PYTHON MASTERY MADE SIMPLE FOR BEGINNERS

The Complete Guide in 10 Days with Practical Exercises and Explosive Secrets to Master Python Programming and Propel Yourself Toward Your Dream Job!

ROBERT MATT

Python Programming and Propel Yourself Toward Your Dream Job!

By

ROBERT MAT

Contents

Introduction

Python is a high-level programming language that can be used for a wide variety of applications and is lauded for its ease of use and readability. It is one of the languages that are used most frequently in the world, and several businesses and organizations make use of it, notably Google, Facebook, and NASA, to name just a few.

Python is an excellent language to learn if you are just starting out since it is simple and straightforward. Nevertheless, it is also a strong language that is appropriate for a broad variety of activities, such as machine learning, data analysis, and the building of websites.

This book will teach you all you need to know to become an expert in Python programming in only ten days by following the steps in the book. It is jam-packed with useful activities that you can do in real life and explosive insights that will make learning Python simpler and faster for you.

By the end of this book, you will be able to:

- Write your own Python code
- Use Python modules and libraries
- Develop Python applications
- Propel yourself toward your dream job

Chapter 1: Introduction to Python

1.1: A Brief History of Python

Python is a programming language that has clarity and accessibility in its design. Because it supports several paradigms, programmers can create code in an object-orientated, structural, or functional manner, depending on their preference. Today, it is used extensively in various domains, including web development, machine learning (via frameworks like TensorFlow and Keras) and more. It is the second most popular programming language in use around the globe, behind JavaScript.

Python's journey started at the end of the 1980s and the beginning of the 1990s, and since then, it has become one of the programming languages most extensively used and adored across a broad variety of sectors. Van Rossum, a programmer from the Netherlands, started to work on Python in December 1989 when he was on holiday at the Center for Wiskunde and Informatica (CWI) in the

Netherlands. He was on vacation over the Christmas holiday. He aimed to design a simple language to read, write, and keep up.

Python 2.0 was made available and has newly added capabilities and upgrades. However, as time passed, Python 2's design decisions resulted in several changes, making it more challenging to continue developing and maintaining the language.

In reaction, Python 3 was developed and made available to the public on December 3, 2008. Python 3 marked a significant step forward thanks to implementing forward-incompatible enhancements to improve the program's consistency and eliminate code deemed excessive.

But Python 3 is partially backward compatible with Python 2, which has surpassed the end of its lifespan and is no longer continuously developed. When moving between versions, you will need to make some modifications to your code to accommodate the differences.

Python's readability, versatility, and ease of use have contributed to the programming language's ever-increasing popularity over the years. The clarity and simplicity of its syntax made it popular among developers since it made it easier for them to write and comprehend code. By the 2010s, Python had already established itself as one of the most widely used programming languages, and its use in the industry has only increased since then. Using Python as a project language by various information technology companies, startups, and other organizations has helped foster Python's community and environment growth.

Python is a strong and malleable programming language that may be used for various applications due to its adaptability. There are several situations in which it is possible that it is not the best choice. Real-time systems, software that requires a lot of resources, the creation of mobile applications, trading at high frequencies, system-level programs programming, memory-intensive soft-

ware, gaming on mobile devices or consoles, and specific cryptographic techniques are all examples of this kind of software.

However, it is crucial to remember that Python's constraints may often be worked around using the necessary libraries and tools. This is something that should be remembered. Ultimately, the project's needs, the team's knowledge, and the development goals will decide which programming language is used. Python is a widely used and helpful programming language that can be used independently or in cooperation with other languages to fulfill various duties. Python may be used alone or in combination with other languages.

1.2: Why Python is important?

Python is a cutting-edge programming language that is object-oriented, can be interpreted, and is utilized in interactive scripting. Because Python was designed to be relatively simple to comprehend, it commonly uses English phrases instead of the punctuation typical in other languages. In ad-

dition, Python has a significantly reduced number of syntactical structures compared to other languages.

Programming in Python is a versatile talent that is used in practically every industry, including the following:

- Study of data
- Calculation in the sciences and mathematics
- The building blocks of a game
- Developing for the web
- Commerce and financial dealings
- Scripting of both a general and application-specific nature
- Automation of systems and administrative management
- Graphics created on a computer.
- Cartography and geography (often using GIS software)
- Examination of vulnerabilities in security systems

Potential for Financial Gain:

According to Indeed, Python is the programming language with the second-highest average salary. A yearly income of around USD 110,026 might be expected on average. There is no need for concern! The biggest money may be made if you get employment with Selby Jennings. There, the average annual wage is 245,862 dollars.

Facilitation of Understanding:

The ease of use and general availability of the Python programming language constitute two of its most recognized strengths. Acquiring it only requires a little work as studying other programming languages since its syntax is composed in English in contrast to the syntax of other programming languages. Because it was named after Monty Python's Flying Circus, we can be sure that the person who designed this code had a sense of humor. This is one of the reasons why it is so simple to implement, and it is also one of the reasons why it was named after the show. It performs an excellent job of managing difficulty, which frees

you to focus on knowing the benefits of Python programming instead of the hassles of learning the minute details. Since it does an excellent job of managing complexity, it opens you to focus on grasping the positive aspects of Python programming. The fact that it is available for free and under an open-source license is the cherry on top.

Mobility and adaptability:

Python is a programming language renowned for its versatility and expedited learning process, characterized by a minimal learning curve. Because over 125,000 third-party libraries for Python are available, it is conceivable to use Python for tasks including machine learning, the processing of online content, and even biology. In addition, it is pre-installed with data-oriented modules like pandas, NumPy, and matplotlib. These libraries provide exceptional capabilities in

processing, manipulating, and presenting data, contributing to its widespread use in data analysis.

1.3: Versatility of Python

Due to its many advantages, Python programming is rapidly becoming more valuable in today's competitive labor market. According to ZDNet, Python is now among the top three programming dialects in the world and is well on its way to being the most used language overall. Python is the world's most widely used programming language, as shown by the PYPL. Therefore, those aspiring to get career opportunities in other countries may have favorable prospects in countries like Switzerland or Australia. In which location do you aspire to get employment? Adding Python to your abilities could be all needed to get you somewhere.

Python is wonderful, but should one bother to learn it? The correct response is yes. To begin, there are a variety of professional paths available using Python. If you are skilled in Python, you can pursue careers as a data scientist or a ma-

chine learning engineer, a developer for the web, a software developer, or even a DevOps engineer or security analyst. Python is used regularly in these roles, including but not limited to scripts, structures, and data models.

Employment using Python is quite lucrative. This may be one of the more significant benefits of acquiring Python for most of us. Based on the reviewing website Glassdoor, the average annual salary for a Python developer in the USA is 102 thousand dollars, while the average annual salary for an engineer specializing in machine learning is 132 thousand dollars. What a juicy! The level of salary is commensurate with the level of demand.

Developing a website:

Python is widely recognized as a very adaptable programming language, offering the potential to streamline the creation process of intricate internet products. Python's many web frameworks, such as the well-known Django, are very helpful. Frameworks such as these may simplify and speed

up the process of developing the back-end and the client-side functionality. Python is utilized for web development by even the largest corporations; for example, YouTube and Google have both made substantial use of Python for many of their technological infrastructures.

Data Science:

Python was shown to be the most popular programming language among data scientists in a poll conducted by Kaggle, a community on the internet for those working in data science. It's not only the time-saving modules that contribute to Python's success; the programming language's capacity to conduct rapid analysis of enormous data sets and complete jobs with a high degree of repetition are also major factors.

App Development:

Anyone interested in app development should seriously consider learning Python as their language of choice. Due to the reduced time and effort

required for its creation, it is a great language for prototyping. Python's ability to run on several platforms simplifies developing mobile and desktop applications, just as it does for websites. Python, in particular, has lately seen a surge in popularity in the rapidly expanding fields of distributed ledger app development and game app development due to the sophisticated infrastructure and real-time testing capabilities that Python has.

1.4: Installing Python

Step 1: Download the Python Installer:

- Go to the Python download page on the official website for Windows.
- Find a Python release that is reliable. Python version 3.10.10 was used for testing purposes in this lesson.
- For downloading the executable file, choose the relevant link for your operating system: either the Windows installer for 64-bit Windows or the Windows installation for 32-bit Windows.

Step 2: Putting the executable installer through its paces:

- Double-clicking the.exe file, such as python-3.10.10-amd 64.exe, will start the Python installation after downloading it.
- By selecting the Download launcher option for all users, you will provide all machine users access to the Python launcher program.
- Users can run Python from the terminal by choosing the "Add python.exe to PATH." checkbox.
- Click the Install Now button and go on to Step 4 to verify that Python was successfully installed if this is your first time using Python and you wish to install it with the default features, as indicated in the dialogue box. To install additional features, both basic and complicated, select the "Customize installation" button and then proceed.

- Even if you do not intend to use any of the tools or resources included in the Python Optional Features, you may still download all of them on your computer.
- It is suggested that you test and learn using the Python test suite.
- Python launcher and the following applies to all users: advised to provide users with the ability to start Python via the command prompt.
- Make your selections from the available choices that correspond to your preferences:
- Installation for each user: strongly suggested if you are not the only occupant of this machine.
- Associating files with Python is highly recommended because selecting this option will link all the Python script types with the editor or launcher.

- Make desktop shortcuts for apps that have been installed; it is advised that you activate shortcuts for Python programs.
- Adding Python to the environment variables is recommended to allow it to be launched.
- Precompiling the standard library is unnecessary, and doing so might slow down the installation.
- Get the symbols used for debugging and the debug binaries: advised just if you want to develop C or C++ add-ons.
- Make a note of the directory in which Python was installed, just in should you need to return to it later.
- To begin the installation, click the Install button.
- After the installation, a notification stating that the setup was successful will show.

Step 3: Adding Python to the Environment Variables (optional):

- If you choose to go to Step 2, skip this step. During the installation process, add Python to the environment variables.
- You can access Python via the command line even if you did not add Python to the system variables during installation. You can do this manually if you want to use Python via the command line.
- Find the directory that Python was installed in on your computer before you get started.
- Python can be found in the folder located at C:\Users\Sarah\AppData\Local\Programs\Python\Python310: a directory that will be in the Windows user path if you didn't pick Install for all users while you were installing it.

It should be noted that the folder's title will vary based on the Python version installed. How-

ever, it will consistently commence with the term "Python."

- Click the Start button, then type advanced system settings into the search window.
- To see the advanced system options, click the see button.
- To change the environment variables for your computer, open the System Properties dialogue and go to the Advanced tab.

Depending on the configuration you have:

- During the installation process, if you choose the option to Install for all users, pick Route from the available list of System Variables and then click the Edit button.
- Once you have chosen "New" and entered the path to the Python list, hit "OK" several times until all the dialogue boxes are gone.

Step 4: Check that the Python installation was successful:

- You may use the command line or the Integrated Development Environment (IDLE) program if you decide to install it to check whether the Python installation was successful. Either way, you can verify that the installation was completed successfully.
- To initiate the process, locate the Start button and proceed to input the command "cmd" into the designated search field. Select the Command Prompt button.
- At the prompt for entering commands, enter the following command: Python -- the latest version.

1.5: How to install Python on MAC, Linux

Install Python on MAC:

Python 2.7 is pre-installed on macOS, but if you want to update to the most recent version of the programming language, then you'll have to follow the following instructions:

- First, check to see what version of Python is currently installed on your machine. To do this, you must launch the terminal, type "python –version," and click the ENTER key. This will provide you with the Python Version installed on your Mac device.
- To install Python in its most recent form on your computer, visit the following website: https://www.python.org/downloads/. Choose macOS as your operating system, then go to the page where you may download the. Pkg file.
- Now, open the download to install Python on your Mac system. Pkg file, and then follow the on-screen instructions to complete the installation.

- You will need to click Continue a few times at this point, and then, as seen in the picture that follows, you will see a button to install the program.
- As illustrated in the picture that follows, after you have pressed the install button, you will be prompted to enter the system password before the installation can begin.
- Click the "Install Software" button after entering the password, and it will immediately begin installing the most recent version of Python. You could confirm if the installation succeeded by using the same command that ed at the beginning to get the Python version.
- Use the command "python3.11 –version" in the terminal to determine the most recent version of Python installed on your computer.

Install Python on LINUX:

Python is pre-installed on Linux operating systems, as we have just gone over, but if you wish to set up Python on your machine, then you need to follow the instructions:

Download and set up the development packages needed as prerequisites to install Python on Debian and Fedora. These packages may be found in the python-devel package. To install the essential development dependencies on your machine, you must perform the following command in your terminal: executing the command sudo apt-get build-dep python3.

When you have successfully installed these packages, you need to obtain the most recent version of Python from the following address: https://www.python.org/downloads/, and you need to be particular for your system, such as for Linux. After that, you may go on to the next step.

It is now possible for you to install Python on Linux by only extracting the tar file and complet-

ing the installation with some standard parameters and instructions entered into your terminal:

Tar -xf archive.tar.xz (command prompt)

Please note that the downloaded file's name may change in certain circumstances, as shown by the following example: "python-3.11.2.tar.xz." Therefore, it would work out best if you extracted the downloaded file using the same name as when you downloaded it.

In the last stage of the process, you must use a code or text editor to check and see whether the installation was successful. One can utilize any text editor to write the following code into a new Python file, which you can then save. Print ('Installation Completed Successfully'). And give this piece of code the name "test.py" when you save it.

To determine whether the code functions properly, you must launch the device and go where you first stored this Python file. The following command should be used to run the file. Python script

named "test.py. "The results should be "Installation accomplished with success."

This indicates that the Python software was successfully installed on your Linux machine and is now operational.

Install Python on UBUNTU:

Python is pre-installed on all Linux distributions; however, you may only sometimes find it pre-installed on Ubuntu operating systems. Installing Python may be done in several ways, depending on the circumstances. Python will be installed on your Ubuntu computer with the help of APT, which stands for Advanced Package Tool and is the standard package manager on Ubuntu. To set up Python on Ubuntu, refer to the instructions that are listed below:

- On Ubuntu, you may launch the terminal by simultaneously pressing the following keys: Control, Option, and Tab

- Now, to access the system's repository, run the following command into the terminal:
- Update apt using sudo.
- To set up Python on Ubuntu, use the command "Sudo apt install python3."
- Then, APT will search for the Python installation package and install it on your computer without further intervention.

1.6: Exercise, Solutions and Explanations

In this instance, we have used the built-in print() method to display the string "Hello, world!" on our screen. A series of characters is referred to as a string. Strings may be encapsulated in a single, double, or triple quote while working in Python.

Exercise 1:

Print "Hello World" in Python.

Code:

```
# Print Hello World
print("Hello World")
```

Output:

```
PS C:\Users\TOWER TECH\Desktop\Codes> & "C:/Program Files/Python311/python.exe" "c:/Users/TOWER TECH/Desktop/Codes/Chapter1_codes.py"
Hello World
PS C:\Users\TOWER TECH\Desktop\Codes>
```

Explanation:

We could see the string "Hello World" on our screen because our application used the built-in technique known as print(). A string is the name given to the characters that make up a string. In Python, strings between single, double, or even triple quotes may be enclosed.

Exercise 2:

Write a Python program that prints your name.

Code:

```
# Get the user's name
name = input("Enter your name: ")
# Print the name
print("Hello, " + name + "!")
```

Output:

Explanation:

To communicate with this program's user, we use the input() method. The input() method prompts the user to provide their name by displaying the text "Enter your name:" and then waiting for them to do so. The variable name will keep a string representation of anything the user puts in there.

The next step involves displaying a message using the print() method. The message is created by using the + operator to concatenate (join) three strings: "Hello, "the user's name (which is saved in the name variable), and "!"

The last step of the process is for the program to print the welcoming message. This message will contain the user's name.

Exercise 3:

Develop a computer software that computes the summation of two numerical values inputted by the user.

Code:

```
# Get two numbers from the user
num1 = float(input("Enter the first number: "))
num2 = float(input("Enter the second number: "))
# Calculate the sum
sum = num1 + num2
# Print the result
print("The sum of {} and {} is: {}".format(num1, num2,
sum))
```

Output:

```
PS C:\Users\TOWER TECH\Desktop\Codes> & "C:/Program Files/Python311/python.exe" "c:/Users/TOWER TECH/Desktop/Codes/Chapter1_codes.py"
Enter the first number: 1
Enter the second number: 17
The sum of 1.0 and 17.0 is: 18.0
PS C:\Users\TOWER TECH\Desktop\Codes>
```

Explanation:

In the first stage of our application, we use the input() function so that the end user may provide us with two numerical values. We ask the user to input their initial quantity, and when they do so,

we preserve it in a variable with the name num1 that we have created for that purpose. In addition, we will ask for another number and save this information in the variable designated as num2. We use the float(input()) method to convert the user's input into a floating-point number so that we may treat it as such. A floating-point number can include decimal places. This is done to ensure that the input is processed correctly.

Once we have both values, we employ the plus symbol (plus) to add them shared, and once the computation is complete, we store the result in a variable that we name total.

To display the result, we use the print() function in combination with an expression previously created. In the prepared string, these characters have been used as substitutes for denoting the contents of num1, num2, and total, respectively. The. Format () method will replace these dummy values with the actual values instead of using them. The

result is a device that is educational and simple to use.

Chapter 2: Fundamentals of Python

2.1: Basic Syntax

Python is a computer language designed from the ground up to be easily readable. Python is quickly becoming among the programming languages that are used by both inexperienced and seasoned developers alike. This is mostly because the language was developed with the user in mind. Because the grammar of Python is so like that of the language of English, it is much easier to write, read, and understand a Python script than a program created with a different programming language, like C or Java.

The "syntax" of a language refers to its structure, or more precisely, to the components that go into producing a well-formed program. Further, the Python syntax consists of a set of rules and building blocks that, when taken together, make up the syntax of Python.

The Line Structure of Python:

The actual lines of text and the statements and lines that explain what each line does are included in the structure of the Python source code. A physical line in a program written in Python is made up of a succession of people, and the line sequence is finished when the very last characters of the line are entered. This contrasts with other programming languages, such as C and C++, where it is necessary to use a semicolon to indicate that the statement has been completed. In Python, the line sequence may be considered complete as soon as the last character of the line has been typed. On the other hand, an orderly path is created by using one or more real boundaries as its foundational building pieces throughout the construction process. While Python programmers are not obliged to use the semicolon in their code, the language does not prevent them from doing so. The appearance of the NEWLINE token denotes that the chain of reasoning has arrived at its logical endpoint.

Python Multiline Statements:

As you have seen, the start of an additional statement in Python is indicated by the start of a new line in the programming language. Despite this, Python can merge many statements into a single logical line and change a single statement to one that spans multiple lines. The clarity with which the sentence may be read has room for improvement.

Explicit Line Joining:

In computer programming, a method known as explicit line joining may be used to split a single statement over many lines of code. The use of a backslash to signify the extension of the statement into the next line is all that is required to accomplish this goal.

Code:

```
# Explicit line joining with \
result = 10 + 20 + 30 + \
         40 + 50 + 60
print(result)
```

Output:

Explanation:

The backslash character that is denoted is the one that is used if an explicit line joining is desired. Explicit line join is a way to separate one logical line into many physical lines. This is accomplished using the backslash to indicate that the line will continue. Specifying that the line persists is how this objective may be met. It gives Python the instruction to treat the line that next follows the line that is now being processed to be an extension of the line that is being processed. After completing this process, the sum of the numbers in this range, which ranges from 10 to 60, will result in a value of 210.

Implicit Line Joining:

Statements enclosed within brackets, braces, or parentheses can be divided into multiple physical lines without a backslash.

Code:

```
# Implicit line joining with parentheses
numbers = (1 + 2 + 3 +
           4 + 5 + 6)
print(numbers)
```

Output:

```
PS C:\Users\TOWER TECH\Desktop\Codes> & "C:/Program Files/Python311/python.exe" "c:/Users/TOWER TECH/Desktop/Codes/Chapter1_codes.py"
21
PS C:\Users\TOWER TECH\Desktop\Codes>
```

Explanation:

Parentheses are being used to link lines together in a more subtle way. Python allows you to split a single conceptual line into numerous physical lines using parenthesis. It has been broken out among numerous lines to make the addition operation easier to understand. Python will continue to scan the next line till it finds an appropriate closing parenthesis if it comes across an open parenthesis (since it understands that the calculation is not yet complete when it sees one). When the digits from 1 to 6 are put together in this scenario, the total value of the resulting numbers is 21.

Whitespaces and Indentation:

In contrast to many programming languages, Python relies on indentation to indicate the starting point of a new block of code. This is opposed to the bulk of programming languages.

Most computer programming languages make code indentation optional, yet they provide it so that the program may be presented more efficiently. However, Python needs this. The significance of using appropriate indentation in Python cannot be emphasized enough.

Code:

```
variable=1
if variable>0 :
print("There is no indentation in this program")
```

Output:

```
PS C:\Users\TOWER TECH\Desktop\Codes> & "C:/Program Files/Python311/python.exe" "c:/Users/TOWER TECH/Desktop/Codes/Chapter1_codes.py"
  File "c:\Users\TOWER TECH\Desktop\Codes\Chapter1_codes.py", line 37
    print("There is no indentation in this statement")
    ^
IndentationError: expected an indented block after 'if' statement on line 36
PS C:\Users\TOWER TECH\Desktop\Codes>
```

Python Quotations:

When manipulating strings in the Python programming language, it is possible to use single or double quotation marks. Neither of these options is recommended. If we start a phrase with a single quotation, we must end it with one. The same idea applies when using two quotation marks.

Code:

```
print ('Single quote example')
print ("Double quote example")
```

Output:

```
PS C:\Users\TOWER TECH\Desktop\Codes> & "C:/Program Files/Python311/python.exe" "c:/Users/TOWER TECH/Desktop/Codes/Chapter1_codes.py"
Single quote example
Double quote example
PS C:\Users\TOWER TECH\Desktop\Codes>
```

2.2: Variables and data types

What Are Variables in Python?

Variables and data types in Python are value pairs that may change at any point during the execution of the program, just as their names indicate. When dealing with a computer language, you may use a

section of memory known as a variable to store a value. Variables are quite common. The value you have stored may change at some time in the future based on the conditions placed upon it.

Python considers a variable to have been formed as soon as it is given a value; in other words, the moment a value is assigned to a variable in Python. Python does not need additional commands or syntax to declare a variable.

Variable Definition & Declaration:

Declaring a variable in Python does not need any further instructions. The variable is considered to have been declared immediately as the value was put into it. When we are defining a variable, there are a few guidelines that we need to always keep in mind:

- The variable name can't begin with a number. It is always required to begin with either a character or an underscore.

- Python is case-sensitive when it comes to its variables.
- They can only include alphabetic, numeric, and underscore characters.
- There must be no use of special characters.

Code:

```
# Variable declaration with type annotations
name: str  # A string variable
age: int  # An integer variable
height: float  # A floating-point variable



# Variable assignments
name = "Alice"
age = 25
height = 5.8



# You can also declare and assign in one step
City: str = "New York"
```

```
print(name)
print(age)
print(height)
print(City)
```

Output:

```
PS C:\Ebooks> & "C:/Program Files/Python311/python.exe" c:/Ebooks/Book_Codes.py
Alice
25
5.8
New York
PS C:\Ebooks>
```

Explanation:

The code supplied has variable declarations annotated with their types; more specifically, 'name' is declared to be a string, 'age' is declared to be a number, and 'height' is declared to be a float. The values for these variables are then determined later. The 'City' variable is likewise initialized as a string, and its value is set to "New York." After that, the software moves on to the next step: show the values allocated to the variables.

Data Types in Python:

There are six different sorts of Python data, and its

attributes classify each.

Numerical Data Types in Python:

The numerical data type stores the value of a number. In the realm of numerical data, there are also four subtypes. The many sub-types of the numerical data type are:

- Integers
- Float
- Complex Numbers
- Boolean

Integers:

Integers are the values that are used for denoting entire numbers. We may use the type() method to determine the data type of any variables we work with. It will give the data type associated with the variable that was provided.

Code:

```
# Integer variable
age = 25
```

```
# Performing operations with integers
x = 10
y = 5
sum = x + y
difference = x - y
product = x * y
quotient = x / y
remainder = x % y



# Printing integer values and results
print("Age:", age)
print("Sum:", sum)
print("Difference:", difference)
print("Product:", product)
print("Quotient:", quotient)
print("Remainder:", remainder)
```

Output:

```
PS C:\Users\TOMER TECH\Desktop\Codes> & "C:/Program Files/Python311/python.exe" "c:/Users/TOMER TECH/Desktop/Codes/Chapter1_codes.py"
Age: 25
Sum: 15
Difference: 5
Product: 50
Quotient: 2.0
Remainder: 0
```

Explanation:

This Python code assigns the value 25 to the integer variable age, executes fundamental mathematical operations (addition, subtracting, multiplication, division, and remainder), and then displays the results along with the age value. x and y are both given the values 10 and 5, respectively. The computer software computes and outputs to the console the following numbers: the total (15), the difference (5), the product (50), the quotient (2.0), and the remainder (0).

Boolean:

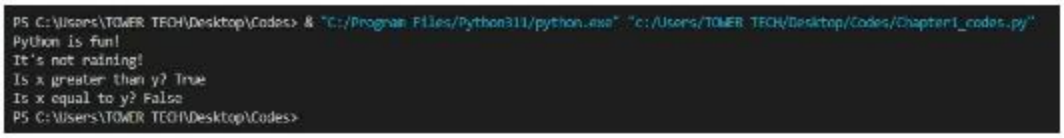
Since the conclusion of a Boolean is true or false, it is used for producing categorical output.

Code:

```
# Boolean variables
is_python_fun = True
is_raining = False
```

```
# Using boolean values in conditional statements
if is_python_fun:
    print("Python is fun!")



If not is_raining:
    print("It's not raining!")



# Comparing values and storing boolean results
x = 10
y = 5
is_greater = x > y
is_equal = x == y



print("Is x greater than y?", is_greater)
print("Is x equal to y?", is_equal)
```

Output:



Explanation:

Python code shows the usage of conditional statements as well as boolean variables. It begins by declaring two boolean variables, one of which, is_python_fun, has the value True, while the other, is_raining, has the value False. After that, it utilizes these boolean values in conditional expressions to check whether Python is fun (which is True, so it prints "Python is fun!"), and if it's not raining (which is also True, it publishes "It's not raining!"). Both conditions are met, so it outputs "Python is fun!" and "It's not raining!" respectively. In the last step of the process, the code compares the values of the integers x and y, saving the boolean results in the variables is_greater (which has the value True since x is bigger than y) and is_equal (which has the value False as x is not equal to y). The code prints these comparisons with the

messages "Is x greater than y? True" and "Is x equal to y? True". To the console with the value "False."

Strings:

Python's way of representing Unicode character values is via using strings. Python does not contain a character data type; a single character is treated like any other string in the programming language. Because of strings' immutable nature, it is impossible to modify a string once it has been replaced.

Code:

```
# Define a string variable
message = "Hello, World!"



# Print the string
print(message)
```

Output:

```
PS C:\Users\TOWER TECH\Desktop\Codes> & "C:/Program Files/Python311/python.exe" "c:/Users/TOWER TECH/Desktop/Codes/Chapter1_codes.py"
Hello, World!
PS C:\Users\TOWER TECH\Desktop\Codes>
```

Explanation:

The following Python code creates a string variable called message, and the text "Hello, World!" is assigned to it. The output of this code will be sent to the console, and it will include whatever is stored in the message variable. This is accomplished via the usage of the print statement. Since the print function displays the contents of the message's variable when invoked, "Hello, World!" will be shown on the screen when the code is run. In its most basic form, this code presents the idea of string variables and explains how to render text in Python using the print command.

2.3: Operators in Python

One description describes the operator as a symbol that causes a certain action to be carried out when placed between two operands. The logic of a program written in a certain programming language is built atop the operators, which are the language's fundamental building blocks. Some op-

erators in each programming language are responsible for more than one function.

Arithmetic Operators:

Arithmetic operators are placed between the two operands during a given arithmetic operation. A wide variety of arithmetic operators are available. It contains the + (addition), - (subtraction), * (multiplication), / (division), % (remainder), and // (floor division) operators in addition to the exponent operator (**).

Addition:

It is used in the process of adding two operands. For instance, if a is equal to 10 and b is equal to 10, then a + b equals 20.

Code:

```
# Addition
a = 10
b = 10
result = a + b
print("Addition:", result)
```

Output:

```
PS C:\Ebooks> & "C:/Program Files/Python311/python.exe" c:/Ebooks/Book_Codes.py
Addition: 20
PS C:\Ebooks>
```

Explanation:

In this code, we define two variables named a and b, assigning each of them the value 10, performing an addition using the plus sign (+), and then saving the total in a variable called result. After that, the code displays the total value of a and b, which is 20.

Subtraction:

It takes away from the initial operand the value represented by the second operand. When the first operand has a value that is lower than the following operand, the value that is calculated is negative. For instance, if (a) equals 20 and (b) equals 5, then (a-b) equals 15.

Code:

```
# Subtraction
a = 20
```

```
b = 5
result = a - b
print("a - b =", result)
```

Output:

```
PS C:\Ebooks> & "C:/Program Files/Python311/python.exe" c:/Ebooks/Book_Codes.py
a - b = 15
PS C:\Ebooks>
```

Explanation:

We will assign the values 20 and 5 to the variables a and b, respectively, in this code. We will subtract using the minus operator (-) and then save the result in the variable named result. After that, the code outputs the variation between a and b, which is 15.

Division:

After dividing the initial operand by the next operand, it will return the obtained quotient. For instance, if a is equal to 20 and b is equal to 10, then a/b equals 2.0.

Code:

```
# Division
a = 20
b = 10
result = a / b
print("a / b =", result)
```

Output:

```
PS C:\Ebooks> & "C:/Program Files/Python311/python.exe" c:/Ebooks/Book_Codes.py
a / b = 2.0
PS C:\Ebooks>
```

Explanation:

In this part of the definition, we will correspondingly give the parameters a and b the values 20 and 10. We execute division using the / operator, then save the result in the named result variable. The code outputs the quotient produced by the division of a by b, which is 2.0.

Multiplication:

It is used in the process of multiplying one operand by another. Take, for instance, the equation a * b = 80, where a = 20 and b = 4.

Code:

```
# Multiplication
a = 20
b = 4
result = a * b
print("a * b =", result)
```

Output:

```
PS C:\Ebooks> & "C:/Program Files/Python311/python.exe" c:/Ebooks/Book_Codes.py
a * b = 80
PS C:\Ebooks>
```

Explanation:

We will assign the values 20 and 4 to the variables a and b, respectively, in this code. We execute multiplication using the * operator, then put the result in the named result variable. The next thing the code does is display the sum of a and b, equal to 80.

Reminder:

After completing the division of the initial operand by the following operand, it returns the remainder. For instance, if a is equal to 20, and b is equal to 10, then a%b = 0

Code:

```
# Remainder (Modulus)
a = 20
b = 10
result = a % b
print("a % b =", result)
```

Output:

```
PS C:\Ebooks> & "C:/Program Files/Python311/python.exe" c:/Ebooks/Book_Codes.py
a % b = 0
PS C:\Ebooks>
```

Explanation:

In this part of the definition, we will give variables a and b values of 20 and 10. We will conduct the modulus operation using the % operator, then save the result in the variable named result. The residual, zero when an is divided by b, is then

printed by the code when the division is performed.

Exponent:

An exponent operator is called because it determines the initial operand's power concerning the second operand.

Code:

```
# Exponentiation
base = 2
exponent = 3
result = base ** exponent
print("base ** exponent =", result)
```

Output:

```
PS C:\Ebooks> & "C:/Program Files/Python311/python.exe" c:/Ebooks/Book_Codes.py
base ** exponent = 8
PS C:\Ebooks>
```

Explanation:

This code's base and exponent variables have been initialized to values 2 and 3, respectively. Exponentiation is accomplished with the usage of the ** operator. It then saves the result in the result variable after the base calculation increases the ex-

ponent's power. In the above instance, the result derived from exponentiating 2 by the value of 3 is 8. Subsequently, the code proceeds to display the result denoted as 8.

Floor Division:

The floor value of the quotient, which is derived by dividing both operands, is provided by this operand.

Code:

```
# Floor Division
a = 21
b = 5
result = a // b
print("a // b =", result)
```

Output:

```
PS C:\Ebooks> & "C:/Program Files/Python311/python.exe" c:/E
a // b = 4
PS C:\Ebooks>
```

Explanation:

Both variables a and b are assigned the integers 21 and 5, respectively, when defined in this code. When doing a floor division, we use the // operator and save the result in the variable named result. The code then prints the floor division of a by b, which yields the value 4. The result of a floor division is the biggest integer that is either less than the quotient of the division or the same size as it.

Exercise:

Write a Python code that executes mathematical operations on two variables, denoted as 'a' and 'b'. These variables are initially assigned the values of 32 and 6, respectively. The operations to be performed include addition, subtraction, multiplication, division, remainder, exponentiation, and floor division. The code should report the results of each operation.

Code:

```
# Initialize two variables with values
a = 32
b = 6



# Addition: Adding the values of a and b
addition_result = a + b
print('Addition of two numbers:', addition_result)



# Subtraction: Subtracting the value of b from a
subtraction_result = a - b
print('Subtraction of two numbers:', subtraction_result)



# Multiplication: Multiplying the values of a and b
multiplication_result = a * b
print('Multiplication of two numbers:', multiplication_re-
sult)



# Division: Dividing the value of a by b
```

```
division_result = a / b
print('Division of two numbers:', division_result)



# Reminder (Modulus): Finding the remainder when a is
divided by b
remainder_result = a % b
print('Reminder of two numbers:', remainder_result)



# Exponent: Calculating a to the power of b
exponent_result = a ** b
print('Exponent of two numbers:', exponent_result)



# Floor Division: Dividing a by b and rounding down to
the nearest integer
floor_division_result = a // b
print('Floor division of two numbers:', floor_division_re-
sult)
```

Output:

Explanation:

The numbers 32 and 6 are assigned to the variables a and b, respectively, when initialized. The value returned by addition_result, subtraction_result, multiplication_result, division_result, remainder_result, exponent_result, and floor_division_result represents the operation's result.

Comparison operator:

The primary function of comparison operators is for comparative reasons. The outcomes of the two operands are compared using comparison operators, and then the operators yield a true or false Boolean result depending on the outcome. Examples of comparison operators are ==,!=, <=, >=, >, <

Equal To:

In this use, ==

The expression will return the value True if the value of a is equivalent to the value of b, but it will return the value False otherwise.

Code:

```
# Equal To
a = 10
b = 10
result = a == b
print("a == b:", result)
```

Output:

```
PS C:\Ebooks> & "C:/Program Files/Python311/python.exe" c:/Ebooks/Book_Codes.py
a == b: True
PS C:\Ebooks>
```

Explanation:

The == operator is used in this piece of code to determine whether or not the value of an is equivalent to the value of b. If they are not equal, it will return False, but if they are, it will return True. Because a and b are equal to 10 in this demonstra-

tion, the program outputs the statement "a == b: True."

Not Equal To:

In this use, !=

The expression will yield the value True if the value of an is not equal to the value of b, and it will return the value False otherwise.

Code:

```
# Not Equal To
a = 10
b = 5
result = a != b
print("a != b:", result)
```

Output:

```
PS C:\Ebooks> & "C:/Program Files/Python311/python.exe" c:/Ebooks/Book_Codes.py
a != b: True
PS C:\Ebooks>
```

Explanation:

To determine whether or not the value of a differs from the value of b, the not equal (!=) operator is used. It will return True if they are not equal but False if they are equal. In this particular instance, an is 10, and b is 5. Hence, the program outputs the statement "a!= b: True."

Less Than:

In this use <

If the numerical value of a is lower than that of b, then this function will return True; otherwise, it will return False.

Code:

```
# Less Than
a = 5
b = 10
result = a < b
print("a < b:", result)
```

Output:

```
PS C:\Ebooks> & "C:/Program Files/Python311/python.exe" c:/Ebooks/
a < b: True
PS C:\Ebooks>
```

Explanation:

By comparing the two, the operator determines whether or not the value of an is lower than the amount of b. It will return True if the value of an is smaller than that of b, but it will return False otherwise. Because an is equal to 5 and b is equal to 10, the code outputs "a b: True" in this illustration.

Less Than Equals To:

In this use <=

That Which Comes Back If the value of a is lower than or equivalent to the value of b, then this variable will return True; otherwise, it will return False.

Code:

```
# Less Than or Equal To
a = 5
b = 5
result = a <= b
print("a <= b:", result)
```

Output:

```
PS C:\Ebooks> & "C:/Program Files/Python311/python.exe" c:/Ebooks/Book_Codes.py
a <= b: True
PS C:\Ebooks>
```

Explanation:

The = operator determines if the amount of an is lower than the value of b or if they are equal. It will return True if the value a is smaller or more equal to the value b, but it will return False otherwise. Because numbers a and b are equal to 5 in this demonstration, the program outputs the statement "a = b: True."

Greater Than:

In this use, >=

The expression will yield the value True if the value of a is higher than that of b, but it will return the value False otherwise.

Code:

```
# Greater Than
a = 10
```

```
b = 5
result = a > b
print("a > b:", result)
```

Output:

```
PS C:\Ebooks> & "C:/Program Files/Python311/python.exe" c:/Ebooks/Book_Codes.py
a > b: True
PS C:\Ebooks>
```

Explanation:

The > operator determines if the numerical value of an is larger than that of b by comparing the two numbers. It will return True if the value a is bigger than the value b, but it will return False otherwise. In this particular instance, a is 10, and b is 5. Hence, the program outputs the statement "a > b: True."

Greater Than Equals To:

In this use, >=

Returns True if the value of a is larger than or equal to the value of b; otherwise, it returns False. Meaning: If the values are not equal, the function returns False.

Code:

```
# Greater Than or Equal To
a = 10
b = 10
result = a >= b
print("a >= b:", result)
```

Output:

```
PS C:\Ebooks> & "C:/Program Files/Python311/python.exe" c:/Ebooks/Book_Codes.py
a >= b: True
PS C:\Ebooks>
```

Explanation:

The >= operator determines if the amount of an is larger than the value of b or whether the two values are equal. It will return True if the value a is larger or more equal to the value b, but it will return False otherwise. Because both a and b are 10 in this demonstration, the program outputs the statement "a >= b: True."

Exercise:

How do I use different comparison functions to compare values a and b in Python? What do these comparisons do?

Code:

```
# Define two variables for comparison
a = 10
b = 5



# Equal To (==)
result_equal = a == b
print(f"Equal To: {a} == {b} -> {result_equal}")



# Not Equal To (!=)
result_not_equal = a != b
print(f"Not Equal To: {a} != {b} -> {result_not_equal}")



# Less Than (<)
result_less_than = a < b
print(f"Less Than: {a} < {b} -> {result_less_than}")
```

```
# Less Than or Equal To (<=)
result_less_than_equal = a <= b
print(f"Less Than or Equal To: {a} <= {b} -> {result_
less_than_equal}")


# Greater Than (>)
result_greater_than = a > b
print(f"Greater Than: {a} > {b} -> {result_greater_than}")


# Greater Than or Equal To (>=)
result_greater_than_equal = a >= b
print(f"Greater Than or Equal To: {a} >= {b} -> {result_
greater_than_equal}")
```

Output:

```
PS C:\Users\TOWER TECH\Desktop\Codes> & "C:/Program Files/Python311/python.exe" "c:/Users/TOWER TECH/Desktop/Codes/Chapter1_codes.py"
Equal To: 10 == 5 -> False
Not Equal To: 10 != 5 -> True
Less Than: 10 < 5 -> False
Less Than or Equal To: 10 <= 5  > False
Greater Than: 10 > 5 -> True
Greater Than or Equal To: 10 >= 5 -> True
```

Explanation:

The above piece of code illustrates the application of six distinct comparison operators. It gives the variables a and b an initial value of 10 and a value of 5 correspondingly. First, the (==) operator determines if a and b are equivalent to one another, and the outcome is saved in the result_equal variable. Similarly, it uses the operator not equal to (!=) to check whether an is not equal to b and then saves the result in a variable called result_not_equal. After that, the less than () operator is used to determine whether or not an is smaller than b, and the outcome is saved in the variable result_less_than. In addition to this, it uses an operator called "less than or equal to" (=) to determine if an is less than or equal to b, and then it saves the outcome in a variable called "result_less_than_equal".Alongside this, it uses the greater than (>) operator to determine whether or not a is greater than b and then saves the result in the variable result_greater_than. At last, it uses the operator for

greater than or equal to (>=) to determine if a is greater than or equal to b, and then it saves the result in a variable called result_greater_than_equal. The findings of each comparison are then output by the code, together with an indication of whether each precondition has been met.

Assignment Operators in Python:

Assignment operators are utilized in programming for assigning values to variables. The assignment is sometimes done directly; sometimes, the operator performs a mathematical operation and assigns the value to the operand.

Assignment Operator:

In this use =

It is used while assigning the value located to its right to the parameter located to its left. For instance, the expression a = 10 tells the computer to give the variable a 10.

Addition Assignment Operator:

In this use, +=

It adds the value to its right with the variable's value to its left, then allocates the outcome to the variable to its left. For instance, the expression a += 5 is the same as saying that a = a + 5.

Subtraction Assignment Operator:

In this use -=

It subtracts by taking the value to its right and subtracting it from the variable to its left, then transferring the outcome to the variable to its left. For instance, the expression a -= 3 is the same as the expression a = a - 3.

Multiplication Assignment Operator:

In this use, *=

The value on its right is multiplied by the variable on its left, and the product is then assigned to the variable on its left. For instance, the expression a *= 2 is the same as saying that a = a * 2.

Modulus Assignment Operator:

In this use, %

It does the calculation necessary to determine the remaining after the value on its right is applied to the variable on its left and then applies that result to the parameter on its left. For instance, a value of a%=4 is the same as a value of a=a%4.

Exponentiation Assignment Operator:

In this use, **

The value on its right is multiplied by the variable on its left, and the resulting value is assigned to the variable on its left. It does this by raising the value on its left to the power of the significance on its right. For instance, the expression a **= 3 is the same as the expression a = a ** 3.

Floor Division Assignment Operator:

In this use //

It divides the variable to its left by the value to its right and then allocates the quotient to the variable to its left. This operation is known as floor division. For instance, the expression a //= 2 is the same as the expression a = a // 2.

Code:

```
A = 13  # Initialize a to 13



# Using assignment operators
a += 5   # Equivalent to a = a + 5
a -= 3   # Equivalent to a = a - 3
a *= 2   # Equivalent to a = a * 2
a %= 4   # Equivalent to a = a % 4
a **= 3  # Equivalent to a = a ** 3
a //= 2  # Equivalent to a = a // 2



print("Result:", a)  # Output will be the final value of 'a'
```

Output:

```
PS C:\Users\TOWER TECH\Desktop\Codes> & "C:/Program Files/Python311/python.exe" "c:/Users/TOWER TECH/Desktop/Codes/Chapter1_codes.py"
Result: 4
PS C:\Users\TOWER TECH\Desktop\Codes>
```

Explanation:

We set a to 13 in this code and then utilize the various assignment operators to change its value. After all these processes have been carried out, we will finally display the outcome: the value assigned to a.

Logical Operators in Python:

Conditional statements are the most common use for logical operators. AND, OR, and NOT are the three varieties of logical operators that may be used in a sentence.

Logical AND:

If the conditions of both parties are satisfied, the resultant will be deemed valid; otherwise, it will be deemed invalid.

Logical OR:

If at least one of the conditions is met, the outcome will be true; otherwise, it will be false.

Logical NOT:

Turn around the current situation.

Code:

```
# Logical AND, OR, and NOT operators
x = True
y = False



# AND operator
result_and = x and y  # False because both x and y are not
True
print(f"x AND y = {result_and}")



# OR operator
result_or = x or y    # True because x is True (at least one
condition is True)
print(f"x OR y = {result_or}")



# NOT operator
result_not_x = not x  # False because x is True, and NOT
negates it
result_not_y = not y  # True because y is False, and NOT
negates it
print(f"NOT x = {result_not_x}")
```

```
print(f"NOT y = {result_not_y}")
```

Output:

```
PS C:\Users\TOMER TECH\Desktop\Codes> & "C:/Program Files/Python311/python.exe" "c:/Users/TOMER TECH/Desktop/Codes/Chapter1_codes.py"
x AND y = False
x OR y = True
NOT x = False
NOT y = True
PS C:\Users\TOMER TECH\Desktop\Codes>
```

Explanation:

We begin by defining two Boolean variables, x and y, and then proceed to show the behavior of these variables by applying the logical operators and, or, and not to them.

2.4: Control structures (if, else, while, for)

Control structures are an important programming component because they enable you to determine the order in which your code will be executed. The following are examples of control statements in Python:

- Break statement
- Continue statement.
- Pass statement

Break statement:

Python's break statement may be used to exit a currently executed loop and remove the control from the loop. This can also be called "breaking out of" the loop. To exit a while or for loop, specifically a nested loop (also known as a loop inside a loop), this is the condition that must be met before using the break statement. The control then moves to the statement inside the outer loop as the inner loop is finished.

Code:

```
While True:
    age = input("Please enter your age: ")


    if int(age) >= 18:
        print("You are eligible to vote.")
        break
    Else:
        print("You are not eligible to vote.")
```

Output:

```
PS C:\Users\TOWER TECH\Desktop\Codes> & "C:/Program Files/Python311/python.exe" "c:/Users/TOWER TECH/Desktop/Codes/Chapter1_codes.py"
Please enter your age: 21
You are eligible to vote.
PS C:\Users\TOWER TECH\Desktop\Codes> []
```

Explanation:

We show the question using input ("Please enter your age:") and get the user's input as a string when parsed.

For age comparison, we turn the input from the user into an integer by utilizing the function int(age).

The program determines if the user is older than 18 or the same age as that threshold. If the answer is affirmative, the program publishes the message "You are eligible to vote" and then breaks out of the loop. If this is not the case, the user is informed that they do not meet the requirements to cast a vote, and the process is repeated until a legitimate age is supplied.

Continue statement:

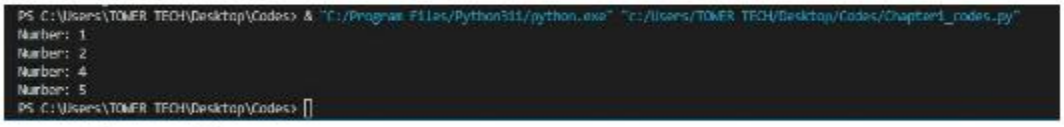
Suppose a Python program runs across a continue statement. In that case, the program will let the loop go to the next iteration even if the condition has been satisfied, even if the program will not ex-

ecute the remaining instructions for the current iteration. It allows the application to continue operating even after it has encountered a break while it is being executed, thanks to the usage of this feature.

Code:

```
for i in range(1, 6):
    if i == 3:
        continue  # Skip iteration when i is 3
    print("Number:", i)
```

Output:



Explanation:

We use range (1, 6) with a for loop to run over the numbers 1 through 5 in order. An if statement is included inside the loop that tests whether the i value equals 3. If i is equal to 3, the continue statement will be carried out, which will cause the

current iteration to be skipped. As a direct consequence of this, when i equals 3, the line that says "Number: 3" is ignored, but the loop proceeds as normal, printing the numbers 1, 2, 4, and 5.

Pass statement:

The pass statement is an example of a null operator, and it is used in situations in which the programmer does not want the computer to take any action when the condition is met. This control statement in Python does neither abort the execution nor go on to the next iteration; rather, it simply moves on to the next iteration. It is impossible to leave a loop empty since it would cause the interpreter to generate an error; a programmer may circumvent this problem by using the pass statement.

Code:

```
for i in range(1, 4):
    if i == 2:
        pass  # This does nothing
    Else:
```

```
    print("Number:", i)
```

Output:

```
PS C:\Users\TOMER TECH\Desktop\Codes> & "C:/Program Files/Python311/python.exe" "c:/Users/TOMER TECH/Desktop/Codes/Chapter1_codes.py"
Number: 1
Number: 3
PS C:\Users\TOMER TECH\Desktop\Codes>
```

Explanation:

We use range(1, 4) with a for loop to run over the numbers 1 through 3 in order.

An if statement is included inside the loop that tests whether the i value equals 2. When "i" reaches the value of 2, the pass statement is carried out. This statement has no effect and just moves on to the next iteration. It outputs "Number: i" for all other possible variable values i.

2.5: Area of Rectangle and Even/ Odd Number Exercise

Exercise 1:

Calculate the Area of a rectangle given its dimensions.

Code:

```
# As requested, Provide the rectangle's dimensions,
specifically the length, as requested.
length = float(input("Please enter the desired length for
the rectangle: "))
width = float(input("Please enter the desired width for
the rectangle: "))



# Calculate the Area of the rectangle
area = length * width



# Display the result
print(f"The area of the rectangle with length {length} and
width {width} is {area}.")
```

Output:

```
PS C:\Users\TOWER TECH\Desktop\Codes> & "C:/Program Files/Python311/python.exe" "c:/Users/TOWER TECH/Desktop/Codes/Chapter1_codes.py"
Enter the length of the rectangle: 12
Enter the width of the rectangle: 5
The area of the rectangle with length 12.0 and width 5.0 is 60.0.
PS C:\Users\TOWER TECH\Desktop\Codes>
```

Explanation:

The user provides the rectangle's length and width to the computer, and the application then uses

those values to determine the Area of the rectangle. After that, it computes the result by using the formula for the region of a rectangular shape, which is area = length * width. This formula is used to get the final value. At long last, it shows the user the Area that they have estimated. This program illustrates the simple use of Python to carry out a typical mathematical calculation. As a result, it serves as an easy and applicable example of gaining knowledge of the fundamentals of input, calculations, and production in Python programming.

Exercise 2:

Create a program that tells whether a number is even or odd.

Code:

```
# Get user input for a number
num = int(input("Enter a number: "))



# Check if the number is even or odd
```

```
if num % 2 == 0:
    print(f"{num} is an even number.")
Else:
    print(f"{num} is an odd number.")
```

Output:

Explanation:

The program commences by soliciting a numerical value from the user using the input() method. Subsequently, the int() function is employed to transform the above value. Above that, it uses the modulo operator, denoted by %, to determine whether the residual after dividing the integer by 2 equals 0. The program will display the message showing that the amount is even if the remainder is 0, and it will print a message showing that the number is odd if the remaining is anything other than 0. This program demonstrates the fundamentals of conditional logic implemented in Python and offers a

straightforward method for determining whether a number is even or odd.

Chapter 3: Functions and Forms

3.1: Creating and Using Functions

In computer programming, functions are used to group a collection of commands that you wish to utilize more than once or that, due to their complexity, are more effectively confined inside a subprogram and invoked only when required. According to this definition, a result is a code section for carrying out a certain operation. It is possible but not guaranteed that the function will need many inputs to complete that job. The function might or might not come back with one or more values depending on the state of the job once it has been completed.

In Python, there are three distinct categories of functions:

- Built-in functions such as help(), which can be used to inquire about assistance; min(), which can be used to get the minimal value; and print(), which can be used to output an object to the console.

- User-defined functions, abbreviated as UDFs, are assistance-oriented functions that individual users develop.
- Anonymous functions are sometimes called lambda functions because the conventional def keyword is not used when declaring them.

There are four distinct processes involved in the process of defining a function in the Python programming language. These steps can be enumerated as follows:

- "def" defines a function, followed by the function name.
- Incorporate parameters into the function by enclosing them between the parenthesis of the function. Please conclude your sentence with a colon.
- Include instructions for the functions to do certain tasks.
- Conclude the function by including a return statement if the desired outcome of the function is to produce an output.

Without a return statement, the function will yield a None object as its output.

Calling a function involves executing a function defined through direct action from the Python console or via another function, as seen in the "Nested Functions" section.

Code:

```
def greet():
    name = input("Enter your name: ")
    If name:
        print("Hello, " + name)
    else:
        print("Hello, World")



greet()  # Calling the greet function
```

Output:

Explanation:

The function is denoted as "greet." Within the function, user input is obtained for the name. The program verifies the presence of a specified name and a non-empty string and generates and displays a customized salutation. Without a specified name, the program outputs the "Hello, World."

3.2: Organizing code into Modules

The likelihood of a person's Python code being chaotic over time grows in proportion to the quantity of code they have written in Python. When code is stored in the same directory as the project's expansion, it might not be easy to maintain. Thanks to packages and modules, Python folders may help structure and categorize content.

Python code containing declarations and statements is in a file known as a module. These organizational tools make it easier to organize several interdependent operations, classes, or code

segments within a single file. Breaking up large sections of Python code into modules consisting of 0 and 400 words is generally recognized as a method that should be followed. Packages are used to organize and categorize modules with similar functionality into separate directories, and their primary goal is to fulfill this purpose. These directories include modules related to one another and a __init__.py script enabling optional initialization at the package level. Your Python program may need you to arrange the modules you create into sub-packages, including doc, core, utils, data, examples, and test. If this is the case, go ahead and give it a try.

The process of developing Python modules:

To successfully construct a module, saving the relevant code in a file that uses the ".py" file type is essential. After that, the function parameter of the module's name is taken on by the assigned value of the Python file.

The use of Python modules:

The import keyword will be used to include the module in our program. In addition, the from keyword allows for the selective importation of certain procedures or methods from the module. When using an outcome from a module, it is essential to remember that the correct syntax needs to be used.

Advantages of organizing code into modules:

Modules written in Python provide several advantages, the most notable being the language's increased popularity and adaptability. To facilitate improved code structure and readability, developers have the opportunity to organize code into discrete and reusable components. Using modules can encapsulate code, which helps hide implementation details and reduces the risk of name conflicts between operations, variables, and classes. Maintainability is simplified because of modularity, simplifying effective collaboration among developers by allowing them to work on separate

modules independently before integrating them without a hitch.

The extensive core library of Python, containing modules covering a wide variety of functions, contributes to expanding the language's capabilities. Modules are essential to managing different versions of a piece of code because they make it possible to precisely track any changes made to the code and aid with debugging by enclosing problematic areas of code into more manageable subsections. Python modules provide a technique for developing systematic and scalable software. This approach may be used to construct both small scripts and massive systems. Python is a programming language that benefits from increased resiliency and adaptability because of this attribute.

Code:

Module for Mathematical Operations:

math_operations.py

```
# math_operations.py
```

```
def add(num1, num2):
    return num1 + num2


def subtract(num1, num2):
    return num1 - num2


def multiply(num1, num2):
    return num1 * num2


def divide(num1, num2):
    if num2 != 0:
        return num1 / num2
    else:
        return "Cannot divide by zero."
```

Module for User Input:

input_values.py

```
# input_values.py



def get_input():
    num1 = float(input("Enter the first number: "))
    num2 = float(input("Enter the second number: "))
    return num1, num2
```

Module for Printing Results:

print_results.py

```
# print_results.py



def print_results(sum_result, difference_result, product_result, quotient_result):
    print(f"Sum: {sum_result}")
    print(f"Difference: {difference_result}")
    print(f"Product: {product_result}")
    print(f"Quotient: {quotient_result}")
```

Main Program:

Main.py

```
# main.py



from math_operations import add, subtract, multiply, di-
vide
from input_values import get_input
from print_results import print_results



# Get input values
num1, num2 = get_input()



# Perform mathematical operations
sum_result = add(num1, num2)
difference_result = subtract(num1, num2)
product_result = multiply(num1, num2)
quotient_result = divide(num1, num2)



# Print results
```

```
print_results(sum_result, difference_result, product_result, quotient_result)
```

Output:

```
PS C:\Users\TOWER TECH\Desktop\Codes> & "C:/Program Files/Python311/python.exe" "c:/Users/TOWER TECH/Desktop/Codes/main.py"
Enter the first number: 20
Enter the second number: 5
Sum: 25.0
Difference: 15.0
Product: 100.0
Quotient: 4.0
```

3.3: Importing External Modules

In addition to the libraries built by the primary Python team, many other libraries designed to expand the features of the Python programming language have been produced by third-party programmers. These extensions are not integral parts of the programming language Python, although they may be added as optional extras if necessary. These components are known as "external modules" in industry. To use an external module, it is essential to first install it on the machine being used. Obtaining the relevant files from an internet source is important to initiate the installation process. Once these files have been included in the core Python library, the programming language

can determine where the module is located. The phrase "pip" refers to a very minute, standardized unit of measurement often used in most cases; the personal setup of modules is a viable option; nevertheless, this process might be somewhat laborious. The pip module, which Python provides, is fortunate in that it offers a straightforward approach to installing modules. This capability can detect modules that are compiled within the Python Package Index, which is a collection of modules that are often used. It is possible that it will continue downloading these modules and will attempt to install them. Traditionally, programmers have carried out operations using the pip command using the terminal. Users can directly modify their computer systems by providing a command interface.

Executing the pip command:

Utilize the following syntax within the interpreter to run the pip command: pip install module-name. This technique will identify the module and the

following download and installation processes. If one module cannot be installed without the preceding installation of another module, an error known as a dependency error may result. Installing the necessary module and running the pip command is often needed to resolve the problem.

Utilizing a pre-installed software component:
After a module has been properly installed, it can write "import module-name" at the top of a Python file. This will allow the module's name to be imported into the file. The module will load just like an integrated library because of this step.

3.4: Custom Module and Calculating Factorial Exercise

Exercise 1:

Develop a computational procedure to determine the factorial of a given numerical value.

Code:

```
def factorial(n):
    """
```

```
    Determine the factorial of a positive integer 'n'.

    Args:
    n (int): The non-negative integer for which the factorial is calculated.

    Returns:
    int: The factorial of 'n'.
    """
    if n == 0:
        result = 1
    else:
        result = n * factorial(n - 1)

    print(f"The factorial of {n} is {result}")
    return result


def main():
    try:
        # Get user input for the number to calculate the factorial.
        n = int(input("Enter a non-negative integer: "))
```

```
        if n < 0:
            print("Please enter a non-negative integer.")
        else:
            result = factorial(n)
            print(f"The factorial of {n} is {result}")
    except ValueError:
        print("Invalid input. Please enter a non-negative integer.")

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

Output:

Explanation:

A docstring has been added to the code to explain the meaning of the factorial operation, its arguments and the result it returns. This helps record

the function so that it may be referred to in the future.

The logic for dealing with the user is now encapsulated behind a main() method I've added to the program. It begins by prompting the user to enter a positive number and then verifies that input before running the factorial method to compute and show the factorial.

The handling of errors has been enhanced. The application will show the user an error message and help them enter proper input if they input erroneous data (for example, a number that is not an integer or a negative number).

Exercise 2:

Create a custom module to perform mathematical operations.

Code:

```
# my_module.py
```

```python
# Define a function to multiply two numbers
def multiply_numbers(a, b):
    return a * b




# main.py



# Import the multiply_numbers function from my_mod-
ule
from my_module import multiply_numbers



# Use the function
result = multiply_numbers(5, 7)
print("The result is:", result)
```

Output:

```
PS C:\Users\TOMER TECH\Desktop\Codes> & "C:/Program Files/Python311/python.exe" "c:/Users/TOMER TECH/Desktop/Codes/main.py"
The result is: 35
PS C:\Users\TOMER TECH\Desktop\Codes>
```

Explanation:

The code that has been supplied consists of two Python files called "my_module.py" and "main.py," which illustrate the concepts of code reuse and modular programming, respectively. A Python module named "multiply_numbers" is created in the file "my_module.py," which encapsulates the method. When called, this function returns the product of its two inputs, denoted by the letters a and b. It is a reusable component that was specifically developed for the process of mathematical multiplication.

We use the modular method, and in "main.py," we do this by integrating the "multiply_numbers" function from "my_module" using the syntax "from... import..." Within the context of the present script, this grants us access to and the ability to use the functionality offered by the "multiply_numbers" function. After that, we put this imported method to use by calling it with the inputs '5' and '7,' computing the result and saving it

in the 'result' variable. In the last step, the result is shown on the display with a notification: "The result is 35."

This code is an excellent illustration of the modular programming principle, which improves code structure, readability, and reusability. Python developers can easily manage and reuse certain portions of code across various projects and scripts since functions may be encapsulated in their distinct modules. This helps to promote sustainable and efficient development software techniques.

Chapter 4: Management of Exceptions

4.1: What is an exception?

The word exception is an abbreviation for "exceptional event."

Definition:

An occurrence that happens while a program is being executed and that disturbs the regular flow of the instructions being carried out by the program is referred to as an exception.

If an error occurs while a method is being executed, the method will generate an object and then pass it over to the system's runtime. This object, which is known as an exception object, stores information on the error, such as the kind of error that occurred in the program's state at the time that the error occurred. The process of constructing an exception object and passing it on to the system's runtime is referred to as throwing an exception.

When a method generates an exception, the environment at runtime looks for a way to deal with it after it has been thrown. The sequential list of the methods that had been called before the method wherein the error occurred is the set of potential "somethings" that may be used to manage the exception. These methods were called to reach the method wherein the problem happened. The set of methods is referred to as a call stack (refer to the following representation for more information).

The runtime system will go through the call stack to find a method that may include a section of code that may deal with the exception. This section of the code is called an exception handler. The search starts with the function where the error popped up and continues across the call stack according to the sequence in which the methods have been called in the opposite direction from how it was originally called. The runtime system will send the exception to a handler after it has identified one that is suitable for the situation. If the kind

of exception object that was thrown is the same as the type that is capable of being handled by the handler, then the exception handler in question is regarded to be acceptable.

It is claimed that the exception handler that was selected "caught" the exception. Suppose the run-time system explores each method on the call stack in great detail but is unable to locate an acceptable exception handler, as illustrated in the figure that follows. In that case, the system's run-time (and, as a result, the program) will end.

4.2: Exception Handling with try-except

Try and Except in Python:

Exceptions may be handled with the try-except statement. Program exceptions may occur when you execute it.

Exceptions are mistakes that arise while the program is running. When Python encounters a syntax problem, it will terminate suddenly rather than display an error message.

It is not good for the developer or the end user to depart suddenly.

To appropriately handle the situation, you could opt for a try-except clause in place of an emergency stop. You will experience an emergency stop if you fail to handle exceptions correctly.

What are Python exceptions?

Built-in exceptions in Python can produce an error. An exception is a bug that appears while the program is running.

This kind of exception is displayed if one arises. If exceptions are not handled, the application may crash. Exceptions are handled with the try-catch block.

There are several exceptions, some of which you may be familiar with: FileNotFoundError, ZeroDivisionError, and ImportError.

Python exceptions all descended from the BaseException class. The Python interactive shell will re-

port all built-in exceptions if you open it and input the following statement:

```
>>> import builtins
>>> print(dir(builtins))
```

The try-except clause's purpose is to handle exceptions or runtime problems. This is how a try-except block is written:

```
try:
  <do something>
except exception:
  <handle the error>
```

The try-except block's concept is as follows:

- **try**: the code that contains the exception or exceptions that need to be caught. The except block is the first thing that is reached if the exception is triggered.
- **except**: This code runs solely if the try block throws an exception. Even if a try block has the pass statement, the block containing except is still necessary.

It might be paired with the terms otherwise, and lastly.

- **else**: If no exceptions have been thrown in the try block, code within the else section is solely performed.
- **finally**: regardless of whether an exception is triggered is not, the code inside the finally section will always run.

Python Exception Catching:

Exceptions may be handled with the try-except block. Thus, the software will only abruptly end when everything goes right. We deliberately create an exception in the case below.

```
try:
  1 / 0
except ZeroDivisionError:
  print('Divided by 0')


print('Should reach this point')
```

This concludes the unless block; the program goes on. The final line would not be reached since the

program would terminate in the absence of a try-except block.

```
Divided by zero
Should reach here
```

We catch the particular error ZeroDivisionError in the example above. Any exception may be handled in this way:

```
try:
  open("fantasy.txt")
except:
  print('Something went wrong')

print('Should reach here')
```

Output:

```
Something went wrong
Should reach here
```

It is possible to define distinct logic for every kind of exception that arises:

```
try:
  # your code here
except FileNotFoundError:
  # handle FileNotFoundError exception
except IsADirectoryError:
```

```
  # handle IsADirectoryError exception
except:
  # deal with all other kinds of exceptions


print('Should reach here')
```

try-except:

Let's examine an actual try-except block in action.

The program requests that users enter numbers. The user instead enters characters in the input area. Normally, the program would crash. But it may be dealt with appropriately using a try-except block.

The try-except line handles the exception appropriately and keeps the program from crashing.

```
try:
  y = float(input("Kindly enter a number: "))
  y = y + 1
  print(y)
except ValueError:
  print("Invalid input. Please enter a valid number.")
```

When incorrect data is entered, the program runs normally:

```
Kindly enter a number: one
Invalid input. Please enter a valid number.
```

Exceptions come in several forms: ZeroDivisionError, NameError, TypeError, and so on. Modules may declare exceptions on their own at times.

The try-except block is also applicable to function calls:

```
def fail():
  1 / 0


try:
  fail()
except:
  print('Exception occured')


print('Program continues')
```

This results in:

```
  Exception occured
  Program continues
```

Handling exceptions during user input:

Code:

```
try:
  user_input = input("Enter a number: ")
  result = int(user_input)
  print("You entered:", result)


except ValueError:
  print("Invalid input. Kindly enter a valid number.")


except exception as e:
  print("An error occurred:", e)
```

Output:

```
Enter a number: one
Invalid input. Please enter a valid number.
```

Explanation:

1. try: Potentially error-prone activities are contained in the code inside the try block. Here, it's attempting to carry out the following actions:

- input ("Enter a number:") = user_input: Entering a number is requested from the

user, and their input is saved as a string within the user_input variable.

- result = int(user_input): This is an effort to utilize the int() function to transform the user's input, which is saved as a string, into an integer. An exception known as ValueError may be triggered if the value being input is an invalid valid integer.

print ("You entered:", result): Should the task of converting to an integer be accomplished, the entered number is printed.

2. except ValueError: This handler is for exceptions. In particular, it detects a ValueError exception, which is thrown when an invalid integer (such as characters, symbols, or floating-point numbers) is supplied by the user and the int(user_input) conversion fails.

- This except block outputs the following message: "Invalid input. Please enter a

valid number." This alert notifies the user that the entered number is invalid.

3. except exception as e: A broader exception handler is this one. It captures any further exceptions that could happen while the try block is being executed. It catches the exception object explicitly using exception as e, enabling you to get information about the exception as required.

- This except block produces the following message that reads: "An error occurred:" and then the specifics of the exception that are kept in the variable e. As a catch-all to unforeseen issues that aren't ValueErrors, this message is sent.

In conclusion, this code manages two different kinds of exceptions and offers a convenient means of obtaining numerical input from the user:

- It notifies the user that their input is invalid if they enter anything that cannot be changed to an integer.

- If another unexpected issue arises while the try block is being executed, it notifies the user of the error and gives more information about it.

4.3: Working with Custom Exceptions

Python Custom Exceptions:

Python's custom exceptions are a powerful feature that provides programmers the ability to design their exception classes that are suited to the needs of a particular project. Error management and reporting are two very important aspects of Python programs, and these exceptions, which are sometimes referred to as programmatic or customized exceptions, play an important part in both aspects. In this in-depth investigation, we will investigate the idea of custom exceptions, their significance, as well as how to successfully construct and utilize them in Python programs.

Python, which is a programming language that is both flexible and dynamic, has many built-in exceptions that may be used to manage a variety

of mistakes that may arise while the program is being executed. However, there are certain circumstances in which these standard exceptions need to be revised to communicate the precise nature of a mistake or to provide the developer or user with useful information. Here is when the use of custom exceptions becomes relevant.

Python's basic Exception class, as well as one of its subclasses, may be extended to create what are known as "custom exceptions," which are user-defined exception classes. With the help of these classes, developers can construct exception hierarchies that are suitable for the organization and specifications of their projects. Programmers have a greater degree of control over how they classify and deal with problems because of the ability to define their unique exceptions.

Python's raise keyword is what's utilized when you want to bring up an exception on purpose. The raise statement can indicate that an error has happened in a program when an unusual circum-

stance happens, and it may also be used to propagate information about the issue to higher levels of the program or the user. When an extraordinary circumstance occurs in a program, the raise statement can be utilized. The nature of the problem, as well as the context in which it occurred, may be communicated via the use of custom exceptions, which are often thrown by a program when certain criteria that the programmer has predetermined are satisfied.

Custom exceptions provide several benefits, including the following:

1. Clarity and Readability:

The addition of custom exceptions to code helps make it more understandable and allows it to document itself. Developers and maintainers may clearly understand the nature of an error if meaningful names are given to exceptions. Examples of such names are FileNotFoundError and PermissionError.

2. Specificity:

The use of custom exceptions enables error handling at a finer granularity level. Developers can catch custom exceptions and then perform the necessary steps rather than catching a generic Exception and attempting to determine its root cause.

3. Error Handling in an Organized Manner:

Custom exceptions make it possible to create hierarchies of exceptions, which in turn enables developers to handle linked problems in an organized manner. For any problems that are connected to files, you could create a basic custom exception class and then derive additional specialized exceptions from that class.

4. Debugging and troubleshooting:

Custom exceptions provide helpful information in the context of the mistake. When an exception is thrown, it brings along with it any data or messages that were declared in the exception class.

This makes it easier to debug and figure out what went wrong.

5. Consistency:

By creating their custom exceptions, developers can guarantee that error-handling techniques remain consistent across the whole of the project. This uniformity results in a codebase that is more reliable and easier to maintain.

6. User-Friendly Error Messages:

Developers can construct user-friendly error messages with the aid of custom exceptions. These error messages provide beneficial information to end users, which makes it simpler for them to comprehend and report problems.

It is essential to take into consideration exception hierarchies if one wants to further improve the usability of custom exceptions. Python allows for the organization of user-defined exception classes into a hierarchical structure, with more exceptions inheriting from more generic exceptions. Because

of this structure, developers can detect and deal with exceptions at a variety of different degrees of granularity.

Examining some best practices for making efficient use of custom exceptions now that we have a good knowledge of the advantages and structure of custom exceptions:

1. Custom exceptions are quite useful, but it's important to remember to only use them when necessary. Reserving them for situations in which the already present Python exceptions are inadequate or in which you need to transmit special information about an issue is recommended.
2. You should use descriptive names for your custom exceptions and ensure that you use proper naming conventions, such as CamelCase. This will make your code more readable and compliant with Python's naming conventions.

3. It is important to ensure that you inherit from the correct base exception class. In most circumstances, it is advisable to derive one's exception class from an exception or a pertinent built-in exception class such as ValueError or TypeError.
4. Ensure that your custom exception classes have instructive error messages. These messages need to include some background information about the mistake and instruct users and developers on how to fix it.
5. You should document your custom exceptions just like you would any other component of your codebase. Include docstrings that describe when each exception should be thrown, why it should be raised, and how it should be handled.
6. Whenever you throw custom exceptions, you need to make sure that your code can catch them and properly treat them. This helps avoid crashes that were not pre-

dicted and delivers a seamless experience for the user.

7. If the complexity of a hierarchy of custom exceptions is required for your project, be sure to think it out well. Maintain a reasonable and straightforward structure and document the hierarchy to help other developers who are working on the project.
8. Create unit tests to ensure that your custom exceptions are thrown and dealt with appropriately. This aids in the early detection of faults throughout the development process.
9. Set up logging methods so that any exceptions and the specifics of them are recorded. This is very necessary in order to monitor and troubleshoot programs when they are running in production settings.

In a nutshell, Python's custom exceptions are a useful feature that may improve the readability, dependability, and maintainability of your program's source code. They make it possible for you to describe precise error circumstances, provide relevant error messages, and efficiently arrange your error-handling logic. You may create Python programs that are simpler to develop and maintain by following best practices and utilizing custom exceptions sparingly. These apps will be more robust and user-friendly as a result.

Steps for creating a Python Custom Exception:

We need to make a class that handles our exception initially. The programmer is expected to design their exception as a class, as all exceptions are classes. The generated class must be a child of the "Exception" class that is already there.

Syntax:

```
class MyException(Exception):
  def __init__(self,arg):
    self.arg=arg
```

Raise the exception, when necessary, in the second step. Programmers may use the raise keyword to raise their exceptions when they think there might be one.

Syntax:

```
raise MyException("message")
```

Program: Using Python to Create a Custom Exception

```
class NegativeError(Exception):
  def __init__(self, negdata):
    self.data = negdata


try:
  x = int(input("Enter a number within the positive integer range: "))
  if x < 0:
    raise NegativeError(x)
except NegativeError as e:
  print("You provided {}. Please provide positive integers only".format(e))
```

Output:

```
Enter a number between positive integer: -3
You provided -3. Please provide positive integer values only
```

Program: Python Custom Exceptions

```
class TooYoungException(Exception):
  def __init__(self, youngarg):
    self.msg = youngarg


class TooOldException(Exception):
  def __init__(self, oldarg):
    self.msg = oldarg


def check_marriage_eligibility(age):
  if age > 60:
    raise TooOldException("You are too old to get married.
There's no possibility of getting married.")
  elif age < 18:
    raise TooYoungException("Please wait a bit longer; you
will soon find the perfect match!")
  else:
        print("You'll shortly receive an email with match
information!")


try:
  age = int(input("Enter your age: "))
  check_marriage_eligibility(age)
```

```
except ValueError:
  print("Invalid input. Please enter a valid age.")
except (TooYoungException, TooOldException) as e:
  print(e)
```

Output:

```
Enter your age: 78
You are too old to get married. There's no possibility of getting married.
```

Creating custom exceptions:

Code:

```
class SalaryNotInRangeError(Exception):
    """ exception raised for the errors within the input salary.

  Attributes:
    salary -- input salary, which caused the error
    message -- explanation of the error
  """

    def __init__(self, rsalary, rmessage="Salary's not in the(5000, 15000) range"):
    self.salary = rsalary
    self.message = rmessage
    super().__init__(self.message)
```

```
salary = int(input("Kindly enter salary amount: "))
if not 5000 < salary < 15000:
    raise SalaryNotInRangeError(salary)
```

Output:

```
Enter salary amount: 2000
Traceback (most recent call last):
  File "c:\Users\Areej\OneDrive\Desktop\Book.py", line 17, in <module>
    raise SalaryNotInRangeError(salary)
SalaryNotInRangeError: Salary is not in (5000, 15000) range
```

Explanation:

1. class SalaryNotInRangeError(Exception):
   - This line creates an entirely novel exception class named SalaryNotInRangeError, which is derived from the Exception class that comes with Python.
   - It has a docstring that describes the characteristics and intent of the exception.
2. def __init__(self, salary, message="Salary is not in (5000, 15000) range"):
   - This is the SalaryNotInRangeError class constructor function. Two characteristics are initialized when the exception object is created:

- Salary: The wage amount that resulted in the mistake.
- Message: An optional statement explaining the mistake. The notification that appears by default is "Salary is not in (5000, 15000) range."
- In order to set the error message, it further invokes the constructor of the underlying Exception class using super().__init__(self.message).

3. salary is equal to int (input ("Enter salary amount: ")) *
- This line asks the user to input their wage, which is then saved inside the salary variable as an integer.

4. If not, salary range < 5,000 < 15000:
- This line determines if the salary entered falls outside of the designated range of $5,000 to $15,000.
- The subsequent code is run if the salary falls outside of this range:

5. raise NotInRangeSalaryError(salary);

- With the salary value given as a parameter, this line produces an instance for the SalaryNotInRangeError exception.
- Since we failed to include a custom message in this particular case, the exception object is created using the specified salary amount and the default error message ("Salary is not within the (5000, 15000) range") is displayed.

In short, when a user submits a salary that is outside of the designated range, this code enables you to throw a custom exception (SalaryNotInRangeError). This might be helpful in some situations when salary data needs to be verified, and if the validation is unsuccessful, it clearly displays an error notice.

4.4: Exercises

Exercise 1:

Exception handling during division by zero.

Code:

```
try:
    numerator = int(input("Enter the numerator: "))
    den = int(input("Kindly enter the denominator: "))

    output = numerator/den

except ZeroDivisionError:
    # Handle the ZeroDivisionError exception
    print("Error: Division by 0 is not permitted.")
except ValueError:
    # Handle the ValueError exception (invalid input)
     print("Error: Please enter valid integer values for numerator and denominator.")
except exception as e:
    # Handle any other exceptions that may occur
    print(f"An error occurred: {e}")

# Code continues here after handling exceptions
if 'result' in locals():
    print(f"Result of division: {result}")
```

Output:

```
Enter the numerator: 3
Kindly enter the denominator: 0
Error: Division by 0 is not permitted.
```

Explanation:

1. try: Operations that are likely prone to error are put in the code that is included inside the try block. In this instance, it is attempting to carry out the processes as follows:
   - It requests input from the user for both the numerator and the denominator, and it expects integer values.
   - Within the confines of the try block, it tries to carry out the division function (numerator/denominator).
2. except ZeroDivisionError: This except block is responsible for catching the ZeroDivisionError exception, which is generated when the denominator is equal to zero.
   - it sends out the following error message: "Error: Division by zero is not allowed."
3. except ValueError: This except block is responsible for catching the ValueError exception that may be thrown if the user

provides improper input (for example, values that are not integers).

- It will print out an error message that says, "Error: Please enter valid integer values for numerator and denominator."

4. except exception as e: This is a handler for broader exceptions. It is responsible for catching any further exceptions that may arise while the try block is being executed.
   - It generates an error message that includes more information regarding the exception (e).
5. The code proceeds to verify whether the variable solution is declared within the local scope employing the statement if 'result' is present in locals (): after it has finished handling any exceptions that may have been thrown.
6. Suppose the value of the result is checked, and it is found to be defined. In that case, this indicates that the division has

finished successfully and, therefore, the variable outcome has an accurate result.

- It reports the outcome of the division by writing f"Result of division: {result}" in the output.

You may organize your code in this manner; doing so allows you to handle exceptions involving division by zero as well as other possible mistakes. The code will only continue to run if there is a legitimate result already available, and it will report the result whenever it is appropriate to do so.

Chapter 5: Working with Lists and Dictionaries

5.1: Introduction to Lists and their Manipulation

Defining a List:

Lists are enclosed inside the square brackets [ ].

```
# Defining a list
z = [3, 7, 4, 2]
```

Lists are collections of objects, sometimes of multiple kinds, that are kept in order. As you may observe below, only some items in a list need to have the same type. The list created above has elements that all appear to have identical types (int).

```
# Defining a list
heterogenousElements = [3, True, 'Michael', 2.0]
```

An integer, boolean, string, and float are included in the list.

Accessing List Values:

Each of the items in a list is designated a unique index value. It is essential to remember the fact that Python constitutes a zero-indexed language. This merely shows that the very first item within the list is located at index 0.

```
# Define a list
z = [3, 7, 4, 2]# Access a list's initial entry at index 0.
print(z[0])
```

Output:

```
3
```

Python supports negative indexing as well. Negative indexing commences at the end. It can be more practical at times to employ negative indexing to obtain the last item from the list since you don't have to know the extent of the list in order to locate the last item.

```
# Define a list
z = [3, 7, 4, 2]
# display the final item within the list
print(z[-1])
```

Output:

```
2
```

Remind yourself that positive indexes may also be used to reach the same object (as seen below).

```
# Define a list
z = [3, 7, 4, 2]
# display the final item within the list
print(z[3])
```

Output:

```
2
```

List Slices:

You may get a subset of the values within your list by using slices. The code sample below will provide a list containing the elements from index 0 to index 2, except index 2.

```
# Define a list
z = [3, 7, 4, 2]
print(z[0:2])
```

Output:

```
[3, 7]
```

```
# Define a list
z = [3, 7, 4, 2]
# all of the mentioned items except index 3
print(z[:3])
```

Output:

```
[3, 7, 4]
```

A list containing items starting from the first index to the last item of the list is returned by the code below:

```
# Define a list
z = [3, 7, 4, 2]
# index 1 to end of list
print(z[1:])
```

Output:

```
[7, 4, 2]
```

Update an item in the list:

In Python, lists may be changed. That indicates that each item in a list may be updated after it has been defined.

```
# Defining a list
z = [3, 7, 4, 2]# Replace the item at index 1 with the string
"fish"
z[1] = "fish"
print(z)
```

Output:

```
[3, 'fish', 4, 2]
```

List Techniques:

Several techniques in Python lists enable you to alter a list. This portion of the course covers several list techniques in Python.

Index Method:

```
# Defining a list
z = [4, 5, 1, 4, 10, 4]
```

The initial index in which a value appears is returned by the index method. The code that follows will return 0.

```
# Define a list
z = [4, 5, 1, 4, 10, 4]
print(z.index(4))
```

Output:

```
0
```

Also, you have the option to specify to start your search.

```
# Define a list
z = [4, 1, 5, 4, 10, 4]
print(z.index(4, 3))
```

Output:

```
3
```

Count Method:

The count method does just as its name suggests. It keeps track of how often a value appears in a list.

```
random_list = [4, 5, 1, 4, 10, 4]
random_list.count(5)
```

Sort Method:

The initial list is sorted and modified using the sort technique.

```
z = [3, 7, 4, 2]
z.sort()
print(z)
```

Output:

```
[2, 3, 4, 7]
```

A list is sorted from low to high using the code above. You may also sort the list from high to low, as seen by the code below:

```
z = [3, 7, 4, 2]
# Sorting and Altering the original list
# high to low
z.sort(reverse = True)
print(z)
```

Output:

```
[7, 4, 3, 2]
```

Append Method:

An element is placed at the end of the list using the append function. This takes place on-site.

```
z = [7, 4, 3, 2]
z.append(3)
print(z)
```

Output:

```
[7, 4, 3, 2, 3]
```

Remove Method:

The remove method eliminates the first instance of a given value from a list.

```
z = [7, 4, 3, 2, 3]
z.remove(2)
print(z)
```

Output:

```
[7, 4, 3, 3]
```

Pop Method:

The item at the index specified is removed using the pop technique. The item you deleted from the

list will also be returned using this method. By default, the item in the last index will be removed if you don't specify an index.

```
z = [7, 4, 3, 3]
print(z.pop(1))
print(z)
```

Output:

```
4
[7, 3, 3]
```

Extend Method:

The approach adds items to a list to make it longer. One advantage of this is that lists may be added together.

```
z = [7, 3, 3]
z.extend([4,5])
print(z)
```

Output:

```
[7, 3, 3, 4, 5]
```

Insert Method:

An item is inserted before the supplied index when using the insert technique.

```
z = [7, 3, 3, 4, 5]
```

```
z.insert(4, [1, 2])
print(z)
```

Output:

```
[7, 3, 3, 4, [1, 2], 5]
```

Advantages of Lists:

1. Lists are examples of ordered collections of items. This indicates that the components in a list are arranged in a certain order, and you can access them in the list by referring to their position or index number. This ordering makes it possible for you to organize sequences of data systematically, such as a list of numbers, names, or dates.
2. The versatility of Python's lists lies in the fact that they are not constrained to only hold components of a single data type. Within a single list, you can combine different data types such as integers, floats, texts, and even additional data structures such as lists and dictionaries. Because of their adaptability, lists are a

useful option for a wide variety of data storage requirements.

3. Lists are changeable, which implies that their contents may be altered after they have been first created. You are free to add to, change, or delete items from a list as the situation demands. This mutability is particularly helpful in situations in which you need to dynamically update or alter data while the program is being executed.
4. Lists in Python can dynamically increase or decrease in size when items are added or removed. This feature is referred to as dynamic sizing. There is no need you declare their size in advance, which makes it simple to deal with collections that have different lengths.
5. Lists provide effective methods to access items by their index and are used to organize the elements. This enables you to access items quickly by utilizing their location inside the list. The zero-based in-

dexing that Python uses enables access to items that are both simple and obvious.

6. Lists are iterable objects, which means you can cycle over their members using loops or other iteration techniques. This is because lists are iterable. Because of this, it is simple to carry out actions on each member in the list without the need to manually maintain the indices.
7. Lists support a broad variety of operations and procedures, making them useful for a variety of everyday activities. If you want to edit a list, you may use methods such as append, insert, delete, pop, and extend. If you want to access information about the list, you can use functions such as len, min, max, and sum.
8. Python has list comprehensions, which are a method that is both concise and expressive for creating lists based on other iterable data or on lists that already exist. Using a single line of code to conduct op-

erations such as transformations, filtering, or computations on list components is made simple by using list comprehensions.

9. Lists are an excellent method for storing data that must be retrieved as well as changed in a short amount of time. For example, lists are often used as a means of storing data obtained through databases, human input, or files. Using list operations, you will have a simple time retrieving and manipulating this data.
10. That Are Flexible Lists may be used to construct more complicated data structures such as stacks, queues, linked lists, and a variety of other data structures. Due to their versatility, lists are an essential component in the construction of a wide variety of different data structures and algorithms.
11. Lists, in conjunction with libraries like NumPy and Pandas, are used

to a great extent in the performance of activities related to data analysis and manipulation. Because they provide a straightforward approach to the storage and processing of data sets, lists are indispensable in the fields of data science and analytics.

12. Because they are simple to work with, lists are appropriate for programmers of all expertise levels, including novices. The syntax that Python uses for building lists and interacting with them is easy to understand and clear.

13. Python's rich ecosystem consists of many modules and frameworks, and one of the key data structures that Python makes use of is a list. Learning how to manipulate lists in Python enables the use of these libraries for a wide range of purposes, such as web development, scientific computation, and machine learning.

In conclusion, lists in Python provide a broad variety of benefits, some of which include their ordered nature, adaptability, mutability, dynamic scaling, simplicity of access and indexing, support for common operations, compatibility with iteration and loops, and their role in the implementation of more complicated data structures. When it comes to organizing and processing data in Python, lists are a very helpful tool for anybody, regardless of their level of programming expertise. To effectively program using the Python language, it is essential to have a solid understanding of their benefits and capabilities.

5.2: Use of Dictionaries in Python

A dictionary in Python is a collection that lets us keep information in key-value pairs.

```
# Replace ___ with your code


# create variable number1 with value 9
number1 = 9
```

```
# create variable number2 with value 5
number2 = 5


# add number1 and number2
result = number1 ___ number2


# print result
print(result)
```

Output:

```
  File "c:\Users\Areej\OneDrive\Desktop\Book.py", line 10
    result = number1 ___ number2
                     ^^^
SyntaxError: invalid syntax
```

Creating a Dictionary:

By putting key: value pairs within curly brackets {} and separating them with commas, we can form dictionaries. For instance,

```
# creating a dictionary
country_capitals = {
 "United States": "Washington D.C.",
 "England": "London"
 "Italy": "Rome"
}
```

```
# printing dictionary
print(country_capitals)
```

Output:

```
{'United States': 'Washington D.C.', 'Italy': 'Rome', 'England': 'London'}
```

Key-value pairs make up the three components of the country_capitals dictionary.

Tuples, strings, integers, and other immutable data types cannot be used as dictionary keys. Mutable (changeable) objects, like lists, cannot be used as keys.

```
# Valid dictionary example

my_dict = {
  1: "Hello",
  (1, 2): "Hello Hi",
  3: [1, 2, 3]
}

print(my_dict)

# Invalid dictionary example
# Error: using a list as a key is not allowed
```

```
my_dict = {
 1: "Hello",
 [1, 2]: "Hello Hi",
}

print(my_dict)
```

Output:

```
{1: 'Hello', (1, 2): 'Hello Hi', 3: [1, 2, 3]}
Traceback (most recent call last):
  File "c:\Users\Areej\OneDrive\Desktop\Book.py", line 14, in <module>
    my_dict = {
              ^
TypeError: unhashable type: 'list'
```

Length of Python Dictionary:

With the help of the len() method, we can determine a dictionary's size.

```
country_capitals = {
 "United States": "Washington D.C.",
 "England": "London"
 "Italy": "Rome"
}

# get the dictionary's length
print(len(country_capitals)) # 3
```

Output:

```
3
```

Length of Python Dictionary:

With the help of the len() method, we can determine a dictionary's size.

```
country_capitals = {
 "United States": "Washington D.C.",
 "Italy": "Rome",
 "England": "London"
}


# get the dictionary's length
print(len(country_capitals)) # 3
```

Output:

```
3
```

Accessing Dictionary Items:

If we enclose the key in square brackets, we can get the value of the dictionary entry.

```
country_capitals = {
 "United States": "Washington D.C.",
 "Italy": "Rome",
 "England": "London"
}
```

```
print(country_capitals["United States"])

print(country_capitals["England"]) # London
```

Output:

```
Washington D.C.
London
```

The get() function may also be used to get dictionary entries.

Modify Dictionary Items:

Mutable dictionaries are available for Python. A dictionary element's value may be altered by consulting its key. For instance,

```
country_capitals = {
 "United States": "Washington D.C.",
 "England": "London"
 "Italy": "Naples"
}

# changing value of "Italy" key to "Rome"
country_capitals["Italy"] = "Rome"
```

```
print(country_capitals)
```

Output:

```
{'United States': 'Washington D.C.', 'Italy': 'Rome', 'England': 'London'}
```

Add Items within a Dictionary:

By giving a new key—one that doesn't already exist in the dictionary—a value, we can add a new entry to the dictionary. For instance,

```
country_capitals = {
 "United States": "Washington D.C.",
 "Italy": "Naples"
}


# with "Berlin" as the value and "Germany" as the key
country_capitals["Germany"] = "Berlin"


print(country_capitals)
```

Output:

```
{'United States': 'Washington D.C.', 'Italy': 'Naples', 'Germany': 'Berlin'}
```

The update method() may also be used to add or modify dictionary entries.

Removing the Dictionary Items:

To remove an item from the dictionary, we use the del statement. For instance,

```
country_capitals = {
 "United States": "Washington D.C.",
 "Italy": "Naples"
}


# taking out the item with the "United States" key
del country_capitals["United States"]
```

```
print(country_capitals)
```

Output:

```
{'Italy': 'Naples'}
```

To take anything out of the dictionary, we can also employ the pop method().

The clear() function may be used to delete every entry from a dictionary at once.

```
country_capitals = {
 "United States": "Washington D.C.",
 "Italy": "Naples"
}
```

```
country_capitals.clear()

print(country_capitals) # {}
```

Output:

```
{}
```

Test of Dictionary Membership

Using the in operator, we can determine whether a key is present in a dictionary.

```
my_list = {1: "Hello", "Howdy": 100, "Hi": 25}

print(1 in my_list) # True

# the not in operator checks whether the key doesn't exist
print("Howdy" not in my_list)

print("Hello" in my_list) # False
```

Output:

```
True
False
False
```

The in-operator does not determine if a value exists; rather, it determines whether a key exists.

Going Over a Dictionary Iteratively:

As of Python 3.7, a dictionary constitutes an ordered set of things. In other words, a dictionary keeps its elements in the same sequence.

Using a for loop, we can cycle over dictionary keys sequentially.

```
country_capitals = {
 "United States": "Washington D.C.",
 "Italy": "Naples"
}


# print each dictionary entry individually
for country in country_capitals:
  print(country)


print("----------")


# print each dictionary entry individually
for country in country_capitals:
  capital = country_capitals[country]
  print(capital)
```

Output:

```
United States
Italy
----------
Washington D.C.
Naples
```

Advantages of Dictionaries:

1. Dictionary entries are stored in a hash table, which makes it possible for data to be retrieved very quickly depending on the keys that are associated with them. This indicates that seeking up an item by its associated key is normally an activity that takes a consistent amount of time, regardless of how large the dictionary may be.
2. Dictionary entries may hold data in the form of key-value pairs thanks to the dictionary's Key-Value Association feature. When attempting to depict connections between different bits of data, this key-value association is of tremendous assistance. Because it is so simple to seek up values based on the keys that relate to them, dictionaries are excellent tools for data retrieval and mapping.

3. Dictionaries are not ordered by nature; nevertheless, beginning with Python version 3.7, dictionaries keep insertion order. This implies that entries are saved in the order in which they were added to the dictionary. This makes dictionaries more efficient. Python versions 3.6 and previous do not ensure order; however, Python versions 3.7 and beyond do. Dictionary lookups continue to be quite effective regardless of the sequence in which the entries are presented.
4. Keys in a dictionary are not restricted to being strings; rather, they may be of a wide variety of data types, such as strings, integers, floats, and even more complicated structures, such as tuples. Because of this versatility, you are able to construct dictionaries that are specifically catered to your requirements.
5. Each key in a dictionary has to be unique. When you add a new key-value pair to an

existing database using the same key, the fact that each pair is unique assures that you won't inadvertently replace an existing value. It ensures the correctness and completeness of the data.

6. In the same way that lists may expand or contract dynamically, dictionaries can do the same. Because it is optional to declare their size in advance, they are appropriate for circumstances in which the amount of your data is either unknown or changeable.
7. Dictionary use is great for circumstances in which you need to associate facts quickly and effectively. Dictionaries are suitable for these kinds of situations. Dictionary entries may, for instance, be used to represent data caches, configuration settings, or data that has been derived from files or databases.
8. Python dictionaries come prepackaged with a broad variety of built-in meth-

ods that facilitate the performance of frequently used tasks. When you want to handle dictionary data without having to write a lot of code, you may utilize methods such as get(), keys(), values(), items(), pop(), and update(), among many others.

9. Python provides dictionary comprehensions, which are a condensed method for creating dictionaries from iterables or other dictionaries. Utilizing this feature, you will be able to construct new dictionaries by applying transformations, filters, or computations to the data that is already there.

10. Dictionary usage is common in contexts where doing data lookups is a regular activity because dictionaries facilitate efficient lookups. For instance, dictionaries are used in online applications for the purpose of swiftly looking up user sessions or routing information.

11. You are able to design sophisticated data structures by nesting dictionaries inside other dictionaries, which gives you the ability to create hierarchical data structures. When working with hierarchical data or developing complex data models, having this capacity is really helpful.

12. Python's rich ecosystem, which consists of a large number of libraries and frameworks, makes heavy use of dictionaries. For instance, dictionaries play a large part in the serialization and deserialization of JSON (JavaScript Object Notation) data, as well as in the interactions between configuration files and databases.

13. Dictionaries may be helpful when converting data from one format to another. They enable the creation of mappings, the translation of data between various representations, and the

facilitation of activities relating to data integration.

In conclusion, Python dictionaries provide a number of benefits, such as rapid retrieval, key-value association, flexibility in key data types, uniqueness of keys, dynamic size, and a comprehensive collection of built-in functions. They are an excellent choice for data search, data representation, and data transformation activities in a variety of contexts, such as web development, data analysis, and general-purpose programming, amongst others. Your capacity to interact with data in Python and to construct programs that are both efficient and resilient may be significantly improved by developing an understanding of dictionaries and making proper use of them.

5.3: Exercises

Exercise 1:

Extracting random elements from a list:

Code:

```
import random

def extract_random_elements(input_list, num_elements):
    """
    Extracts random elements from a list.

    Args:
        input_list (list): The list from which to extract random elements.
        num_elements (int): The number of random elements to extract.

    Returns:
        list: A list containing the randomly extracted elements.
    """
    if num_elements > len(input_list):
        raise ValueError("There are more components to extract than there are in the input list.")

    # Use random.sample to generate a list of unique random indices
```

```
    random_indices = random.sample(range(len(in-
put_list)), num_elements)

  # Extract elements at the random indices
    random_elements = [input_list[i] for i in ran-
dom_indices]

  return random_elements

# Example usage:
List1 = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
num_to_extract = 3

random_elements = extract_random_elements(my_list,
num_to_extract)
print("Randomly extracted elements:", random_ele-
ments)
```

Output:

```
Randomly extracted elements: [4, 2, 9]
```

Explanation:

1. The very first thing we do is import the random module, which enables us to

work with arbitrary integers and is responsible for generating random numbers.

2. The input_list and num_elements arguments are necessities for the function that is known as extract_random_elements. The num_elements parameter determines the number of random elements that are to be extracted from the list, and the input_list argument describes the list from which the random elements are to be taken.
3. We carry out a check to make certain that the amount of num_elements does not exceed more than the entire number of items that are included in the input list. If this is the case, we will issue a ValueError because there is a maximum number of elements that may be removed from the list. The size of the list determines this limit.

4. We make use of the method called random.sample(range(len(input_list)), num_elements), which allows us to generate a list of random indices that are unique to one another. random.sample verifies that each of the indices is unique and does not show up more than once. It achieves this by preventing duplicates.
5. After that, we make use of the list comprehension to extract the components from the input_list that correspond to the random indices that we gathered in the step before this one. This step comes after step 4, "Collecting Random Indices."
6. At the conclusion of this process, we will provide a return of the list of components that were randomly extracted.
7. To begin the example usage, we first generate a sample list that we refer to as my_list and indicate that we want to choose three members at random from the list. After inserting these inputs into

the extract_random_elements function that we called, we then report the result.

This piece of code will construct the following list for you, which will consist of a predefined number of random elements selected from your input list. The number of these entries will be decided in advance. Because there won't be any replacements, the components will be chosen at random, and that implies that each component will only ever be offered for selection once.

Exercise 2:

Handling a list of contacts with a dictionary.

Code:

```
# Initialize an empty dictionary to store contacts
contacts = {}


def add_contact(cname, cphone, cemail):
    """
    Add a contact to the dictionary.

    Args:
```

```
    cname (str): The name of the contact.
    cphone (str): Contact's phone number.
    cemail (str): Contact's email address.
    """
    contacts[cname] = {'phone': cphone, 'email': cemail}

def get_contact(cname):
    """
    Retrieve contact information by name.

    Args:
        name (str): The name of the contact to retrieve.

    Returns:
        dict: A dictionary containing the contact's phone number and email address.
    """
    return contacts.get(cname, "Contact not found")

def list_contacts():
    """
    List all contacts in the dictionary.
    """
```

```
    for name info in contacts.items():
        print("Name:", cname)
        print("Phone:", info['phone'])
        print("Email:", info['email'])
        print("---")

# Add contacts
add_contact("Alice", "123-456-7890", "alice@example.com")
add_contact("Bob", "987-654-3210", "bob@example.com")

# Retrieve and print contact information
contact_name = "Alice"
contact_info = get_contact(contact_name)
if contact_info != "Contact not found":
    print(f"Contact information for {contact_name}:")
    print("Phone:", contact_info['phone'])
    print("Email:", contact_info['email'])
else:
    print(f"{contact_name} not found in contacts.")

# List all contacts
print("\nList of all contacts:")
```

```
list_contacts()
```

Output:

```
Contact information for Alice:
Phone: 123-456-7890
Email: alice@example.com

List of all contacts:
Name: Alice
Phone: 123-456-7890
Email: alice@example.com
---
Name: Bob
Phone: 987-654-3210
Email: bob@example.com
---
```

Explanation:

1. To begin, a blank dictionary with the names of contacts that will be used to hold contact information is initialized. Each contact will be saved as the key-value pair, with the name of the contact serving as the key and another dictionary holding the person's email address and telephone number serving as the value.
2. We will define the following three functions:
   - add_contact is a function that, when called, inserts a contact into the contacts dictionary by requiring the user to

provide the contact's name, phone number, and email address.

- The name given to the contact is required for the get_contact function in order to get contact information. It will provide a dictionary containing the contact's phone number & email address, or it will return a message in case the contact cannot be located.
- list_contacts is a function that, given a set of keys, will iterate over those keys and print out the contact information for each contact in the contacts' dictionary.

3. Using the add_contact method, we add 2 contacts to the list: "Alice" and "Bob."
4. Using the get_contact method, we obtain the contact information of "Alice" and then print it.
5. Finally, we show the data of each contact that has been saved in the dictionary

by listing all the contacts that have been saved using the list_contacts method.

The following line of code explains how to effectively store and maintain the list of contacts by making use of a dictionary. You can effortlessly add new contacts, recover previously saved contacts, and list contacts based on name, phone number, or email address.

Chapter 6: Advanced File and Data Management

Python offers a rich array of built-in functions for creating, composing, and perusing files. Within the Python realm, two primary file types are encountered: regular text files and binary files, the latter being encoded in binary language, comprised solely of 0s and 1s.

In the realm of text files, each line culminates with a distinct character, termed EOL (End of Line). In Python, the default EOL character is the newline character ('\n'). On the other hand, binary files lack such line terminators, storing data in a machine-readable binary format.

This discourse focuses exclusively on the art of handling text files, from their inception to closure, encompassing the realms of reading and writing data.

6.1: File Access Modes

Access modes play a pivotal role in shaping the

nature of operations permissible within an opened file. These modes dictate how a file is intended to be utilized once it is accessed and define the positioning of the File Handle within the file—a cursor-like entity that indicates the location for reading or writing operations. Python offers six distinct access modes.

1. Read Only ('r'): This mode opens a text file purely for reading. The file handle is poised at the outset of the file. If the file doesn't exist, an I/O error is raised. This represents the default mode for file opening.
2. Read and Write ('r+'): In this mode, the file is opened for both reading and writing, with the handle positioned at the beginning of the file. An I/O error is raised if the file is missing.
3. Write Only ('w'): When a file is opened in this mode, it is specifically for writing purposes. If an existing file is opened, its

contents are truncated, and the new data overwrites the old. The file handle is set to the file's beginning, and the file is created if it doesn't already exist.

4. Write and Read ('w+'): Similar to 'w,' this mode permits both reading and writing. It truncates and overwrites existing data if the file exists, with the handle at the file's inception.
5. Append Only ('a'): In 'a' mode, the file is designated for writing, even creating it if absent. The file handle is located at the end of the file, and newly written data is appended after the existing content.
6. Append and Read ('a+'): This mode combines reading and writing. It opens the file for both operations, creating the file if necessary. The file handle resides at the file's end, and new data is appended after the existing data.

How Files are Loaded into Primary Memory:

In the realm of computers, two distinct forms of memory exist: Primary and Secondary memory. Any file saved, whether by you or another user, is stored in secondary memory. Primary memory, in contrast, loses its contents when the computer is powered off. To manipulate a text file in Python or effect any changes, the file must be loaded into primary memory. Python interacts with files residing in primary memory through "file handlers," which serve as conduits permitting Python to interact with the file. Essentially, the operating system searches for the file in its memory and, if found, returns a file handler, enabling you to work with the file.

Opening a File:

The process of opening a file is executed via the open() function without the need to import any external modules. The syntax for opening a file is:

```
File_object = open(r"File_Name", "Access_Mode")
```

If the file is not located in the same directory as the Python programme file, its complete address should be given in lieu of the filename. The file should exist in the same folder as the Python programme file. Note that the letter r is inserted before the filename so that every character in the filename string won't be interpreted as special characters. This prevents the filename string from being altered in any way. For instance, if the file address contains the notation temp, the notation t will be interpreted as the tab character, and an error will be thrown since the address is illegal. The r character converts the string into a raw format; more specifically, it indicates that the string does not include any characters having particular meaning. If the file is located in the same location and its address is not being inserted, the r may be disregarded as unnecessary.

```
# Open function to open the file "MyFile1.txt"
# (same directory) in append mode and
file1 = open("MyFile1.txt","a")
```

```
# store its reference in the variable file1
# and "MyFile2.txt" in D:\Text in file2
file2 = open(r"D:\Text\MyFile2.txt","w+")
```

Here, file1 is created as an object for MyFile1 and file2 as object for MyFile2

Closing a file

close() function closes the file and frees the recall space acquired by that file. It is used at the period when the file is no longer crucial or if it is to be set off in a different file mode.

```
File_object.close()


# Opening and Closing a file "MyFile.txt"
# for object name file1.
file1 = open("MyFile.txt","a")
file1.close()
```

Writing to a file

There are two ways to write in a file.

1. write() : Inserts the string str1 in a individual line in the text file.

File_object.write(str1)

2. writelines() : For a list of string basics, each string is appended in the text file.Used to insert several strings at a single time.

File_object.writelines(L) for L = [str1, str2, str3]

Reading from a file

To read data from a text file:

1. read() : The bytes that have been read are returned in string format. Reads n bytes; if n is not supplied, the file's whole is read instead.

File_object.read([n])

2. readline() : Reads a line of the file and returns in the form of a string. For specified n, reads at most n bytes. But can not read more than a line.

File_object.readline([n])

3. readlines() : Look through all the lines and yields them as each line is a string portion in a list.

File_object.readlines()

Note: '\n' is treated as a special character of two bytes

Exercise 1:

Create a program to show various ways to read and write data in a file.

Code:

```
file1 = open("myfile.txt","w")
L = ["This is Delhi \n","This is Paris \n","This is London \n"]


# \n is placed to indicate EOL (End of Line)
file1.write("Hello \n")
file1.writelines(L)
file1.close() #to change file access modes

file1 = open("myfile.txt","r+")


print("Output of Read function is ")
print(file1.read())
print()
```

```
# seek(n) takes the file handle to the nth
# byte from the beginning.
file1.seek(0)

print( "Output of Readline function is ")
print(file1.readline())
print()

file1.seek(0)

# To show difference between read and readline
print("Output of Read(9) function is ")
print(file1.read(9))
print()

file1.seek(0)

print("Output of Readline(9) function is ")
print(file1.readline(9))

file1.seek(0)
# readlines function
```

```
print("Output of Readlines function is ")
print(file1.readlines())
print()
file1.close()
```

Output:

Myfile.txt:

```
Hello
This is Delhi
This is Paris
This is London
```

Explanation:

- The code starts by opening a file named "myfile.txt" in write mode ('w') using file1 = open("myfile.txt","w").
- It defines a list L containing three strings, each representing a line of text with the "This is..." format, and each line ends with a newline character ('\n').
- The code writes "Hello" followed by a newline character to the file using file1.write("Hello \n").

- Then, it uses the writelines() method to write the list L to the file. This appends the three lines to the existing content.
- Finally, the file is closed using file1.close()
- Next, the code opens the same file for reading and writing in 'r+' mode with file1 = open("myfile.txt","r+").
- It uses file1.read() to read the entire content of the file and print it out. This displays the entire content of the file.
- Then, it uses file1.seek(0) to set the file position back to the beginning (offset 0).
- Following the seek(0) operation, it reads a single line from the file using file1.readline(). This reads and returns the first line of the file, which is "Hello " (including the newline character).
- The code seeks the beginning of the file again with file1.seek(0).
- It uses file1.read(9) to read the next 9 characters from the file, including spa-

ces and newline characters. The output is "Hello \nTh".

- The code seeks the beginning of the file once more with file1.seek(0).
- Finally, it reads the first 9 characters of the current line using file1.readline(9). The output is "Hello ".

In summary, this Python program demonstrates various methods of reading and writing data in a file. It writes multiple lines of text to "myfile.txt," reads the entire content, reads a single line, and showcases the difference between reading characters using read() and readline().

Appending to a File:

Appending data to a file can be accomplished using 'a' mode, which adds new content to the end of the file without overwriting existing data. Conversely, 'w' mode overwrites the existing content with new data.

```
# Python program to illustrate Append vs write mode
file1 = open("myfile.txt", "w")
```

```
L = ["This is Delhi \n", "This is Paris \n", "This is London
\n"]
file1.writelines(L)
file1.close()

# Append-adds at last
file1 = open("myfile.txt", "a") # append mode
file1.write("Today \n")
file1.close()

file1 = open("myfile.txt", "r")
print("Output of Readlines after appending")
print(file1.readlines())
print()
file1.close()

# Write-Overwrites
file1 = open("myfile.txt", "w") # write mode
file1.write("Tomorrow \n")
file1.close()

file1 = open("myfile.txt", "r")
print("Output of Readlines after writing")
```

```
print(file1.readlines())
print()
file1.close()
```

In summary, Python offers robust file-handling capabilities, facilitating a range of operations on text files. Understanding file access modes, loading files into primary memory, and mastering the methods for opening, closing, writing to, and reading files are essential skills for any Python programmer. This proficiency empowers programmers to effectively manage and manipulate data stored in text files, a fundamental aspect of many real-world applications.

6.2: Understanding XML and JSON in Python

XML and JSON are two commonly used data formats in the world of programming, particularly for storing and exchanging data. They provide a structured way to represent information that can be easily understood by both humans and ma-

chines. In Python, you can work with XML and JSON data efficiently using libraries designed for this purpose.

XML (eXtensible Markup Language)

XML is a markup language, which means it uses tags to define the structure of data. These tags help organize and describe the information contained within the document. Here's a basic example of XML:

Code:

```
<book>
  <title>Python Programming</title>
  <author>John Doe</author>
  <price>29.99</price>
</book>
```

Explanation:

In this XML, <book> is the root element, and it contains child elements like <title>, <author>, and <price>, each storing specific information.

Reading and Writing XML in Python

Python provides the xml.etree.ElementTree library for working with XML data. You can use it to parse (read) and generate (write) XML files.

Reading XML:

```
import xml.etree.ElementTree as ET


# Parse an XML file
tree = ET.parse("my_file.xml")
root = tree.getroot()


# Accessing elements in XML
title = root.find("title").text
author = root.find("author").text
price = root.find("price").text


print("Title:", title)
print("Author:", author)
print("Price:", price)
```

Writing XML:

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

```
# Creating an XML structure
book = ET.Element("book")
title = ET.SubElement(book, "title")
title.text = "Python Programming"
author = ET.SubElement(book, "author")
author.text = "John Doe"
price = ET.SubElement(book, "price")
price.text = "29.99"

# Creating a tree and writing to a file
tree = ET.ElementTree(book)
tree.write("my_file.xml")
```

JSON (JavaScript Object Notation)

JSON is a format for the transfer of data that is very lightweight. Both people and robots will have an easy time reading and writing this language. Because it stores data in the form of key-value pairs, JSON is a format that is often used for online APIs and configuration files. The following is a simple example of JSON:

```
{
  "title": "Python Programming",
  "author": "John Doe",
  "price": 29.99
}
```

In this JSON object, you have keys like "title", "author", and "price" associated with their respective values.

Reading and Writing JSON in Python

Python includes a built-in library called json for handling JSON data.

Reading JSON:

```
import json

# Read JSON data from a file
with open("my_file.json", "r") as f:
  data = json.load(f)

# Accessing JSON data
print("Name:", data["name"])
print("Age:", data["age"])
```

Writing JSON:

```
import json


# Creating a Python dictionary
data = {
  "name": "John Doe",
  "age": 30
}


# Writing JSON data to a file
with open("my_file.json", "w") as f:
  json.dump(data, f)
```

In summary, XML and JSON are valuable data formats used in Python and many other programming languages to structure and exchange information. Python provides libraries like xml.etree.ElementTree and json to work with these formats seamlessly. Understanding how to read and write XML and JSON data is essential for tasks like data exchange, configuration management, and

web development.

6.3: Hands-on Exercises

Exercise 1:

You have an XML file named "books.xml" that contains a list of books. Each book has a title, author, and price. Write a Python program to parse this XML file and print the details of each book.

Code: books.xml

```
<library>
  <book>
    <title>Python Programming</title>
    <author>John Doe</author>
    <price>29.99</price>
  </book>
  <book>
    <title>Data Science Handbook</title>
    <author>Jane Smith</author>
    <price>39.95</price>
  </book>
</library>
```

Solution:

```
import xml.etree.ElementTree as ET


# Parse the XML file
tree = ET.parse("books.xml")
root = tree.getroot()


# Iterate through each book and print details
for book in root.findall("book"):
    title = book.find("title").text
    author = book.find("author").text
    price = book.find("price").text
    print("Title:", title)
    print("Author:", author)
    print("Price:", price)
    print()
```

Output:

```
Title: Python Programming
Author: John Doe
Price: 29.99

Title: Data Science Handbook
Author: Jane Smith
Price: 39.95
```

Explanation:

- We start by importing the xml.etree.ElementTree module as ET, which allows us to work with XML data in Python.
- Next, we parse the XML file "books.xml" using ET.parse("books.xml") and get the root element of the XML tree using tree.getroot(). The root element, in this case, is <library>.
- We then use a for loop to iterate through each <book> element inside the <library>.
- For each <book>, we extract the text within the <title>, <author>, and <price> elements using the find() method. This method allows us to locate the specific child elements within the current <book>.
- We print the extracted details for each book, including title, author, and price.

Exercise 2:

You have a JSON file named "data.json" containing information about products and their prices.

Write a Python program to read this JSON file, calculate the total price of all products, and then save the result to a new JSON file named "total_price.json."

Code: data.json

```
{
  "products": [
    {
      "name": "Laptop",
      "price": 799.99
    },
    {
      "name": "Mouse",
      "price": 19.99
    },
    {
      "name": "Keyboard",
      "price": 49.99
    }
  ]
}
```

Solution:

```
import json
```

```
# Read JSON data from data.json
with open("data.json", "r") as f:
  data = json.load(f)

# Calculate the total price of all products
total_price = sum(product["price"] for product in
data["products"])

# Create a new JSON object with the total price
result = {"total_price": total_price}

# Write the result to total_price.json
with open("total_price.json", "w") as f:
  json.dump(result, f)

print("Total Price:", total_price)
```

Output:

```
Total Price: 869.97
```

Explanation:

In this exercise, we first read the JSON data from

"data.json," calculate the total price by summing the prices of all products, and then create a new JSON object with the total price. Finally, we save this result to "total_price.json."

- We imported the json module to work with JSON data.
- We read the JSON data from "data.json" and stored it in the variable data.
- We calculated the total price by iterating through the list of products inside the "products" key of the JSON data and summing up their prices.
- We created a new JSON object called result with a key "total_price" and assigned the calculated total price to it.
- Finally, we saved the result JSON object to a new file, "total_price.json," which now contains the total price of all products.

Chapter 7: Advanced Object Orientation in Python

7.1: Advanced Concepts of Object-Oriented Programming

Python is an adaptable programming language that works with many different kinds of programming methods. One of these styles, known as object-oriented programming (OOP), may be implemented in Python with the help of objects and classes.

An entity that has both properties and behaviors is referred to as an object. A bird, like a parrot, is an illustration of an object. It has:

- attributes such as name, color, age, and so on.
- behaviors such as dancing, singing, and so forth.

Similarly, a class may be thought of as an object's template.

Python Class & Object:

```
class Parrot:


  # attribute of a class
  name = ""
  age = 0


# creating object parrot1
parrot1 = Parrot()
parrot1.name = "Blu"
parrot1.age = 10


# create an additional object par2
par2 = Parrot()
par2.name = "Woo"
par2.age = 15


# access attributes
print(f"{parrot1.name} is {parrot1.age} years old")
print(f"{par2.name} is {parrot2.age} years old")
```

Output:

```
Blu is 10 years old
Woo is 15 years old
```

We built a class called Parrot with the two characteristics of name and age in the example above.

Next, we construct Parrot class instances. References (value) to the new objects are represented by parrot1 and parrot2.

Next, we used the object name and the notation to access the instance attributes and change their values.

Python Inheritance:

Using inheritance allows you to create a new class that uses the information of the previously created class without having to change it.

The newly created class is a child class, often known as a derived class. The current class is also a base class, also known as a parent class.

```
# parent class
class Animal:


  def eat(self):
    print( "I can eat!!")
```

```
  def sleep(self):
    print("I can sleep!!")

# child class
class Dog(Animal):

  def bark(self):
    print("I can bark!! Woof woof!!")

# Create an object of the Dog class
dog1 = Dog()

# Calling members of the parent class
dog1.eat()
dog1.sleep()

# Calling a member of the derived class
dog1.bark();
```

Output:

```
I can eat!
I can sleep!
I can bark! Woof woof!!
```

Members of the parent class Animal are accessible

to dog1, the object of the derived class Dog, in this case. This is so because Dog descended from Animal.

```
# Calling members of the parent class
dog1.eat()
dog1.sleep()
```

Python Encapsulation:

Encapsulation is a crucial part of object-oriented programming and is one of its primary components. The process of combining many classes' worth of methods and attributes into one is referred to as "encapsulation."

It stops other classes from accessing the methods and properties of a class so other classes can't alter them. Additionally, the concealing of data is made easier by this.

In Python, the prefix underscore is used to indicate private characteristics, such as single _, double __. For instance,

```
class Computer:
```

```
  def __init__(self):
    self.__maximumprice = 900

  def sell(self):
    print("Selling Price: {}".format(self.__maximumprice))

  def setMaxPrice(self, price):
    self.__maximumprice = price

c = Computer()
c.sell()

# change the price
c.__maximumprice = 1000
c.sell()

# using the setter function
c.setMaxPrice(1000)
c.sell()
```

Output:

```
Selling Price: 900
Selling Price: 900
Selling Price: 1000
```

We created a Computer class in the program mentioned above.

To save the maximum sale price of the computer, we utilized the __init__() function. Look at the code here.

```
c.__maxprice = 1000
```

Here, we've attempted to change __maxprice's value independently of the class. But, as __maxprice constitutes a private variable, the output does not reflect this change.

As can be seen, in order to modify the value, we must use the setter method setMaxPrice(), which requires the price as an argument.

Polymorphism:

Polymorphism is an additional fundamental concept in object-oriented programming. It simply implies a number of different shapes.

To put it another way, the same entity (method, operator, or object) may perform several activities

depending on the context in which they are executed.

Using this as an example:

```
class Polygon:
  # method for displaying a shape
  def render(self):
    print("Rendering a Polygon...")


class Square(Polygon):
  # renders a Square
  def render(self):
    print("Rendering a Square...")


class Circle(Polygon):
  # renders a circle
  def render(self):
    print("Rendering a Circle...")


# creating Square class object
s1 = Square()
s1.render()
```

```
# creating Circle class object
c1 = Circle()
c1.render()
```

Output:

```
Rendering Square...
Rendering Circle...
```

In the example that was just shown, we created a superclass called Polygon along with two sub-classes called Square and Circle. Take into consideration the application of the render() function.

The render() method's major focus is on accurately representing the form being drawn. However, the process of generating a square requires a different set of stages than that of a circle.

As a direct consequence of this, the render() method might behave differently depending on the class. Alternately, the put render() function has several uses.

7.2: Use of Special Methods and Advanced Inheritance

How Do Dunder Methods Work?

You may improve the functionality of your classes by using Python's special methods, which are a set of predefined methods. It is straightforward to recognize them thanks to the fact that they begin and finish with a pair of double underscores, such as in the case of __init__ and __str__.

The term "under-under-method-under-under" quickly grew tedious; therefore, Pythonistas came up with the word "dunder methods," which is an abbreviation of the phrase "double under."

Sometimes, the term "magic methods" is used to refer to these "dunders" or "special methods" that are available in Python. Nevertheless, the use of this word may convey the impression that they are more complicated than they really are; in the end, there is nothing "magical" about them. Consider these methods to be characteristics of language that are considered to be conventional.

By using Dunder methods, you are able to simulate the behavior of built-in types. For example, to get the length of the string, you may use the

len('string') function. Nevertheless, an empty class declaration will not automatically enable this behavior in its default state:

```
class NoLenSupport:
  pass
obj = NoLenSupport()
len(obj)
```

Output:

```
Traceback (most recent call last):
  File "c:\Users\Areej\OneDrive\Desktop\Book.py", line 4, in <module>
    len(obj)
TypeError: object of type 'NoLenSupport' has no len()
```

You may correct this by adding a __len__ dunder function in your class:

```
class LenSupport:
  def __len__(self):
    return 42

obj = LenSupport()
len(obj)
```

Slicing is another illustration. Implementing a __getitem__ function enables you to leverage the list-slicing syntax of Python, which is obj[start:stop].

Unique Approaches & the Python Data Model:

Programmers can make use of a vast array of language features thanks to the advanced architecture that the Python data model provides. These capabilities include attribute access, operator overloading, iteration, and sequences.

You can leverage Python's data architecture as a powerful API if you put a few dunder methods into practice. If you want to write more Pythonic code, you must have a solid understanding of when and how to use dunder methods.

However, for someone who is just starting, this might initially seem rather daunting. There is no need to be concerned; in this piece, for example, we will be employing dunder methods while working with a simple Account class.

Enhancing a Basic Account Class:

In order to enable the following language features, many dunder methods will be added to a basic Python class:

- New object initialization
- Representation of objects
- Enable iteration
- Overloading operators (comparison)
- Overloading the operator (addition)
- Invocation of a method
- Support for context managers (with statement)

```
class Account:
  """An ordinary account class"""


  def __init__(self, accowner, accamount=0):
    """
    This is the constructor that allows us to create
    objects for this class
    """
    self.owner = accowner
    self.amount = accamount
    self._transactions = []
```

The constructor handles the object's configuration. The owner's name, the optional start amount, and the definition of the internal trans-

actions list are sent to it in this instance so that it may monitor deposits and withdrawals.

This enables us to create fresh accounts like this one:

```
acc1 = Account('bob') # default amount = 0
acc1 = Account('bob', 10)
```

Representative Objects: __str__, __repr__

In Python, it is customary to provide the user of your class with a string representation related to that object (kind of like API documentation). With dunder techniques, there are 2 ways to do this:

The "official" string representation of the item is denoted by __repr__. You would create an object of that class in this manner. To be clear is the aim of __repr__.

An object's "informal" or easily readable string representation is called __str__. For the end user, this is.

Now, let's use these two techniques in the Account class:

```
class Account:
  """A simple account class"""

  def __init__(self, owner, amount=0):
    """
    This is the constructor enables us to make
    objects out of this class
    """
    self.owner = owner
    self.amount = amount
    self._transactions = []

  def __repr__(self):
        return 'Account({!r}, {!r})'.format(self.owner, self-
.amount)

  def __str__(self):
    return 'Account of {} with starting amount: {}'.format(
      self.owner, self.amount)
```

You may use self.__class__.__name__ to retrieve the class programmatically if you choose not to hardcode "Account" as its name.

Make sure it's __repr__ if you were to put in merely one of those to-string functions on a Python class.

I can now query the object in a variety of ways and consistently get a beautiful string representation:

Output:

```
Account of bob with starting amount: 10
```

Iterations: __getitem__, __len__, and __reversed__

We need to add a few transactions so that I can iterate through our account object. I'll provide a basic procedure to add transactions initially. Since this is only setup code explaining dunder techniques rather than a production-ready accounting system, I'll keep it simple:

```
class Account:
  """An ordinary account class"""


  def __init__(self, accowner, accamount=0):
    """
    We are able to create instances for this class
    using its constructor
```

```
    """
    self.owner = accwner
    self.amount = accamount
    self._transactions = []

  def __repr__(self):
        return 'Account({!r}, {!r})'.format(self.owner, self-
.amount)

  def __str__(self):
    return 'Account of {} with starting amount: {}'.format(
      self.owner, self.amount)

#acc = Account('bob')  # default amount = 0
#acc = Account('bob', 10)

def add_transaction(self, accamount):
  if not isinstance(accamount, int):
    raise ValueError('please use int for amount')
  self._transactions.append(accamount)
```

In order to have easy access to the account balance using the account.balance, I also developed a property to compute the account balance. This tech-

nique adds the total of each of the transactions to the starting amount:

```
class Account:
  """A simple account class"""
```

```
  def __init__(self, owner, amount=0):
    """
    We are able to create instances for this class
    using its constructor
    """
    self.owner = owner
    self.amount = amount
    self._transactions = []

  def __repr__(self):
        return 'Account({!r}, {!r})'.format(self.owner, self-
.amount)

  def __str__(self):
    return 'Account of {} with starting amount: {}'.format(
      self.owner, self.amount)
```

```
#acc = Account('bob')  # default amount = 0
#acc = Account('bob', 10)


def add_transaction(self, accamount):
  if not isinstance(accamount, int):
    raise ValueError('please use int for amount')
  self._transactions.append(accamount)


  @property
  def balance(self):
    return self.amount + sum(self._transactions)
```

Let us proceed with the account's deposits and withdrawals:

```
class Account:
  """A simple account class"""


  def __init__(self, owner, amount=0):
    """
    We are able to create instances for this class
    using its constructor
    """
    self.owner = owner
    self.amount = amount
```

```
    self._transactions = []

  def __repr__(self):
        return 'Account({!r}, {!r})'.format(self.owner, self-
.amount)

  def __str__(self):
    return 'Account of {} with starting amount: {}'.format(
      self.owner, self.amount)

  def add_transaction(self, accamount):
    if not isinstance(accamount, int):
      raise ValueError('please use int for amount')
    self._transactions.append(accamount)

  @property
  def balance(self):
    return self.amount + sum(self._transactions)

acc = Account('bob', 10)
acc.add_transaction(20)
acc.add_transaction(-10)
acc.add_transaction(50)
```

```
acc.add_transaction(-20)
acc.add_transaction(30)
```

Using this data, we want to know:

1. What number of transactions took place?
2. To get the transaction number, index the account object...
3. Cycle through the transactions

Currently, this is only feasible if the class definition is possessed. The following sentences all result in TypeError exceptions:

```
len(acc)
TypeError


for t in acc:
  print(t)
TypeError


acc[1]
TypeError
```

Output:

```
Traceback (most recent call last):
  File "c:\Users\Areej\OneDrive\Desktop\Book.py", line 36, in <module>
    len(acc)
TypeError: object of type 'Account' has no len()
```

Dunder techniques to the fore! To make the class iterable, add the following little amount of code:

```
class Account:
  """A simple account class"""

  def __len__(self):
    return len(self._transactions)

  def __getitem__(self, accposition):
    return self._transactions[accposition]
```

Output:

The preceding propositions are now valid:

```
len(acc)
5

for t in acc:
  print(t)
20
-10
50
```

```
-20
30


acc[1]
-10
```

Output:

With the __reversed__ special method, you may iterate through transactions in reverse order:

```
def __reversed__(self):
        return self[::-1]
    list(reversed(acc))
    [30, -20, 50, -10, 20]
```

The reverse list slice technique in Python is used to reverse a list of transactions. Because reversed() produces a reverse iterator rather than a list object that can be printed elegantly in the REPL, I also had to wrap the output of reversed(acc) within a list() call.

Comparing Accounts using Operator Overloading: __eq__, __lt__

Every day, in order to compare Python objects, we all write hundreds of statements:

```
2 > 1
True


'a' > 'b'
False
```

Although everything about this seems so natural, what's really going on behind the scenes is rather incredible. The behavior of > is the same with strings, integers, and other objects of the same type because these entities implement several different comparison Dunder methods; polymorphic behavior is conceivable.

Using the built-in dir() function is a simple method to confirm this:

```
dir('a')
['__add__',
...
'__eq__',  <--------------
'__format__',
'__ge__',  <--------------
```

```
'__getitem__',
'__getattribute__',
__getnewargs__',
'__gt__',  <--------------
...]
```

As we add a few transactions for later usage, let's create another account object and then compare it to the first:

```
acc2 = Account('tim', 100)
acc2.add_transaction(20)
acc2.add_transaction(40)
acc2.balance
160


acc2 > acc
TypeError:
"'>' not supported between instances of 'Account' and 'Ac-
count'"
```

Output:

```
Traceback (most recent call last):
  File "c:\Users\Areej\OneDrive\Desktop\Book.py", line 37, in <module>
    acc2 > acc
TypeError: '>' not supported between instances of 'Account' and 'Account'
```

Because we haven't built any comparison functions or inherited any from a parent class, we get a TypeError.

Add them, please. Using the functools.total_ordering decorator, I can bypass implementing all of the comparison dunder methods and only implement __eq__ and __lt__.

```
from functools import total_ordering


@total_ordering
class Account:
  """An ordinary account class"""

  def __init__(self, accowner, accamount=0):
    """
    This is the constructor that enables us to create
    objects for this class
    """
    self.owner = accowner
    self.amount = accamount
    self._transactions = []
```

```
  def __eq__(self, other):
    return self.balance == other.balance


  def __lt__(self, other):
    return self.balance < other.balance
```

And we can contrast Account instances without any issues now:

```
acc2 > acc
True


acc2 < acc
False


acc == acc2
False
```

Operator Overloading for Combined Accounts: __add__

Everything represents an object in Python. Adding two strings or two numbers using the + (plus) sign is perfectly acceptable, and it operates as expected:

```
1 + 2
3
```

```
'hello' + ' world'
'hello world'
```

Once again, polymorphism is in evidence: Have you noticed that + acts differently depending on what kind of thing it is? It concatenates texts and sums integers. The relevant "dunder" interface inside the data model is revealed by doing a fast dir() on the item once more:

```
dir(1)
[...
'__add__',
...
'__radd__',
...]
```

There is a TypeError when you attempt to add 2 instances of our Account object since it does not yet enable addition:

```
acc + acc2
TypeError
```

Output:

```
Traceback (most recent call last):
  File "c:\Users\Areej\OneDrive\Desktop\Book.py", line 46, in <module>
    acc + acc2
    ~~~~^~~~~~
TypeError: unsupported operand type(s) for +: 'Account' and 'Account'
```

Let's put __add__ into practice so that two accounts may be merged. The owner's name, beginning amounts, and transactions should all be combined per the anticipated behavior. We can utilize the iteration support that we previously added to help us with this:

```
def __add__(self, other):
    if not isinstance(other, Account):
        raise ValueError('Can only add another Account')
    owner = '{}&{}'.format(self.owner, other.owner)
    start_amount = other.amount + self.amount
    acc = Account(owner, start_amount)
    for t in self._transactions + other._transactions:
        acc.add_transaction(t)
    return acc
```

However, the earlier approach would have been more in line with what a customer of the class would anticipate.

We now have a new combined account with a total balance of about $240 (160 + 80) and an initial amount of $110 (10 + 100):

```
acc3 = acc2 + acc
acc3
Account('tim&bob', 110)


acc3.amount
110
acc3.balance
240
acc3._transactions
[20, 40, 20, -10, 50, -20, 30]
```

Keep in mind that because we are adding items of the same type, this works both ways. Typically, the __add__ function of a builtin (int, str,...) wouldn't be aware of your object if you added it to it. If so, you also need to use the reverse add technique (__radd__). Here's an example of it in action.

Python Callable Objects: __call__

The __call__ dunder function may be added to an object to provide callback functionality, much like

a standard function. We may create a beautiful report for our account class that includes all the transactions that go into its balance:

```
def __call__(self):
    print('Start amount: {}'.format(self.amount))
    print('Transactions: ')
    for transaction in self:
      print(transaction)
    print('\nBalance: {}'.format(self.balance))
```

Now, when we utilize a double-parentheses acc() syntax to access the object, I get a lovely account statement that shows the current amount as well as a summary of all transactions:

```
Start amount: 10
Transactions:
20
-10
50
-20
30

Balance: 80
```

Please remember that this is only an example from a toy. If you used a function call syntax for one of the instances of a "real" account class, it's unlikely that the console would output anything. The drawback of using a __call__ method on the objects

you create is that it may require help understanding why the object is being called.

Therefore, it is usually preferable to add a method that is explicit to the class. It would have been more apparent in this instance to have a different Account.print_statement() function.

The With Statement and Context Manager Support: __enter__, __exit__

Python has now evolved to include context managers and the addition of support for using the with statement.

Your object must adhere to a straightforward "protocol" (or interface) called a context manager in order to be utilized with a "with" statement. Adding the __enter__ & __exit__ functions to an object is all that is required to make it a context manager.

Let's extend our Account class with a rollback mechanism using the context manager support.

We return to the original state if, after adding another transaction, the balance becomes negative.

By adding two additional dunder methods, we can make use of the Pythonic with statement. Additionally, I'm adding a few print calls to help clarify the example when we demonstrate it:

```
def __enter__(self):
    print('ENTER WITH: Storing transaction backups for
rollback')
    self._copy_transactions = list(self._transactions)
    return self


  def __exit__(self, exc_type, exc_val, exc_tb):
    print('EXIT WITH:', end=' ')
    if exc_type:
      self._transactions = self._copy_transactions
      print('Rolling back to previous transactions')
      print('Transaction resulted in {} ({})'.format(
        exc_type.__name__, exc_val))
    else:
      print('Transaction OK')
```

We created a fast helper function to verify the transactions in the account as a rollback requires the raising of an exception:

```
def validate_transaction(acc, amount_to_add):
    with acc as a:
     print('Adding {} to account'.format(amount_to_add))
      a.add_transaction(amount_to_add)
      print('New balance would be: {}'.format(a.balance))
      if a.balance < 0:
        raise ValueError('sorry cannot go in debt!')
```

We may now use the statement with an Account object. Everything works okay when we initiate a transaction to sum a positive amount:

```
Balance start: 10
ENTER WITH: Making backup of transactions for rollback
Adding 20 to account
New balance would be: 30
EXIT WITH: Transaction OK

Balance end: 30
```

However, the logic in __exit__ activates and reverses the transaction when I attempt to remove too much money:

```
acc4 = Account('sue', 10)


print('\nBalance start: {}'.format(acc4.balance))
```

```
try:
  acc4.validate_transaction(-50)
except ValueError as exc:
  print(exc)


print('\nBalance end: {}'.format(acc4.balance))
```

In this instance, the outcome is different:

```
Balance: 80

Balance start: 10
ENTER WITH: Making backup of transactions for rollback
Adding -50 to account
New balance would be: -40
EXIT WITH: Rolling back to previous transactions
Transaction resulted in ValueError (sorry cannot go in debt!)
sorry cannot go in debt!

Balance end: 10
```

Python Inheritance:

Inheritance is one of the fundamental ideas of object-oriented programming (OOP) languages. By inheriting a class coming from another class, you may use this process to build a structure of classes that share a set of attributes and methods. One class's capacity to get or inherit properties from a different class is known as inheritance.

Inheritance's advantages are:

- The ability to inherit properties from one class, or base class, to another, or derived class is known as inheritance. The following are some advantages of inheritance in Python:
- Relationships in the actual world are well-represented.
- It offers a code's capacity for reuse. It is not necessary for us to rewrite the code. Furthermore, it enables us to expand a class's functionality without changing it.
- Because of its transitive nature, all of class B's subclasses would inevitably inherit from class A if class B were to descend from another class A.
- It is easy to grasp the model structure provided through inheritance.
- An inheritance usually translates into lower development and maintenance costs.

Python Syntax via Inheritance:

In Python, basic inheritance has the following syntax:

```
Class BaseClass:
  {Body}
Class DerivedClass(BaseClass):
  {Body}
```

Making a Parent Class:

A class that the child class inherits its attributes from is known as the parent class. To show the person's data, let's construct a parent class named Person with a show function.

```
# A Python program to illustrate inheritance
class Person(object):

 # Constructor
 def __init__(self, pname, pid):
  self.name = pname
  self.id = pid

 # Determining if this person is an employee
 def Display(self):
```

```
  print(self.name, self.id)



# Driver code
emp = Person("Satyam", 102) # Person class Object
emp.Display()
```

Output:

```
Satyam 102
```

Establishing a Child Class:

A class that inherits its parent class's attributes is called a child class. In this case, the Person class's properties will be inherited by another class called Emp (base class).

```
class Emp(Person):


 def Print(self):
  print("Emp (Employee) class called")


Emp_details = Emp("Mayank", 103)


# invoking parent class's function
```

```
Emp_details.Display()


# invoking child class's function
Emp_details.Print()
```

Output:

```
Mayank 100
Emp class called
```

An Example of Python Inheritance:

Let's look at an example of basic Python inheritance where a child class receives its parent class's attributes. In this instance, the parent class is "Person," and the child class is "Employee."

```
# Python program demonstrating inheritance


# Parent or Super class. Note the object in the bracket.
# (Generally, the object is made of ancestors of all classes.)
# In Python 3.x "Person class" is
# equivalent to "Person(object) class"



class Person(object):

```

```
    # Constructor
    def __init__(self, name):
        self.name = name

    # To get the name
    def getName(self):
        return self.name

    # Determining if this person is an employee
    def isEmployee(self):
        return False


# Inherited / Subclass (Note Person in bracket)
class Employee(Person):

    # Here, we return true
    def isEmployee(self):
        return True


# Driver code
person1 = Person("Geek1") # An Object of Person
```

```
print(person1.getName(), person1.isEmployee())


employee1 = Employee("Geek2") # An Object of Employee
print(employee1.getName(), employee1.isEmployee())
```

Output:

```
Geek1 False
Geek2 True
```

In Python, what is the object class?

In Python (as of version 3.x), the object is the base of all classes, much similar to the Java Object class.

- "class Test(object)" and "class Test" are equivalent in Python 3.x.
- "class Test(object)" generates a new-style class in Python 2. x, which is a class having the object serving as the parent; "class Test" makes an old-style class (in the absence of an objecting parent).

Subclassing (invoking the parent class' constructor):

The parent class of a kid class must be identified. One way to do this is by including a parent class name within the child class specification.

Class subclass_name (superclass_name) as an example!

"A" is the instance established for a particular class Person in this example. The mentioned class's __init__() is called by it. The class Person's declaration has "object" written in it. All classes in Python derive from an 'object' built-in fundamental class. The object variable or class instance is created by invoking the constructor, often known as the "__init__" function of the class.

Instance variables, sometimes referred to as objects, are the variables declared in __init__(). Therefore, "name" and "idnumber" are the class Person's objects. Similarly, the class Employee's objects are "salary" and "post." "Name" and "idnumber" are likewise objects of type Employee as the class Employee derives from class Person.

```
# Python code demonstrating the calling of parent con-
structors


# parent class
class Person(object):


  # __init__ is the constructor
  def __init__(self, pname, pidnumber):
    self.name = pname
    self.idnumber = pidnumber


  def display(self):
    print(self.name)
    print(self.idnumber)


# derived class
class Employee(Person):
  def __init__(self, empname, empidnumber, empsalary,
emppost):
    self.salary = empsalary
    self.post = emppost


    # invoking the parent class's __init__
```

```
    Person.__init__(self, pname, pidnumber)


# creation of an instance of an object variable
a = Employee('Rahul', 886012, 200000, "Intern")


# Using the instance Person class to invoke a function
a.display()
```

Output:

```
Rahul
886012
```

A Python program that shows an error if the parent's __init__() method is not called:

The child class will only have access to the parent class's instance variables if you call its __init__() method. For the same reason, the code that follows generates an error.

```
class A:
  def __init__(self, n='Rahul'):
    self.name = n


class B(A):
  def __init__(self, broll):
    self.roll = broll
```

```
object = B(23)
print(object.name)
```

Output:

```
Traceback (most recent call last):
  File "c:\Users\Areej\OneDrive\Desktop\Book.py", line 10, in <module>
    print(object.name)
          ^^^^^^^^^^^
AttributeError: 'B' object has no attribute 'name'
```

super() function:

A built-in method called super() retrieves the objects that make up the parent class. It allows the child class to access the properties and methods of the parent class.

Super() code, for instance, uses basic Python inheritance!

In this instance, we generated the child class's object, or "obj." The constructor for the inherited class "Student" initialized its data members according to the values supplied when the object was created when we invoked it. The parent class constructor was then called by utilizing the super() method.

```
# base class
class Person():
 def __init__(self, pername, perage):
  self.name = pername
  self.age = perage

 def display(self):
  print(self.name, self.age)

# derived class
class Student(Person):
 def __init__(self, name, age):
  self.sName = name
  self.sAge = age
  # taking on the parent class's properties
  super().__init__("Rahul", age)

 def displayInfo(self):
  print(self.sName, self.sAge)

obj = Student("Mayank", 23)
obj.display()
obj.displayInfo()
```

Output:

```
Rahul 23
Mayank 23
```

Adding Properties:

The ability to add additional attributes of our own to a child class and inherit the properties of a parent class is one of the capabilities inheritance offers. First, let's look at an example:

```
# parent class
class Person():
 def __init__(self, name, age):
  self.name = name
  self.age = age

 def display(self):
  print(self.name, self.age)

# child class
class Student(Person):
 def __init__(self, name, age, dob):
  self.sName = name
  self.sAge = age
  self.dob = dob
```

```
  # taking on the parent class's properties
  super().__init__("Rahul", age)


 def displayInfo(self):
  print(self.sName, self.sAge, self.dob)


obj1 = Student("Mayank", 23, "16-03-2000")
obj1.display()
obj1.displayInfo()
```

Output:

As we can see, we have added the date of birth (dob) as a new attribute to the kid class.

```
Rahul 23
Mayank 23 16-03-2000
```

Various Python Inheritance Types:

In Python, there are five distinct kinds of inheritance. They are listed below:

- **Single inheritance:**

A child class is said to have a single inheritance if it has an inheritance from a single-parent class. Above, we saw an example.

- **Multiple inheritances:**

A child class is said to have multiple inheritances when it derives from more than one parent class.

```
# Python example to demonstrate the working of multi-
ple
# inheritance


class Base1(object):
  def __init__(self):
    self.str1 = "Geek1"
    print("Base1")
```

```
class Base2(object):
  def __init__(self):
    self.str2 = "Geek2"
    print("Base2")
```

```
class Derived(Base1, Base2):
  def __init__(self):


    # Calling constructors of Base1
```

```
    # and Base2 classes
    Base1.__init__(self)
    Base2.__init__(self)
    print("Derived")


  def printStrs(self):
    print(self.str1, self.str2)
```

```
ob = Derived()
ob.printStrs()
```

Output:

```
Base1
Base2
Derived
Geek1 Geek2
```

- **Multilevel inheritance:**

It occurs when there is a child and grandchild relationship. So, a child class is going to inherit from the parent class, which will consequently inherit from its parent class, and so on.

```
# A Python program to demonstrate inheritance
```

```
# Base or Super class. Note the object in bracket.
# (Generally, the object is made of ancestors of all classes)
# In Python 3.x "Person class" is
# equivalent to "Person(object) class"

class Base(object):

  # Constructor
  def __init__(self, name):
    self.name = name

  # To get the name
  def getName(self):
    return self.name


# Inherited / Sub class (Note Person in bracket)
class Child(Base):

  # Constructor
  def __init__(self, cname, cage):
    Base.__init__(self, cname)
    self.age = cage
```

```
  # To get the name
  def getAge(self):
    return self.age

# Inherited / Sub class (Note Person in bracket)


class GrandChild(Child):

  # Constructor
  def __init__(self, gname, gage, gaddress):
    Child.__init__(self, gname, gage)
    self.address = address

  # To get the address
  def getAddress(self):
    return self.address


# Driver code
gchild = GrandChild("Geek1", 23, "Noida")
```

```
print(g.getName(), g.getAge(), g.getAddress())
```

Output:

```
Geek1 23 Noida
```

- **Hierarchical inheritance:**

A single base may be used to construct several derived classes.

- **Hybrid inheritance:**

This type blends many inheritance patterns. It is a combination of many inheritance types.

Parent Class Private Members:

We can make certain parent class instance variables private so that the child class will only have access to them if we sometimes want the parent class instance variables to be inherited via the child class.

An instance variable in Python inheritance may be made private by prefixing its name with two double underscores. For instance:

```
# Python program to demonstrate private members
# of the parent class
```

```
class C(object):
  def __init__(self):
    self.c = 21

    # d is a private instance variable
    self.__d = 42
```

```
class D(C):
  def __init__(self):
    self.e = 84
    C.__init__(self)

object1 = D()

# produces an error as d is a private instance variable
print(object1.c)
print(object1.__d)
```

Output:

Here, we are able to observe that the variable "c" prints its value 21, as shown on the console when

we attempt to print it. However, an error was produced when we attempted to print the letter "d." This is a result of the underscores being used to make the variable "d" private. The error occurs because the child class "D" cannot access it.

```
21
Traceback (most recent call last):
  File "c:\Users\Areej\OneDrive\Desktop\Book.py", line 21, in <module>
    print(object1.__d)
          ^^^^^^^^^^^
AttributeError: 'D' object has no attribute '__d'
```

7.3: Exercises

Exercise 1:

Creating a hierarchy of classes for the management of humans.

Code:

```
class Human:
  def __init__(self, name, age, gender):
    self.name = name
    self.age = age
    self.gender = gender

  def introduce(self):
     print(f"My name is {self.name}, I am {self.age} years
old, and I am a {self.gender}.")
```

```
class Student(Human):
  def __init__(self, name, age, gender, student_id):
    super().__init__(name, age, gender)
    self.student_id = student_id

  def study(self, subject):
    print(f"{self.name} is studying {subject}.")

class Employee(Human):
  def __init__(self, name, age, gender, employee_id, posi-
tion):
    super().__init__(name, age, gender)
    self.employee_id = employee_id
    self.position = position

  def work(self):
    print(f"{self.name} is working as a {self.position}.")

# Create instances of Human, Student, and Employee
person1 = Human("Alice", 30, "Female")
student1 = Student("Bob", 20, "Male", "S12345")
employee1 = Employee("Charlie", 25, "Male", "E98765",
```

```
"Software Developer")


# Demonstrate methods
person1.introduce()
student1.introduce()
student1.study("Math")
employee1.introduce()
employee1.work()
```

Output:

```
My name is Alice, I am 30 years old, and I am a Female.
My name is Bob, I am 20 years old, and I am a Male.
Bob is studying Math.
My name is Charlie, I am 25 years old, and I am a Male.
Charlie is working as a Software Developer.
```

Explanation:

1. First, we define the basic human class. Name, age, and gender are the three characteristics of the Human class. Additionally, it contains a feature called introduce that writes out human data.
2. Next, we use the super() method to construct two subclasses, Student and Employee, which both inherit directly from the Human class.

3. The Student class has a study method that enables the Student to declare the fact that they are pursuing a certain topic, as well as an extra feature named student_id.
4. The Employee class has a work method that enables the Employee to declare the fact that they have a job in a certain capacity, along with two new attributes: employee_id and position.
5. To represent various people, we build instances of the Human, Student, and Employee classes. Specific characteristics are initialized for every instance.
6. Lastly, we call introduce, work, and study with the corresponding examples to show off the approaches. This enables us to observe how every class may relate to and represent people in various capacities.

An elementary class structure for managing people is shown in this code. It demonstrates inheritance, in which you may represent people with particular roles and behaviors by using the Student & Employee classes, which inherit methods and properties from the Human class. Depending on your unique use case, you can expand this hierarchy further to add more specialized class methods.

Chapter 8: Popular Python Libraries

A library is an invaluable collection of pre-written code snippets and functions that programmers may use to carry out certain tasks or create different functionality inside their software applications. In the world of programming, a library is referred to as a code archive or code repository. These libraries are quite similar to toolkits in that they offer a wide variety of components that may be used by programmers to assemble their software in a manner that is both more efficient and effective. Libraries, in their most basic form, perform the function of a repository for reusable code modules, which streamlines the process of writing code and speeds up the creation of complicated programmes.

The fundamental objective of libraries is to facilitate the simplification of the development process by providing ready-made answers to frequent issues that arise in programming. In order to speed their projects, developers may tap into the power

of libraries rather than rewriting every line of code from scratch. This saves them from having to "reinvent the wheel." These libraries are developed by experienced developers who have previously overcome certain obstacles. As a result, they make it possible for others to benefit from the developers' knowledge and previous work. Libraries are essential to the creation of contemporary software due to the fact that they promote the reusability of code, modularity, and the simplicity of maintenance.

8.1: Python Library Resources:

A More In-Depth Look

In particular, Python libraries are aggregations of pre-written code modules with the purpose of expanding the capabilities of the Python coding language. Python is a commonly used programming dialect that has obtained recognition in a variety of fields because of its user-friendliness and adaptability. Some of these fields include web development, data analysis, and machine learn-

ing. Python libraries extend the usefulness of the language by making available a broad range of features that may be customized to meet the specific requirements of individual programmers.

These libraries serve as important add-ons to Python, expanding its capabilities and allowing it to be used in a wider variety of contexts. These libraries are designed to be easily included into Python applications by developers, which results in considerable time and effort savings. They do this so that they may also draw into the collective knowledge and collaborative spirit of the Python community, which continuously contributes to the creation and enhancement of these libraries. Specifically, they tap into the collective wisdom of the Python community.

Python libraries may be conceptualised as specialised toolkits, each of which is designed to handle certain programming issues or application areas. Let's have a look at some of the most popular

Python libraries and the kind of projects that they are best suited for:

- NumPy: NumPy is a fundamental library for doing numerical computations in the Python programming language. It offers support for huge, multi-dimensional arrays and matrices, in addition to a rich collection of mathematical functions that may be used to perform operations on such arrays and matrices. Because this library is required for any jobs that involve doing calculations with numbers, it is considered to be an essential component in domains such as data science and scientific computing.
- Pandas is a data manipulation and analysis package that makes it easier to deal with structured data such as tables and data frames. Because it makes it possible to do activities such as cleaning, trans-

forming, and exploring data, it is a crucial tool for data scientists and analysts.

- Matplotlib: Matplotlib is a charting package that makes it easier to create a variety of visualisations in Python, including those that are interactive, animated, or static. Data visualisation relies heavily on it, since it helps with both the display of data and the understanding of that data.
- Scikit-Learn: Machine learning library that provides an extensive array of resources for the creation and deployment of machine learning models. Because it provides support for a wide variety of algorithms, model assessment methods, and preprocessing approaches, machine learning practitioners often choose to work with it.
- Django: Django stands as a sophisticated web framework designed for Python, specifically tailored to streamline web application development. It provides a large

number of features and standards that may be used while rapidly developing safe and dependable online applications.

- Flask On the other hand, Flask is a light-weight web framework that is ideal for the construction of more compact web applications and APIs. It gives developers access to the fundamentals for web development while enabling them to choose and include additional components according to their own requirements.
- Suggestions for: If you want to make HTTP requests in Python, you absolutely need to use the Requests package. It makes operations like as sending GET and POST requests, handling authentication, and keeping cookies more simpler, making it an essential component for web scraping, API integration, and the building of online applications.

Only a small portion of the vast ecosystem that Python supports is represented by these libraries. Because each library caters to a different set of criteria, it is possible for developers to pick and select the tools that are the most appropriate for the specifications of their individual projects. By using these libraries, programmers are able to construct dependable programmes that are packed with features without having to design each individual piece of functionality from the ground up.

In a nutshell, programming libraries are a developer's best friend since they provide a wide reservoir of pre-written code and functions that accelerate the process of software creation. Python is a very open programming language, and it has a large collection of libraries that allow its capabilities to be extended into many other fields. These libraries provide developers with the tools necessary to handle complicated jobs in an effective manner, encourage the reuse of code, and tap into the collective wisdom of the Python community.

Python libraries will continue to be a vital tool for software developers who are interested in developing cutting-edge and potent software solutions while the world of technology undergoes continuous advancement.

8.2: Numpy

One of the most important libraries for the computer language Python is referred to as NumPy, which is an acronym for the term "Numerical Python." It is an essential piece of software for data scientists, engineers, researchers, and programmers working on a diverse range of projects since it is an essential component in enabling efficient numerical computations and data manipulations. This comprehensive analysis will delve into the complexity of the NumPy library, covering its application examples as well as the outputs that it generates.

The Acquiring of Knowledge Regarding NumPy

NumPy is a programming language that provides aid with multidimensional arrays and matrices, as

well as to a wide range of mathematical functions that can be used to carry out operations on these arrays. These functions may be used to conduct operations such as adding, subtracting, and multiplying. As a result of the qualities outlined above, using NumPy as a foundation for a broad range of scientific and data-driven projects is an essential step in the development process. The library is open to the public without charge, and both the academic community and the corporate sector make substantial use of its resources.

At its core, NumPy introduces the concept of a NumPy array, which is also referred to as an ndarray. An ndarray is a multidimensional array that is homogeneous and has the ability to contain objects of the same data type. NumPy arrays are also known as numpy arrays. For the purpose of carrying out numerical computations, these arrays are quite helpful due to the fact that they are homogeneous and have a size that is always the same. NumPy offers an array object that is not only

simpler to use than Python's built-in data structures, but also offers a wide range of operations that are geared to the requirements of numerical work. NumPy was developed by the Python software development team.

The following is a list of characteristics and capabilities that NumPy possesses:

1. Arrays that include several dimensions

NumPy's fundamental data structure is called a ndarray, and it is used throughout the library. It allows you the ability to produce arrays of various dimensions, such as a one-dimensional (vectors), a two-dimensional (matrices), and even higher-dimensional arrays than that. For example, you can create matrices with two dimensions, vectors with one dimension, and so on. It is possible to utilise it efficiently in a broad variety of scenarios, ranging from simple numerical calculations to complex data manipulations because to its versatility, which enables it to adapt to its environment.

Code

```
import numpy as np


# Creating a one-dimensional NumPy array
arr1d = np.array([1, 2, 3, 4, 5])


# Creating a two-dimensional NumPy array
arr2d = np.array([[1, 2, 3], [4, 5, 6]])
```

2. The Processes Involved in Mathematics

NumPy offers an extensive library of mathematical functions and operators that may be used in conjunction with ndarray objects in a seamless manner. The ability to conduct operations like as addition, subtraction, multiplication, division, and other operations elementwise across arrays is very valuable for doing numerical calculations and may be achieved via the usage of these functions.

Code:

```
import numpy as np

```

```
arr1 = np.array([1, 2, 3])
arr2 = np.array([4, 5, 6])


# Element-wise addition
result = arr1 + arr2  # Output: [5, 7, 9]


# Element-wise multiplication
result = arr1 * arr2  # Output: [4, 10, 18]
```

3. Transmission of Signals

The broadcasting feature of NumPy makes it possible to perform actions on arrays of varying shapes and sizes without having to convert them to the same shape manually. This feature streamlines the writing of code and improves efficiency by preventing the copying of superfluous data.

Code

```
import numpy as np


arr1 = np.array([1, 2, 3])
scalar = 2
```

```
# Broadcasting: Multiplying the scalar with the array
result = arr1 * scalar  # Output: [2, 4, 6]
```

4. Ufuncs; universal functions

NumPy universal functions are functions that act element-wise on ndarray objects. This kind of operation results in calculations that are both quick and efficient. A wide variety of mathematical and logical processes are included in this group of functions.

Code:

```
import numpy as np

arr = np.array([1, 2, 3, 4])

# Applying a universal function (ufunc)
squared = np.square(arr)  # Output: [1, 4, 9, 16]
```

5. Slicing and Sorting the Food

Arrays may be sliced and indexed with NumPy utilising approaches that are both simple and very powerful. With the help of this functionality, it is simple to extract certain items or subsets from an array.

Code:

```
import numpy as np


arr = np.array([10, 20, 30, 40, 50])


# Slicing to get a subset of the array
subset = arr[1:4] # Output: [20, 30, 40]


# Indexing to access a specific element
element = arr[2] # Output: 30
```

6. The Generation of Random Numbers

The random submodule of NumPy is where the routines for generating random numbers may be found. Applications like simulations, statistical

analysis, and artificial intelligence may all benefit greatly from having this information.

Code:

```
import numpy as np


# Generating random numbers
random_numbers = np.random.rand(5)  # Output: Array of 5 random numbers between 0 and 1
```

NumPy's many uses in the real world

Because of its flexibility and high performance, NumPy has become an essential library in many different fields. Listed below are a few real-world applications of NumPy that come to mind:

1. Analysing the Data and Statistical Models

Data scientists and analysts often make use of NumPy as an effective tool for data manipulation and analysis. It gives users access to crucial tools for carrying out activities such as data cleansing, aggregation, and statistical analysis. The ndarray

data structure provided by NumPy serves as the basis for other data structures, such as pandas DataFrames, and enables smooth interaction with other libraries for data analysis.

Code:

```
import numpy as np


# Calculating mean and standard deviation
data = np.array([10, 15, 20, 25, 30])
mean = np.mean(data)
std_dev = np.std(data)

```

2. Computational Science and Mathematics

NumPy is widely used for scientific computing jobs by academics and professionals in the field. The ability of NumPy to solve differential equations, simulate physical systems, and handle experimental data are all examples of the use of these numerical skills.

Code:

```
import numpy as np
```

```
# Solving a system of linear equations
coeff_matrix = np.array([[2, 3], [1, -2]])
constants = np.array([8, -4])
solution = np.linalg.solve(coeff_matrix, constants)
```

3. The Process of Images

NumPy is used by a wide variety of image processing applications, including computer vision and medical imaging, since it facilitates efficient pixel-level computations and conversions.

Code:

```
import numpy as np
import matplotlib.pyplot as plt

# Reading and manipulating an image using NumPy
image = plt.imread('image.jpg')
image_gray = np.mean(image, axis=2)  # Convert to
grayscale
```

4. Automatic Learning Machines

NumPy is the basis upon which other machine learning frameworks such as Scikit-Learn and TensorFlow are built. NumPy arrays are used to effectively represent data as well as model parameters, which enables for more effective training and inference procedures.

Code:

```
import numpy as np
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression


# Loading a dataset and training a machine learning model
data = load_iris()
X_train, X_test, y_train, y_test = train_test_split(data.
data, data.target, test_size=0.2)
model = LogisticRegression()
model.fit(X_train, y_train)
```

5. Modelling and Computer Simulation

The simulation and modelling tools that make use of NumPy's random number generating capabil-

ities are essential. Researchers and engineers are able to simulate complicated systems and analyse how they behave under a variety of different scenarios thanks to this tool.

Code:

```
import numpy as np


# Simulating a random walk
steps = np.random.choice([-1, 1], size=1000)
positions = np.cumsum(steps)
```

Results

The output of NumPy is mostly composed of ndarray objects, which are effective storage containers for numerical data. Depending on the application and the particular activity at hand, these arrays may be used in a variety of different ways.

8.3: Pandas

The Python package known as Pandas has become an industry leader in the domain of data manip-

ulation and analysis in recent years. It offers data scientists, analysts, and developers' access to a broad variety of tools and data formats that make the process of managing, cleaning, and analyzing data much more straightforward. During this in-depth investigation, we will dig into the inner workings of the Pandas library, complete with explanations, real-world applications, and code samples.

Pandas, which is an abbreviation for "Python Data Analysis Library," was developed by Wes McKinney and made available for the first time in the year 2008. Since that time, it has developed into a tool that is necessary for both data professionals and data aficionados. Series and DataFrame are the two fundamental data structures that are introduced by Pandas at the framework's core.

A Series is an object that functions similarly to a one-dimensional array and may store a wide variety of data types, including strings, integers, and floats, among others. It is analogous to a single col-

umn in a spreadsheet or a feature alone on its own in a dataset.

DataFrame: A DataFrame is a kind of data structure that is similar to a table and has two dimensions. It is made up of rows and columns. It may be used in a variety of contexts and looks like a spreadsheet or a SQL table. One way to think of a DataFrame is as a collection of Series objects, with each column in the DataFrame representing one of those Series.

Pandas perform very well when it comes to ingesting data from a wide variety of sources, such as CSV files, Excel spreadsheets, SQL databases, and many more. After the data has been put into a Pandas DataFrame, the DataFrame turns into a playground for the exploration, manipulation, and analysis of the data.

Pandas Comprises the Following Features:

1. Data Cleaning and Transformation

The capacity of Pandas to ease the process of data cleansing and transformation is one of the program's most notable characteristics. It offers a wide variety of strategies for dealing with missing data, removing columns that aren't essential, and rearranging data.

Example 1: Managing the Case of Missing Data

Let's say we have a dataset that contains missing values that are denoted by the symbol 'NaN.' Pandas make it possible for us to simply manage these data that are missing.

Code:

```
import pandas as pd


data = {'A': [1, 2, None, 4, 5], 'B': [None, 2, 3, 4, 5]}
df = pd.DataFrame(data)


# Fill missing values with a specific value (e.g., 0)
df.fillna(0, inplace=True)
```

2. The aggregation and classification of data

Data aggregation and summarization are made easier using the sophisticated features provided by Pandas. Data may be categorized into groups according to predetermined standards, and then mathematical operations such as mean, sum, or count can be carried out on each category.

Example 2: Grouping and Aggregating

Let's say we have a dataset of sales, and we want to figure out how much each product category contributed to the overall sales.

Code:

```
import pandas as pd


data = {'Product': ['A', 'B', 'A', 'C', 'B'],
    'Sales': [100, 200, 150, 120, 180]}
df = pd.DataFrame(data)


# Group by 'Product' and calculate the sum of 'Sales'
result = df.groupby('Product')['Sales'].sum()
```

3. The representation of data

Popular data visualisation tools, such as Matplotlib and Seaborn, may be easily integrated with Pandas without any problems. Users are able to generate meaningful plots and charts as a result of this, which helps them visualise data trends.

Example 3: Bar Plots

Let's pretend we want to create a visual representation of the sales data from the earlier example.

Code:

```
import pandas as pd
import matplotlib.pyplot as plt


data = {'Product': ['A', 'B', 'A', 'C', 'B'],
    'Sales': [100, 200, 150, 120, 180]}
df = pd.DataFrame(data)


# Group by 'Product' and calculate the sum of 'Sales'
result = df.groupby('Product')['Sales'].sum()


# Create a bar plot
```

```
result.plot(kind='bar')
plt.xlabel('Product')
plt.ylabel('Total Sales')
plt.title('Total Sales by Product Category')
plt.show()
```

4. Combining and combining of data

In a manner similar to that of SQL JOIN procedures, Pandas makes the process of combining and connecting datasets much more straightforward. You may combine DataFrames by using columns or indices that they have in common.

Example 4: Combining Data Frames

Suppose we have two datasets: one having information on customers, and another including information about customers' past purchases. We are able to combine them using a standardised 'CustomerID' column as the basis.

Code:

```
import pandas as pd
```

```
# Create two DataFrames
customers = pd.DataFrame({'CustomerID': [1, 2, 3],
                          'Name': ['Alice', 'Bob', 'Charlie']})

purchases = pd.DataFrame({'CustomerID': [2, 3, 1],
                          'Product': ['Widget', 'Gadget', 'Widget']})

# Merge the DataFrames on 'CustomerID'
merged_data = pd.merge(customers, purchases, on='CustomerID')
```

Real-World Applications of Pandas

The usefulness of pandas may be shown in a very broad variety of different fields and applications. The following are some examples of real-world situations in which Pandas excels:

1. Analysing the Data and Conducting Exploratory Research

When it comes to jobs involving data analysis and exploration, Pandas is the tool of choice. Pandas is a data analysis tool that makes the process of

analysing data distributions, summarising statistics, and locating outliers much easier.

Example 5: Investigating the Data

Let's say we have a dataset consisting of various property prices, and we wish to investigate the dataset's fundamental statistics.

Code:

```
import pandas as pd


# Load housing price data from a CSV file
data = pd.read_csv('housing_prices.csv')


# Get basic statistics of the 'Price' column
summary_stats = data['Price'].describe()
```

2. Machine Learning Data Preprocessing

Preprocessing data is often necessary before the models are trained using machine learning. Pandas is useful for encoding categorical variables, scaling, and feature selection.

Example 6: Preparing Data for Use

We may need to use Pandas to scale numerical features in a machine learning project.

Code:

```
import pandas as pd
from sklearn.preprocessing import StandardScaler

# Load a dataset with numerical features
data = pd.read_csv('numerical_data.csv')

# Initialize a StandardScaler
scaler = StandardScaler()

# Scale the numerical features
scaled_data = scaler.fit_transform(data[['Feature1', 'Feature2']])
```

3. Analysis of Time Series

Because of its exceptional ability to handle time series data, pandas is a priceless tool for applications

like forecasting, financial analysis, and sensor data processing.

Example 7: Analysis of time series

Let us consider an example where we want to compute daily returns using historical stock price data.

Code:

```
import pandas as pd


# Load historical stock price data
data = pd.read_csv('stock_prices.csv', index_col='Date', parse_dates=True)


# Calculate daily returns
daily_returns = data['Close'].pct_change()
```

4. Data Transformation and Cleaning

Data transformation and cleansing are often needed for a variety of sources. Pandas makes these chores easier, freeing up data scientists to

concentrate on insights rather than data manipulation.

Example 8: Converting Data

Let's say we want to extract sentiment ratings from a collection of user evaluations.

Code:

```
import pandas as pd
from textblob import TextBlob


# Load a dataset of user reviews
data = pd.read_csv('user_reviews.csv')


# Define a function to get sentiment scores
def get_sentiment(text):
  analysis = TextBlob(text)
  return analysis.sentiment.polarity


# Apply the function to the 'Review' column
data['Sentiment'] = data['Review'].apply(get_sentiment)
```

Pandas has transformed the world of data analysis and manipulation. It is a vital tool for data professionals and academics due to its user-friendly data structures, comprehensive toolbox, and smooth interface with other Python modules. Pandas accelerates the process of examining data, preparing it for machine learning, visualising trends, or executing sophisticated data transformations, saving you time and effort.

We've explored Pandas' core data structures, major features, and real-world applications in this investigation. You can use Pandas to extract useful insights from your data, opening the path for informed decision-making and data-driven innovation. Pandas is an essential tool in the toolset of anybody dealing with data as data continues to play a fundamental role in our digital world.

Chapter 9: Web Development with Flask

9.1: Creating Simple Web Applications with Flask

Building a website using Python:

Numerous modules and frameworks, such as Flask, Django, and Bottle, enable you to create web-pages with Python. Yet Django and Flask are the well-liked ones. In comparison to Flask, Django is easier to use, while Flask offers more programming flexibility.

You must be familiar with a few common terminologies in order to comprehend what Flask is.

1. WSGI

The Web Server Gateway Interface (WSGI) is being embraced as the industry standard for developing Python online applications. A standard interface for web servers and web applications is called Web Server Gateway Interface (WSGI).

2. Werkzeug:

It is a WSGI toolkit that provides utility functions, response objects, and requests. Creating a web framework over top of it is made possible by this. Werkzeug serves as one of the foundations for the Flask framework.

3. Jinja2

One well-liked Python templating engine is called jinja2. To display dynamic web pages, the web templating system integrates a template with a specific data source.

The Flask framework for web applications is written in Python.

The Jinja2 template engine and Werkzeug WSGI tools serve as the foundation for Flask. Projects from Pocco are both.

Installation:

Two packages are needed to configure your environment. A user may construct many Python envi-