** and calculates the mean of **`value_column`** for each category.

Numerical computing with NumPy

NumPy is designed for numerical operations, supporting arrays, matrix operations, and high-level mathematical functions. Let's consider the following code that performs basic array operations with NumPy.

Listing 12-c: basicnumpy.py

```
# Importing NumPy
import numpy as np

# Creating an array
arr = np.array([1, 2, 3, 4, 5])

# Performing mathematical operations
arr_squared = arr ** 2
```

```
# Creating a 2D array
matrix = np.array([[1, 2], [3, 4]])

# Matrix multiplication
result = np.dot(matrix, matrix)
print("Matrix multiplication result:\n", result)
```

Explanation:

- **Array creation:** **`np.array`** creates a NumPy array, which is more efficient than a regular Python list. The **`np.array`** function infers a data type that depends on the data passed to it, type **`np.int64`** in this example.
- **Element-wise operations:** **`arr ** 2`** squares each element in the array.
- **2D arrays and matrix multiplication:** We create a 2D array (**`matrix`**) and use **`np.dot`** for matrix multiplication, an essential operation in linear algebra for machine learning.

Traditional machine learning with scikit-learn

scikit-learn provides a comprehensive suite of machine learning algorithms for classification, regression, clustering, and more, along with utilities for preprocessing and evaluation.

Let's look at an example of how to build a simple linear regression model with scikit-learn.

Listing 12-d: regmodel.py

```
# Importing necessary modules from scikit-learn
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error

# Sample data
X = [[1], [2], [3], [4], [5]]
y = [2, 4, 6, 8, 10]

# Splitting data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)

# Initializing and training the model
model = LinearRegression()
model.fit(X_train, y_train)

# Making predictions
```

```
predictions = model.predict(X_test)

# Evaluating the model
mse = mean_squared_error(y_test, predictions)
print("Mean Squared Error:", mse)
```

Explanation:

- **Data splitting:** `train_test_split` divides data into training and testing sets for model evaluation.
- **Model initialization and training:** We initialize and train a `LinearRegression` model using fit.
- **Prediction and evaluation:** predict generates forecasts on the test data, and `mean_squared_error` computes the error between actual and predicted values, providing insight into model accuracy.

Simple neural network with TensorFlow

TensorFlow and PyTorch offer sophisticated deep learning capabilities for more complex tasks, such as image recognition and natural language processing, enabling developers to create and train neural networks.

Let's look at a super simple example of how to build a neural network with TensorFlow.

Listing 12-e: nn.py

```
# Importing TensorFlow
import tensorflow as tf

# Sample dataset: Simple XOR problem
X_train = [[0, 0], [0, 1], [1, 0], [1, 1]]
y_train = [[0], [1], [1], [0]]

# Define a Sequential model
model = tf.keras.Sequential([
    tf.keras.layers.Dense(8, activation='relu', input_shape=(2,)),
    tf.keras.layers.Dense(1, activation='sigmoid')
])

# Compile the model
model.compile(optimizer='adam', loss='binary_crossentropy',
metrics=['accuracy'])

# Train the model
```

```
model.fit(X_train, y_train, epochs=100, verbose=0)

# Make predictions
predictions = model.predict(X_train)
print("Predictions:\n", predictions)
```

Explanation:

- **Data preparation:** We define the XOR problem as a dataset for training.
- **Model definition:** A Sequential model is created with two layers—one with `8` neurons and rectified linear unit (`relu`) activation, and another with `1` neuron and `sigmoid` activation for binary output.
- **Compilation and training:** `compile` defines the optimizer, loss function, and metrics, and `fit` trains the model for `100` `epochs`.
- **Prediction:** `predict` provides model outputs for the training set, which should approximate XOR results after training.

Deep learning with PyTorch

Here's an example of how to build the same XOR neural network model with PyTorch.

Listing 12-f: dl.py

```
# Importing PyTorch
import torch
import torch.nn as nn
import torch.optim as optim

# Sample data
X_train = torch.tensor([[0, 0], [0, 1], [1, 0], [1, 1]],
dtype=torch.float32)
y_train = torch.tensor([[0], [1], [1], [0]], dtype=torch.float32)

# Define the model
class XORModel(nn.Module):
    def __init__(self):
        super(XORModel, self).__init__()
        self.layer1 = nn.Linear(2, 8)
        self.layer2 = nn.Linear(8, 1)

    def forward(self, x):
        x = torch.relu(self.layer1(x))
        x = torch.sigmoid(self.layer2(x))
        return x
```

```
# Instantiate the model, define loss function and optimizer
model = XORModel()
criterion = nn.BCELoss()
optimizer = optim.Adam(model.parameters(), lr=0.01)

# Training loop
for epoch in range(100):
    optimizer.zero_grad()  # Reset gradients
    output = model(X_train)  # Forward pass
    loss = criterion(output, y_train)  # Compute loss
    loss.backward()  # Backward pass
    optimizer.step()  # Update weights

# Make predictions
with torch.no_grad():
    predictions = model(X_train)
    print("Predictions:\n", predictions)
```

Explanation:

- **Data preparation:** We define the XOR dataset with **`torch.tensor`**.
- **Model definition:** **`XORModel`** defines a two-layer neural network with **`relu`** and **`sigmoid`** activations.
- **Loss and optimizer:** Binary cross-entropy (BCE) is used as the loss function, while Adam optimizer updates model weights.
- **Training loop:** The loop performs forward and backward passes, updating weights with each epoch.
- **Prediction:** After training, predictions are made without tracking gradients (**`with torch.no_grad()`**), which should approximate the XOR pattern.

Recap

Python libraries like Pandas, NumPy, scikit-learn, TensorFlow, and PyTorch enable you to handle data efficiently, create machine learning models, and build deep learning architectures for complex tasks.

Mastering these libraries prepares you to work across the data science pipeline, from data preparation to model deployment, making Python an invaluable tool for data science and machine learning applications.

As these libraries are extensive and complex—and we've barely scratched the surface of what you can accomplish with them—I encourage you to dive deeper into each one if you are into data science and machine learning. The goal of this chapter was to give you a quick taste of their capabilities.

Chapter 13 Web Dev Basics

Quick intro

Python offers a robust suite of tools and frameworks for backend web development. Two popular frameworks are Flask and Django.

Flask is a lightweight, flexible framework for simple applications or when you need granular control. At the same time, Django is a high-level framework designed to make larger applications more accessible to build and maintain.

In this chapter, we'll create a simple RESTful API using Flask, illustrating how to set up routes, handle requests, and return responses, following RESTful principles.

> ***Note: You can run the following code examples by opening the built-in terminal within VS Code and executing the command:*** *`python <script>.py`*. ***Replace*** *`<script>`* ***with the name of the respective Python file to execute.***

Flask: Creating a simple REST API

In this example, we'll build a small API that manages a list of books. Each book will have an ID, title, and author.

We'll cover how to set up Flask, create routes to handle various HTTP methods, and build a RESTful structure. Before starting, you'll need to install Flask if it's not already in your environment.

Listing 13-a: Command to Install Flask

```
pip install Flask
```

Now, let's explore the following Flask code.

Listing 13-b: flaskapi.py

```
from flask import Flask, jsonify, request

# Initialize the Flask app
app = Flask(__name__)

# Sample data for the API
books = [
    {"id": 1, "title": "1984", "author": "George Orwell"},
```

```
    {"id": 2, "title": "To Kill a Mockingbird", "author": "Harper Lee"},
]

# Define routes below
```

Explanation:

- **Importing libraries:** Flask initializes the application, **`jsonify`** converts dictionaries to JSON format, and **`request`** retrieves data sent to the API.
- **Initialize the app:** **`app = Flask(__name__)`** creates the Flask app.
- **Sample data:** **`books`** is a list of dictionaries representing our data.

Defining RESTful API routes

We'll create routes that correspond to typical RESTful operations: **`GET`** to retrieve data, **`POST`** to add data, **`PUT`** to update data, and **`DELETE`** to remove data.

Let's expand the previous code (**flaskapi.py**) to include a route to get all books.

Listing 13-c: flaskapi.py

```
@app.route('/books', methods=['GET'])
def get_books():
    return jsonify(books), 200
```

Explanation:

- **Route:** **`@app.route('/books', methods=['GET'])`** binds this function to **`/books`** URL with the **`GET`** method.
- **Function:** **`get_books()`** returns all books in JSON format with a **`200`** HTTP status code, indicating a successful response.

Now, let's create a route to get a single book ID.

Listing 13-d: flaskapi.py

```
@app.route('/books/<int:id>', methods=['GET'])
def get_book(id):
    book = next((book for book in books if book["id"] == id), None)
    if book:
        return jsonify(book), 200
    else:
        return jsonify({"error": "Book not found"}), 404
```

Explanation:

- **Dynamic route:** `/<int:id>` allows dynamic handling of book IDs.
- **Logic:** Uses `next()` to retrieve a book with the given ID; if not found, returns a `404` error message.

Next, let's add a route to add a new book.

Listing 13-e: flaskapi.py

```
@app.route('/books', methods=['POST'])
def add_book():
    new_book = request.get_json()
    new_book["id"] = books[-1]["id"] + 1 if books else 1
    # Auto-increment ID
    books.append(new_book)
    return jsonify(new_book), 201
```

Explanation:

- `POST` **method:** This route listens for POST requests at `/books`.
- **Adding data:** `request.get_json()` fetches the JSON payload; the ID is incremented, and the new book is appended to the books list.
- **Response:** Returns the new book data and a `201` status code, indicating a resource was successfully created.

Now, let's add a route to update an existing book.

Listing 13-f: flaskapi.py

```
@app.route('/books/<int:id>', methods=['PUT'])
def update_book(id):
    book = next((book for book in books if book["id"] == id), None)
    if book:
        data = request.get_json()
        book.update(data)
        return jsonify(book), 200
    else:
        return jsonify({"error": "Book not found"}), 404
```

Explanation:

- `PUT` **method:** Listens for `PUT` requests at `/books/<int:id>`.
- **Updating data:** Finds the book by ID and updates it with data from `request.get_json()`.
- **Conditional response:** If the book is not found, it returns a `404` error.

Now, let's add a route to delete a book.

Listing 13-g: flaskapi.py

```
@app.route('/books/<int:id>', methods=['DELETE'])
def delete_book(id):
    global books
    books = [book for book in books if book["id"] != id]
    return jsonify({"message": "Book deleted"}), 200
```

Explanation:

- **`DELETE` method:** Listens for `DELETE` requests at **`/books/<int:id>`**.
- **Deleting data:** Uses a list comprehension to filter out the book with the specified ID.
- **Response:** Returns a message confirming deletion.

Run the Flask app

To run the application, we must add this at the end of the **flaskapi.py** script.

Listing 13-h: flaskapi.py

```
if __name__ == '__main__':
    app.run(debug=True)
```

Then, execute the app by invoking the **`python flaskapi.py`** command. The app will run on **http://127.0.0.1:5000**, where you can request your API routes.

This entire code snippet outlines how to use Flask to build a simple RESTful API, including routes for all CRUD operations, as previously discussed.

Listing 13-i: Finished - flaskapi.py

```
from flask import Flask, jsonify, request

# Initialize the Flask app
app = Flask(__name__)

# Sample data for the API
books = [
    {"id": 1, "title": "1984", "author": "George Orwell"},
    {"id": 2, "title": "To Kill a Mockingbird", "author": "Harper Lee"},
]
```

```python
# Define routes below
@app.route('/books', methods=['GET'])
def get_books():
    return jsonify(books), 200

@app.route('/books/<int:id>', methods=['GET'])
def get_book(id):
    book = next((book for book in books if book["id"] == id), None)
    if book:
        return jsonify(book), 200
    else:
        return jsonify({"error": "Book not found"}), 404

@app.route('/books', methods=['POST'])
def add_book():
    new_book = request.get_json()
    new_book["id"] = books[-1]["id"] + 1 if books else 1
    # Auto-increment ID
    books.append(new_book)
    return jsonify(new_book), 201

@app.route('/books/<int:id>', methods=['PUT'])
def update_book(id):
    book = next((book for book in books if book["id"] == id), None)
    if book:
        data = request.get_json()
        book.update(data)
        return jsonify(book), 200
    else:
        return jsonify({"error": "Book not found"}), 404

@app.route('/books/<int:id>', methods=['DELETE'])
def delete_book(id):
    global books
    books = [book for book in books if book["id"] != id]
    return jsonify({"message": "Book deleted"}), 200

if __name__ == '__main__':
    app.run(debug=True)
```

RESTful API design principles

REST—which stands for representational state transfer—is an architectural style for designing networked applications, which has become the cornerstone of most web APIs used today. Key principles include:

- **Stateless operations:** Each request is independent and contains all necessary information.
- **Uniform interface:** API design follows consistent, predictable URL structures and response formats.
- **HTTP methods:** `GET`, `POST`, `PUT`, `DELETE`, and `PATCH` map directly to CRUD operations.
- **Resource-based:** URLs represent resources (e.g., `/books/1` for a single book).
- **Stateless communication:** Each API call is independent, making it easier to scale.

Recap

In this chapter, we covered web development basics using Flask, a popular framework for Python. We created a RESTful API with CRUD capabilities and explored principles of REST API design, such as statelessness, resource-based URLs, and HTTP method conventions.

By following these principles, you can design and build scalable APIs that form the backbone of modern web and mobile applications.

With its simplicity and flexibility, Flask provides an excellent foundation for learning backend development in Python.

Chapter 14 Deploying Python Apps

Quick intro

Deploying Python applications in a consistent, scalable, and maintainable way is essential for moving from development to production. This chapter will cover two critical deployment topics:

- Packaging and managing dependencies with pip and virtual environments.
- Containerization with Docker and continuous integration (CI) and deployment (CD).

These approaches will help you manage dependencies, create reproducible environments, and automate deployment for seamless delivery.

Packaging with pip and virtual environments

Packaging your application is fundamental, allowing you to effectively distribute it and manage dependencies.

Step 1: Creating a virtual environment

A virtual environment isolates dependencies to avoid conflicts with other projects or system packages.

Listing 14-a: Creating a Virtual Environment

```
# Create a virtual environment
python3 -m venv myenv

# Activate the virtual environment
# On macOS/Linux
source myenv/bin/activate

# On Windows
myenv\Scripts\activate
```

Explanation:

- **Virtual environment:** The **`python3 -m venv myenv`** command creates a folder **`myenv`** where Python and **`pip`** will install packages in isolation.
- **Activation:** Activating the virtual environment directs Python to use this isolated environment.

Step 2: Installing dependencies with pip

Using **`pip`**, you can install required libraries, for example:

Listing 14-b: Install Required Libraries

```
pip install requests Flask
```

Explanation:

- **Package installation:** **`pip install requests Flask`** installs the **`requests`** and **`Flask`** libraries into the virtual environment, allowing your project to use them independently of global packages.

Step 3: Create a requirements.txt file

Once dependencies are installed, you can create a **requirements.txt** file that lists them. This file will let others install the same dependencies.

Listing 14-c: Creating a Requirements File

```
pip freeze > requirements.txt
```

Explanation:

- **Dependencies list:** **`pip freeze`** generates a list of currently installed packages and their versions, which **`> requirements.txt`** then writes into a file.

Step 4: Installing from requirements.txt

To recreate the environment elsewhere, install dependencies from **`requirements.txt`**.

Listing 14-d: Installing from Requirements

```
pip install -r requirements.txt
```

Explanation:

- **Install from the list:** **`pip install -r requirements.txt`** reads the file and installs the exact versions of each listed package, helping to replicate the environment precisely.

Docker and CI/CD

For consistent deployment, containerization with Docker and automation with CI/CD provide powerful tools for packaging and deploying applications.

Docker is a set of products and technology that uses OS-level virtualization to deliver software in packages called containers.

Step 1: Create a Dockerfile

A Dockerfile is a script defining the steps to create a Docker image, a blueprint for containers.

Listing 14-e: Dockerfile

```
# Use a base image with Python
FROM python:3.9-slim

# Set the working directory
WORKDIR /app

# Copy the requirements file and install dependencies
COPY requirements.txt .
RUN pip install -r requirements.txt

# Copy the application code
COPY . .

# Expose port and define startup command
EXPOSE 5000
CMD ["python", "app.py"]
```

Explanation:

- **Base image:** `FROM python:3.9-slim` defines a lightweight base image with Python 3.9.
- **Working directory:** `WORKDIR /app` sets the working directory inside the container to `/app`.
- **Dependency installation:** `COPY requirements.txt .` and `RUN pip install -r requirements.txt` copy and install project dependencies.
- **Application files:** `COPY . .` copies all local files to `/app` in the container.
- **Port exposure:** `EXPOSE 5000` tells Docker to expose port `5000`, commonly used by Flask.
- **Startup command:** `CMD ["python", "app.py"]` runs the `app.py` script, starting the application.

Step 2: Building and running the Docker image

Build the Docker image using the Dockerfile, then run the container.

Listing 14-f: Building and Running the Docker Image

```
# Build the Docker image
docker build -t myapp .

# Run the container
docker run -p 5000:5000 myapp
```

Explanation:

- **Build:** The **`docker build -t myapp .`** command builds an image named **`myapp`** based on the Dockerfile.
- **Run:** The **`docker run -p 5000:5000 myapp`** command starts the container, mapping port **`5000`** of the container to port **`5000`** on the host, making the app accessible at **`localhost:5000`**.

Step 3: Setting up CI/CD with GitHub actions

CI/CD pipelines automate testing and deployment. We'll create a simple GitHub Actions workflow to build and test the application.

Let's create a **.github/workflows/main.yml** file in your repository.

Listing 14-g: main.yaml

```
name: Python Application CI/CD

on: [push, pull_request]

jobs:
  build:
    runs-on: ubuntu-latest

    steps:
    - name: Checkout code
      uses: actions/checkout@v2

    - name: Set up Python
      uses: actions/setup-python@v2
      with:
        python-version: '3.9'

    - name: Install dependencies
      run: |
        python -m venv venv
```

```
        source venv/bin/activate
        pip install -r requirements.txt

    - name: Run tests
      run: |
        source venv/bin/activate
        python -m unittest discover -s tests
```

Explanation:

- **Trigger events:** `on: [push, pull_request]` initiates the workflow on pushes and pull requests.
- **Setup steps:**
  - **Checkout:** `actions/checkout@v2` checks out the repository code.
  - **Python setup:** `actions/setup-python@v2` sets up Python 3.9.
  - **Dependencies and tests:** Installs dependencies in a virtual environment and runs tests using unittest.

Step 4: Automating Docker deployment

You can automate Docker image deployment by adding Docker login and push steps to the GitHub Actions workflow.

Listing 14-h: final.yaml

```
 - name: Build Docker image
   run: docker build -t myapp .

 - name: Log in to Docker Hub
   uses: docker/login-action@v1
   with:
     username: ${{ secrets.DOCKER_USERNAME }}
     password: ${{ secrets.DOCKER_PASSWORD }}

- name: Push Docker image
  run: |
        docker tag myapp your-dockerhub-username/myapp:latest
        docker push your-dockerhub-username/myapp:latest
```

Explanation:

- **Docker build:** Builds the Docker image using `docker build`.
- **Docker hub login:** Uses GitHub secrets (`DOCKER_USERNAME` and `DOCKER_PASSWORD`) to authenticate.
- **Docker push:** Tags and pushes the image to Docker Hub.

Docker offers limitless deployment possibilities, which I encourage you to explore in your own time, as we barely scratched the surface with these examples. The idea was to give you a taste of what's possible with Docker.

Recap

Deploying Python applications involves packaging with pip and virtual environments, containerization with Docker, and automation with CI/CD tools.

Understanding these steps ensures your application is portable, scalable, and consistently deployed across different environments.

Appendix CRUD Command-line App

Quick intro

As a treat, here's a small CRM command-line app in Python, which uses SQLite as a database and SQLAlchemy for ORM.

This app enables freelancers to keep track of customers and projects with basic CRUD (create, read, update, delete) operations.

Prerequisites

Before running the finished code, you'll need to install SQLAlchemy using the **`pip install sqlalchemy`** command.

Project structure

We'll structure our code into several parts, all within the same Python file:

- **Database setup:** Define our database and models.
- **CRUD functions:** Define functions to create, read, update, and delete customers and projects.
- **User Interface:** Add command-line prompts for interaction.

App code

Here's the finished application code.

Listing Appendix-a: crm.py

```
from sqlalchemy import create_engine, Column, Integer, String, ForeignKey
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import sessionmaker, relationship

# Step 1: Database setup
# ----------------------

# Initialize the database engine for SQLite
engine = create_engine("sqlite:///crm.db")
Base = declarative_base()
```

```
# Define the Customer model
class Customer(Base):
    __tablename__ = 'customers'

    id = Column(Integer, primary_key=True)
    name = Column(String, nullable=False)
    email = Column(String, nullable=False)
    phone = Column(String, nullable=True)

    # Establish one-to-many relationship with projects
    projects = relationship("Project", back_populates="customer",
cascade="all, delete-orphan")

# Define the Project model
class Project(Base):
    __tablename__ = 'projects'

    id = Column(Integer, primary_key=True)
    title = Column(String, nullable=False)
    description = Column(String, nullable=True)
    customer_id = Column(Integer, ForeignKey('customers.id'))

    # Establish link to the customer
    customer = relationship("Customer", back_populates="projects")

# Create tables
Base.metadata.create_all(engine)

# Set up session factory for database interaction
Session = sessionmaker(bind=engine)
session = Session()

# Step 2: CRUD functions
# ----------------------

# Create a new customer
def create_customer(name, email, phone=None):
    new_customer = Customer(name=name, email=email, phone=phone)
    session.add(new_customer)
    session.commit()
    print(f"Customer '{name}' created.")

# Read all customers
def read_customers():
```

```
    customers = session.query(Customer).all()
    for customer in customers:
        print(f"{customer.id}: {customer.name} ({customer.email},
{customer.phone})")

# Update a customer
def update_customer(customer_id, name=None, email=None, phone=None):
    customer = session.query(Customer).get(customer_id)
    if customer:
        customer.name = name if name else customer.name
        customer.email = email if email else customer.email
        customer.phone = phone if phone else customer.phone
        session.commit()
        print(f"Customer '{customer.name}' updated.")
    else:
        print("Customer not found.")

# Delete a customer
def delete_customer(customer_id):
    customer = session.query(Customer).get(customer_id)
    if customer:
        session.delete(customer)
        session.commit()
        print(f"Customer '{customer.name}' deleted.")
    else:
        print("Customer not found.")

# Create a new project
def create_project(title, description, customer_id):
    customer = session.query(Customer).get(customer_id)
    if customer:
        new_project = Project(title=title, description=description,
customer=customer)
        session.add(new_project)
        session.commit()
        print(f"Project '{title}' created for customer '{customer.name}'.")
    else:
        print("Customer not found.")

# Read all projects for a customer
def read_projects(customer_id):
    customer = session.query(Customer).get(customer_id)
    if customer:
        for project in customer.projects:
```

```
            print(f"{project.id}: {project.title} ({project.description})")
    else:
        print("Customer not found.")

# Update a project
def update_project(project_id, title=None, description=None):
    project = session.query(Project).get(project_id)
    if project:
        project.title = title if title else project.title
        project.description = description if description else
project.description
        session.commit()
        print(f"Project '{project.title}' updated.")
    else:
        print("Project not found.")

# Delete a project
def delete_project(project_id):
    project = session.query(Project).get(project_id)
    if project:
        session.delete(project)
        session.commit()
        print(f"Project '{project.title}' deleted.")
    else:
        print("Project not found.")

# Step 3: User Interface
# ----------------------

def main():
    while True:
        print("\n--- CRM System ---")
        print("1. Create Customer")
        print("2. View Customers")
        print("3. Update Customer")
        print("4. Delete Customer")
        print("5. Create Project")
        print("6. View Projects for Customer")
        print("7. Update Project")
        print("8. Delete Project")
        print("9. Exit")

        choice = input("Choose an option: ")
```

```
        if choice == "1":
            name = input("Enter customer name: ")
            email = input("Enter customer email: ")
            phone = input("Enter customer phone (optional): ")
            create_customer(name, email, phone)
        elif choice == "2":
            read_customers()
        elif choice == "3":
            customer_id = int(input("Enter customer ID to update: "))
            name = input("Enter new name (blank to keep current): ")
            email = input("Enter new email (blank to keep current): ")
            phone = input("Enter new phone (blank to keep current): ")
            update_customer(customer_id, name, email, phone)
        elif choice == "4":
            customer_id = int(input("Enter customer ID to delete: "))
            delete_customer(customer_id)
        elif choice == "5":
            customer_id = int(input("Customer ID for new project: "))
            title = input("Enter project title: ")
            description = input("Enter project description (optional): ")
            create_project(title, description, customer_id)
        elif choice == "6":
            customer_id = int(input("Customer ID to view projects: "))
            read_projects(customer_id)
        elif choice == "7":
            project_id = int(input("Enter project ID to update: "))
            title = input("New title (blank to keep current): ")
            description = input("New description (blank as is): ")
            update_project(project_id, title, description)
        elif choice == "8":
            project_id = int(input("Enter project ID to delete: "))
            delete_project(project_id)
        elif choice == "9":
            print("Exiting CRM System.")
            break
        else:
            print("Invalid choice, please try again.")

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

The application begins by importing SQLAlchemy modules, which are essential for managing the database and defining data models. **`create_engine`** connects to our SQLite database, while **`declarative_base`** serves as a foundational base class for our data models.

We also import **`Column`**, **`Integer`**, **`String`**, and **`ForeignKey`** to define specific attributes of each table in the database, and **`sessionmaker`** and **`relationship`** to manage database sessions and establish relationships between tables.

A SQLite database is created and connected using **`create_engine`**. This database is stored as a file named **crm.db**. The **`declarative_base`** class is then set up, allowing us to define our models (tables) using a common base.

This setup establishes the environment for creating tables and connecting Python data models to SQL database structures.

The **`Customer`** model represents the customer table in the database, defining the structure and fields each customer record will contain. Key attributes include:

- An **`id`** column serves as the primary key that uniquely identifies each customer.
- Additional columns for customer **`name`**, **`email`**, and an optional **`phone`** field.
- A **`relationship`** attribute connects customers with multiple **`projects`**, enabling a one-to-many link. Through this, we set up cascading delete rules, ensuring that all associated projects are removed from the database when a customer is deleted.

The **`Project`** model represents a table for storing project data linked to a customer. It includes:

- An **`id`** column as the primary key for uniquely identifying each project.
- Fields for a project **`title`** and an optional **`description`**.
- A foreign key **`customer_id`** to connect each project to a specific customer, forming the other side of the one-to-many relationship.
- This setup allows each project to access its associated customer through a relationship to the **`Customer`** model.

The **`Base.metadata.create_all(engine)`** instruction generates the tables in the SQLite database according to the **`Customer`** and **`Project`** models. If the tables already exist, they won't be recreated, ensuring data persistence and avoiding overwrites.

A **`session`** is configured to handle interactions with the database. The session allows us to add, retrieve, update, and delete records.

This session layer abstracts raw SQL queries, making database operations smoother and more Pythonic by leveraging SQLAlchemy's ORM capabilities.

The app includes CRUD functions for managing customers and projects:

- **Create:** Functions for adding new customers and projects to the database by accepting relevant information as arguments and committing them as new records.
- **Read:** Retrieval functions allow us to view all customers, specific customer details, all projects, and details of particular projects. The session.query method retrieves data based on specified criteria, making it easy to fetch records.

- **Update:** Update functions enable modifications to customer or project details. By finding the record, modifying its fields, and committing the changes, we ensure the database remains up-to-date without redundant entries.
- **Delete:** Deletion functions allow for removing specific customer or project records. SQLAlchemy's delete functionality cascades deletions for linked entries, ensuring data integrity and eliminating orphan records.

Finally, the command-line interface allows users to run each CRUD operation. Users can add, view, update, or delete customer and project records by calling specific functions through a simple command-line prompt.

The entire app, from database setup to CRUD operations, is designed to be a practical, user-friendly tool for freelance projects and customer management.

Each step in this codebase demonstrates essential Python, SQLAlchemy, and database management concepts, providing a solid foundation for building more complex applications.

Multiple files

It's also possible to split this application into multiple Python files rather than having all the logic in a single **.py** file.

To structure this CRM into multiple Python files, we would organize it into modular components, separating each functional part of the application into distinct files.

This approach improves maintainability and readability, making navigating and updating individual components easier.

Listing Appendix-b: Multiple Files Project Structure

```
crm/
├── models/
│   ├── __init__.py
│   ├── customer.py
│   ├── project.py
├── database.py
├── crud_operations.py
├── main.py
├── requirements.txt
└── README.md
```

models: Folder for data models (customer and project tables)

- **models/__init__.py:**
  - Initializes the **models** package so we can easily import these models elsewhere.
  - Contains imports for all models, making it simpler to import them from other application parts (e.g., **from models import Customer, Project**).

Listing Appendix-c: __init__.py

```
from .customer import Customer
from .project import Project
```

Explanation:

- **models/customer.py:**
  - Defines the **Customer** model.
  - Contains the SQLAlchemy model class for the customer, including its attributes and relationships.
  - Defines **Customer**'s structure, such as **id**, **name**, **email**, and the relationship with **Project**.

Listing Appendix-d: customer.py

```
from sqlalchemy import Column, Integer, String
from sqlalchemy.orm import relationship
from database import Base

class Customer(Base):
    __tablename__ = 'customers'

    id = Column(Integer, primary_key=True)
    name = Column(String, nullable=False)
    email = Column(String, nullable=False)
    phone = Column(String, nullable=True)

    projects = relationship('Project', back_populates='customer',
cascade="all, delete-orphan")

    def __repr__(self):
        return f"<Customer(id={self.id}, name={self.name},
email={self.email})>"
```

Explanation:

- **models/project.py:**
  - Defines the `Project` model.
  - Contains the SQLAlchemy model class for **`Project`**, including attributes like **`id`**, **`title`**, **`description`**, and the foreign key linking it to **`Customer`**.

Listing Appendix-e: project.py

```
from sqlalchemy import Column, Integer, String, ForeignKey
from sqlalchemy.orm import relationship
from database import Base

class Project(Base):
    __tablename__ = 'projects'

    id = Column(Integer, primary_key=True)
    title = Column(String, nullable=False)
    description = Column(String, nullable=True)
    customer_id = Column(Integer, ForeignKey('customers.id'),
nullable=False)

    customer = relationship('Customer', back_populates='projects')

    def __repr__(self):
        return f"<Project(id={self.id}, title={self.title},
customer_id={self.customer_id})>"
```

database.py: Database setup and session management

- Establishes the database connection (using **`create_engine`**).
- Imports **`Base`** from **`declarative_base`** and initializes the database tables.
- Configures a session factory (**`sessionmaker`**) for managing database transactions.
- Example usage: **`from database import session`**, to access and work with the database session in other application parts.

Listing Appendix-f: database.py

```
from sqlalchemy import create_engine
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import sessionmaker

# Connect to the SQLite database
DATABASE_URL = 'sqlite:///crm.db'
```

```
engine = create_engine(DATABASE_URL)
SessionLocal = sessionmaker(autocommit=False, autoflush=False, bind=engine)
Base = declarative_base()

# Create all tables
def init_db():
    Base.metadata.create_all(bind=engine)
```

crud_operations.py: Contains all CRUD functions

- Defines the CRUD functions for **`Customer`** and **`Project`**, including:

  `create_customer`, **`get_customer`**, **`update_customer`**, **`delete_customer`**.

  `create_project`, **`get_project`**, **`update_project`**, **`delete_project`**.

- Imports **`models`** from the **models** folder and session management from **database.py**.
- This organization separates the app logic from database access functions, which can be imported into **main.py**.

Listing Appendix-g: crud_operations.py

```
from models.customer import Customer
from models.project import Project
from database import SessionLocal

# Create a new customer
def create_customer(name, email, phone=None):
    session = SessionLocal()
    new_customer = Customer(name=name, email=email, phone=phone)
    session.add(new_customer)
    session.commit()
    session.close()
    return new_customer

# Get all customers
def get_customers():
    session = SessionLocal()
    customers = session.query(Customer).all()
    session.close()
    return customers

# Update a customer's information
```

```
def update_customer(customer_id, name=None, email=None, phone=None):
    session = SessionLocal()
    customer = session.query(Customer).get(customer_id)
    if customer:
        customer.name = name if name else customer.name
        customer.email = email if email else customer.email
        customer.phone = phone if phone else customer.phone
        session.commit()
    session.close()

# Delete a customer
def delete_customer(customer_id):
    session = SessionLocal()
    customer = session.query(Customer).get(customer_id)
    if customer:
        session.delete(customer)
        session.commit()
    session.close()

# Create a new project for a customer
def create_project(title, description, customer_id):
    session = SessionLocal()
    new_project = Project(title=title, description=description,
customer_id=customer_id)
    session.add(new_project)
    session.commit()
    session.close()
    return new_project

# Get all projects for a specific customer
def get_projects_by_customer(customer_id):
    session = SessionLocal()
    projects = session.query(Project).filter(Project.customer_id ==
customer_id).all()
    session.close()
    return projects
```

main.py: Entry point for the CRM app

- The command-line interface logic for interacting with the user.
- Imports CRUD functions from **crud_operations.py** and organizes them into a user-friendly interface.

- Provides the main loop or menu system where users can select options (create, read, update, delete).
- Calls CRUD functions based on user input.

Listing Appendix-h: main.py

```
from database import init_db
from crud_operations import create_customer, get_customers,
update_customer, delete_customer
from crud_operations import create_project, get_projects_by_customer

# Initialize the database
init_db()

def main_menu():
    print("Welcome to the CRM CLI App")
    print("1. Add a new customer")
    print("2. View all customers")
    print("3. Update a customer")
    print("4. Delete a customer")
    print("5. Add a project to a customer")
    print("6. View projects for a customer")
    print("0. Exit")

def main():
    while True:
        main_menu()
        choice = input("Enter your choice: ")

        if choice == "1":
            name = input("Enter customer name: ")
            email = input("Enter customer email: ")
            phone = input("Enter customer phone (optional): ")
            create_customer(name, email, phone)
            print("Customer added successfully!")

        elif choice == "2":
            customers = get_customers()
            for customer in customers:
                print(f"{customer.id}: {customer.name} - {customer.email}")

        elif choice == "3":
            customer_id = int(input("Enter customer ID to update: "))
            name = input("Enter new name (leave blank to skip): ")
            email = input("Enter new email (leave blank to skip): ")
```

```
            phone = input("Enter new phone (leave blank to skip): ")
            update_customer(customer_id, name, email, phone)
            print("Customer updated successfully!")

        elif choice == "4":
            customer_id = int(input("Enter customer ID to delete: "))
            delete_customer(customer_id)
            print("Customer deleted successfully!")

        elif choice == "5":
            customer_id = int(input("Customer ID for new project: "))
            title = input("Enter project title: ")
            description = input("Enter project description (optional): ")
            create_project(title, description, customer_id)
            print("Project added successfully!")

        elif choice == "6":
            customer_id = int(input("Customer ID to view projects: "))
            projects = get_projects_by_customer(customer_id)
            for project in projects:
                print(f"{project.id}: {project.title} -
{project.description}")

        elif choice == "0":
            print("Exiting the CRM CLI App.")
            break

        else:
            print("Invalid choice. Please try again.")

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

requirements.txt: Lists dependencies

- Includes **`sqlalchemy`** and possibly other libraries like **`sqlite3`**, if needed.
- This file helps to easily install dependencies by running the following command: **`pip install -r requirements.txt`**.

Listing Appendix-i: requirements.txt

```
sqlalchemy
```

README.md: Documentation

Finally, the **README.md** file describes the app's purpose, setup instructions, and how to use it. It is also helpful in providing an overview of any particular configuration steps.

Although there are many ways to organize a Python project, this suggested file separation structure keeps your code organized and makes locating and editing specific functionalities easier.

It also supports scalability, as adding new features or models can be done with minimal changes to the overall structure.

Closing thoughts

Congratulations on completing this book! This journey has taken you through the core aspects of Python, from basic constructs like variables and control flow to advanced techniques, including asynchronous programming, file handling, and error management.

Each section has been designed to give you both a foundational understanding and practical experience, enabling you to tackle real-world problems with Python.

You've explored critical libraries for data manipulation, machine learning, and web development and learned best practices for deploying and optimizing Python applications.

As you continue to build upon these skills, you'll find Python a versatile and powerful tool, adaptable to various applications, from web development to data science.

With this knowledge, you can further develop as a Python programmer. Remember that programming is a continuous learning process, so keep experimenting, exploring, and challenging yourself.

Thank you for choosing this book as your guide—here's to your future success in Python! Until next time, take care and keep learning.