Python Notes

By

@codewith_random

Happy Learning.....

Index

Python Basics

1.1 Introduction

Python is a well-liked high-level programming language known for its ease of use and adaptability. "Guido Van Rossum" developed Python in the late 1980s, and it has since become widely used in a variety of fields such as web development, data analysis, artificial intelligence, scientific computing, and more.

.Python is known for its readable syntax, which focuses readability of the code and lowers of the cost of program maintenance. Code blocks are defined by indentation, which eliminates the need for explicit braces or keyword and improves the consistency and clarity of the code.

The procedural and object-oriented programming paradigms are both supported by Python, giving developers more freedom when creating software structures. Through classes and modules, it promotes code reuse and modularity, making it simples to handle complicated projects.

Python's huge standard library, which provides a variety or pre-built modules and functions that makes certain jobs easier, is one of its strengths. This library makes it easier for developers to manipulate files, communicate over networks, analyse data and more.

Python's dynamic typing system promotes quick development by allowing variables to be assigned without

functional programming..

Libraries like NumPy, pandas, and scikit-learn highlight the language's attraction in data science and machine learning. These technologies enable innovative AI research and applications development by enabling effective data manipulation, analysis, and model training.

Python's web frameworks, such as "Django" and "Flask" make it easier to construct online applications, allowing for the development of dynamic and interactive webpages. Its connectivity with several operating systems, such as Windows, macOS, and Linux, Further enhances its adaptability.

Python code may be dynamically and interpretively performed without the need for compilation, which speeds up the development process. Its interactive shell makes testing and experimenting easier, assisting both experts and students.

1.2 Variables

In Python, variables are essential building blocks for managing and storing data inside programs. To store data like numbers, text or objects, they serve as named placeholders. Python variables feature dynamic typing, which allows them to change type while being used, so you don't need to explicitly indicate what type you want to them to be.

Simple rules for naming must be followed when choosing a variable's name, and the "=" sign is used for adding on

a value. For Example:
age = 20
name = "Pawan"
"age" and "name" are variables here, and the values "20" and "Pawan" are what they represent. Based on the value supplied to the variable, Python finds the proper data types to assign.

# Integers (int), Floating-point numbers (Float), strings (str), lists (list), dictionaries (dict) and other data types include several data types of data that may be stored in variables, For Example:
temperatire = 97.6
days_of_week = ["Monday", "Tuesday", "Sunday"]

# The usage of variable sin your code enables dynamic and interactive behaviour by allowing calculations to be performed, user input to be stored, and data manipulation to be made possible. For Example:
radius = float (input ("Enter the radius: "))
area = 3.14 * radius ** 2
printf ("The area of the circle is: ", area)

Important: -
Python variables must be begin with letter (a-z, A-Z) or an underscore (_), and they are case-sensitive ("myVariable" and "myvariable") are two separate variable. They are not Python keywords and may also have numbers (after the first character).
Keep in mind that variables are pointers to places in memory where data is kept, not just simple container.

Multiple variables may all refer to the same value, and unless they all point to same memory address, altering the value of one variable has no impact on the value of the others.

1.3 Data Types

Python programs depend heavily on data types because they specify how data is expressed, Saved, and processed. In order to write effective and efficient code, one must first understand data types. Python is very well-liked for a variety of applications. From straightforward scripting to intricate web applications and data science projects, thanks to its dynamic typing and large built-in data types.

In Python, We have 4 different Data Types.

i. Numeric Data Type

The numeric data type is used in Python to represent and wrok with numerical numbers. "Integers" (int) and "Floating-point numbers" (float) are two primary categories of numeric data types.

Let's examine these subtypes with some example:

"integer (int)" type

Integer is whole numbers without decimal points. They can be positive, negative or zero. For Example:

```
age = 20
temperature = -12
population = 100

# Many arithmetic operations, including addition,
subtraction, multiplication, and division are supported by
```

numerical data types. For example:

```
x = 10
y = 3

//Addition
print( "Sum:" , sum_result) // Output : 13

//Subtraction
difference = x-y
print("Difference: " difference) //Output: 7

//Multiplication
product = x*y
print("Product: " , product) // Output: 30

//Division
quotient = x/y
print("Quotioent: ' quotient) //Output: 3.333333
```

"Floating-Point (float)" Type

Floating-point numbers can have decimal points and represent actual numbers. They can also be written using the "e" or "E" notation in scientific notation. For Example:

```
pi = 3.14159
temperature = -6.5
distance = 1.6e6 //1.6 * 10^6
```

ii. Text Data Type

The str (string) class in python is used to represent the text data types. Sequences of characters, including letters, numbers, and symbols, are stored in strings.

Many programming activities, such as user input, text processing, formatting, and others frequently include text data.

Example of using the text data type (str) in Python:

```
"Creating Strings"
# By using single quotes
message = 'Hello, Pawan'

# By using double quotes
name = "Pawan"

# By using triple quotes
quote = "'Hello, am Pawan'"

"Connecting String Together"
age = 30
# Using the format method
info = "Name : {}, Age: {}". Format (name, age)
# Using F-Strings or Formatted string lateral
f_info = f "Name: {name}, Age:{age}"

"Accessing characters in string"
# Accessing the first character
first_letter = name [0]

# Slicing to extract a substring
substring = message [7:12]

"String Methods"
# getting the length of a string
length = len (name)
```

```
# Converting to uppercase
uppercase_name = name.upper()

# Converting to lowercase
lowercase_name = name.lower()

# capitalizing the first letter
capitalized_name = name.capitalize()

"Checking For Substrings"
#checking if a string starts with a certain substrings
contains_hello = message.startwith("Hello")

# Checking if a substrings is present
conatins_pawan = "Pawan" in message

"String Manipulation"
# Removing leading and trailing whitespace
trimmed_message = message.strip()


"Splitting and joining strings"
#splitting a string into a list of words
word_list = message.split (" , " )

#joining a list of words into single string
joined_words = " - " .join(word_list)


"String Formatting"
# Formatting a floating-point number
formatted_float = "{:, 2f}".format(3.14159)
```

print (formatted_flaot)

iii. Set Data Type

A "set" is built- in data type in python that represent an unordered group of distinct items. When you need to keep a collection of elements without care to order and guarantee that each component only shows once, sets come in very handy. Curly braces "{ }" are used to surround sets.

Here is the example of using set in Python:

"Creating a Set"

fruits = {"apple" , "banana" , "orange" , "banana"}

vegetable = {"carrot" , "Spanish" , "broccoli"}

"Displaying a set"

print(fruits)

"Checking membership"

print ("banana" in fruits) #Output: True

print("pear" in fruits) #Output: False

#Adding Elements to the set

fruits.add("pear")

print(fruists)

#Output: {"apple" , "banana" , "orange" , "pear"}

"Removing elements to the set"

fruits.remove("apple")

print(Fruits) # Output: {"banana" , "orange" , "pear"}

"Basic Set Operations"

#union of sets

food = fruit.union(vegetables)
print(food)
#output: {“banana” , “Spanish” , “orange” , carrot” , “pear” , “broccoli”}

#Intersection of sets
common_items = fruits.intersection(vegetables)
print(common_items)

#Difference of sets
unique_fruits = fruits.difference (vegetables)
print(unique_fruits)
#Output: {“banana” , “grape” , “orange” , “pear”}

iv. Boolean Data Type
The Boolean data type, which may represent “True or false”, is a Boolean data type in python. In Python program, Boolean are mostly used for logical operations, comparisons and conditional expressions.
Here is the example of using the Boolean data type in Python.
“Basic Usage of Boolean in variables”
is_sunny = True
is_raining – False
print (is_sunny) #Output: True
print(is_raining) #Output: False
Here we define two Boolean variable “is_sunny” and “is_raining” and they assigned values “True” and “false” respectively.

1.4 Operators

Python supports wide range of operators that help manipulate values, make decision, and control the flow of your program. Operators in python are symbols or special keywords that allow you to perform various operations on data, including arithmetic calculations, comparison, logical evaluations, and more.
In Python, we have eight types' operators.

i. Arithmetic Operators: - These operators are used for mathematical calculations and operations.
Symbols Names
+ Addition
- Subtraction
* Multiplication
/ Division
% Modulus
** Exponentiation
// Floor Division

ii. Comparison Operator: - These Operators compare values and return Boolean result.
Symbols Names
== Equal to
!= Not Equal to
> Greater than
< Less than
>= Greater than or equal to
*<= Less than or equal to

iii. Logical Operators: - These operators are used for logical operations and combine Boolean value.
Symbols Work

and Return "true" if both operations "true"
or Return "true' if at least one operation is "true"
not
Return the opposite Boolean value of the operations.

iv. Assignment Operators: - These operators are used to assign value to variable.
Symbols Name
"=" Assignment
"+=", "-=" , "*=" , "/=" , "%=" Compound Assignment

v. Bitwise Operators: - These operators perform operations on individual bits of integer.
Symbols Name
& Bitwise AND
| Bitwise OR
^ Bitwise XOR

vi. Membership operators: - These operators test if a value is a member of a sequence (like a list, tuple, or string).
Symbols Work
"in" Return "true" if the value is found in the sequences.
"not in" Return "true" if the value is not found in the sequences

vii. Identify Operators: - These operators compare the memory addresses of two objects.
Symbols
Work
"is" Return "true" if both variables point to the same

objects.
"is not " Return "true" if the variable points to different objects.

viii. Ternary operators: - The Operator provides a shorthand for conditional expressions.
Symbols Work
"x if condition else y" Returns "x" if the condition is "true", otherwise returns "y".

1.5 Control Statements
Programmers use control statements to change a program's execution flow based on specific condition or criteria. They provide programmers the ability to make choices, reuse code, and selectively run particular chunks of code. Control statements are essential for writing flexible and dynamic programs in Python.

In python, we have three types of Control Statements
i. Conditional Statements
Making judgements in your code based on particular conditions is made possible via conditional statements, fundamental notion in programming with the help of these statements, your program may run various blocks of code in response to whether a particular condition is true or not. The "if", "elif", and "else" keywords are often used to create conditional statements in python code.

"if" – A conditional statements fundamental format is an "if" keywords followed by an evaluation condition. The code block positioned behind the "if" statement is executed "if" the condition is satisfied. The code block is

skipped if the condition is false. Here is an Example:

```
age = 18
if age > = 18;
print ("Adult")
```

The program in this example determines if the age variable is greater than or equal to 18. The indented code block is run and the message "Adult" is printed since the condition is true (18 is in fact more than or equal to 18).

"else" – you may wish to provide another option of action to perform if the condition is false. The "else" keyword useful in this situation, For Example:

```
age = 13
if age >= 18;
print("Adult")
else:
print("No Adult")
```

"elif" – The "elif (else-if)" keyword can be used for more complex situations when there are many conditions to take into considerations, For Example:

```
score = 72
if score >= 90;
print("A Grade")
elif score >= 80;
print(" B Grade")
elif score >= 70;
print(" C Grade")
else:
print("Improved Yourself")
```

The program in this case compares the score variables to several situations. The subsequent statements are

bypassed in favour of the initial condition that evaluates to true. The code block after the else line is run if none of the statement in true.

ii. Loop Statements

Loop Statements are fundamental programming structures that enable a certain piece of code to be run repeatedly in response to predefined conditions. They give us a mechanism to automate difficult activities, cycle across data structures, and manage program flow. "For Loops" and "While Loops" are the two primary forms of loop statement used in Python.

"For Loop" – Iterating over a sequences (such as list, tuple, String) with a "for loop" allows you to run a block of code for reach element of the sequences. When the number of iterations is known or when you to process every item in collection, this kind of loop is quite helpful.

For Example:

```
fruits = ["apple", "banana", "cheery"]
for fruits in fruits;
print(fruit)
```

The "Fruit" variables value is assigned to the current item in the list each time the "For loop" iterates over the list of fruits. For each item in the list, the loop runs, the indented bit of code (in this case, printing the fruit name).

"while Loop" – While a certain condition is true, a while loop is used to continually run a block code. When the number of iterations is not known in advance or when you need to carry out a job repeatedly up until a certain condition is satisfied, this type of loop is helpful.

count = 0
while count < 5;
print ("Count", count)
count += 1
As long a count is less than 5, the while loops keeps running the block of code indented. Every time the loop is executed, the count value is increased.

iii. Control Flow Statements
Developers can change the direction in which "loops" and "conditional" statements execute by using control flow statements. "break" and "continue" are the two primary form of control flow statements used in python.

"break" – The 'break" statement enables a quick end to a loop depending on a specific condition. This is helpful if you want to end a loop before all of iterations are complete.
numbers = [1,2,3,4,5]
for num in numbers;
if num == 3;
break
print(num)
When the value of "num" reaches 3, the break statement is met, and the loop ends. The loop continuous to cycle over the list of integers.

"continue" – By using the "continue" statement, you may skip the remaining iterations of the loop and go straight to the next one. It is useful in situations where you wish to have not handled a certain item.
numbers = [1,2,3,4,5]

for num in numbers;
if num == 3;
continue
print(num)
The "continue" statement is met and the current iteration is ignored when the value of "numb" is 3. Printing the leftover integers on the next iterations of the loop.

Functions

2.1 Introduction

In Python, a function is a reusable unit of code that completes an action. It is a fundamental idea in programming that encourages organization, modularity, and reusability. Code becomes more understandable and manageable when you use functions to group a group of instructions together under a single name.

The "def" keyword is used to define a function, which is then followed by the function name and any necessary parameters in a pair of brackets. The function will use the values in these arguments as placeholders. You put the code for the function's behaviour, such as calculations, conditionals, loops, and more, inside the function's indented block. Functions can take as many input arguments as they need and return values. The function's caller receives a result from the function using the "return" statement. You may use this to do out calculations or transformations and get the results for later use.

The use of functions encourages the reuse of code since you may use the same function again inside one program or across numerous programs, minimizing duplication and simplifying maintenance. Functions improve the readability and maintainability of code by dividing complicated tasks into smaller, easier-to-manage components.

2.2 Define a Function

A function in Python is a named section of code that

brackets to define a function. Within a block, the logic of the function is indented.
For Example:

```
def greet(name): """Hello, How are you..???"""
print(f"Hello, {name}!")

greet("Pawan")
greet("Preet")
```

The "greet" function prints a greeting and receives a "name" input. You may reuse the code for multiple welcomes by invoking the function with different names.

2.3 Calling a Function

In Python, calling a function requires entering the name of the function, followed by brackets, and optionally, arguments that correspond to the parameters of the function. The code included in a function's block is executed when it is invoked. For Example:

```
def add(a, b): """Sum of Two"""
return a + b

# Calling the function and storing the result
result = add(3, 5)
print(result) # Output: 8
```

The "add" function takes two arguments, adds them, and returns the result.

2.4 Returning Value

The "return" statement in Python allows functions to return values. Using the return keyword, a function that has been called and has completed its code block may "return" a result to the caller. The caller code might then

use this returned value for additional calculations, assignments, or other uses. For Example:

```
def multiply(a, b): """Product of two Numbers."""
result = a * b
return result

# Calling the function and using the returned value
product = multiply(4, 7)
print(product) # Output: 28

# Using the returned value directly
print(multiply(3, 9)) # Output: 27
```

The "multiply" function determines the outcome of the product of two integers. The returned value from the function call can either be utilized directly in additional operations "(print(multiply(3, 9))" or assigned to a variable "product".

2.5 Default Arguments

When a function is defined in Python, parameters can have default arguments, or predetermined values, provided to them. This enables you to use the default values when calling the function rather than individually providing values for those arguments. The flexibility and usefulness of functions are improved by default parameters. For Example:

```
def greet(name, greeting="Hello"):"""Greet With Person Name."""
print(f"{greeting}, {name}!")

# Calling the function with both arguments
```

greet("Pawan", "Hi") # Output: Hi, Pawan

Calling the function with only one argument
greet("Preet") # Output: Hello, Preet
The default option for "greeting" in the "greet" function is "Hello". The default "greeting" is used when the function is invoked with just one argument ("name"). Second arguments overrule the default if they are given. This gives you the ability to greet someone with many texts by utilizing a single function. Function calls are made easier and handle frequent use situations with default parameters, which results in more understandable and clear code.

2.16 Variable Scope

In Python, the term "variable scope" describes how easily accessible and visible variables are throughout the code. It establishes the location at which a variable may be used, changed, or assigned. The two primary forms of variable scope in Python are local and global.
Variables declared inside a function are considered to be "local scope". These variables can only be accessed from the function block. When the function is invoked, they are generated, and when the function is finished, they vanish.
Variables declared outside of any function are said to have a "global scope". These variables are accessible from anywhere in the code and may be changed from both within and outside of functions.
For Example:
global_var = 10 -> Global variable

```
def my_function():
local_var = 5 -> Local variable
print(global_var) -> Accessing global variable
print(local_var) -> Accessing local variable

my_function()
print(global_var) -> Accessing global variable outside
function

-> This would result in an error as local_var is not
accessible here
# print(local_var)
```

It is possible to access "global_var" both within and outside the function. "local_var" is a local variable that can only be accessed inside the "my_function" function. It would be incorrect to try to access "local_var" outside of the function.

Modules and Packages

3.1 Introduction

Organizing and structuring code in Python to improve code reuse, maintainability, and cooperation in bigger projects, modules, and packages are important ideas.

Modules

A single file containing Python code, which might include functions, classes, and variables, is known as a module in the language. It functions as an independent chunk of code that may be imported and applied in other applications. In order to make a program simpler to understand and maintain, modules help divide it into more manageable, smaller parts.

Packages

Associated modules arranged in a directory structure make up a package. It offers a method to combine comparable features, making it simpler to maintain and distribute code across many files.
Packages provide programmers the ability to organize their codebase logically, enabling advanced organization and concern separation. This is especially helpful for bigger projects where several modules must work in concert together.

3.2 Creating and Using Module

An effective method for organizing and reusing code across various sections of your program or even across

into smaller, more manageable parts.

Creating a Module

Simply write your Python code in a different ".py" file to build a module. Let's imagine you wish to develop a module named "math_utils" that contains some fundamental mathematical operations. You might write add, subtract, and multiply definitions in a file called "math_utils.py". For Example:

```
# math_utils.py

def add(x, y):
return x + y

def subtract(x, y):
return x - y
def multiply(x, y):
return x * y
```

Using a Module

Once a module has been developed, you may import it into other Python scripts or modules to use it. Suppose you wish to utilize the "math_utils" module in a different script named "main.py." You may import the module and utilize its features in the following way:

```
# main.py
import math_utils

result1 = math_utils.add(5, 3)
print(result1) # Output: 8
```

```
result2 = math_utils.subtract(10, 4)
print(result2) # Output: 6
```

Alternatively, you can import specific functions from the module using the, from ... import ... syntax:

```
# main.py
from math_utils import multiply

result = multiply(7, 2)
print(result) # Output: 14
```

3.3 Creating and Using Packages

When you organize and structure your code into logical directories using Python packages, you improve the organization, reuse, and maintenance of your project. Packages are groups of similar modules that have been grouped under a single namespace.

Create a Package

To begin, make a directory that will act as the package's root. Make a subfolder for each significant part of your product inside this directory. Your package's submodules will be created from each subfolder. For Example:

```
my_package/
|-- __init__.py
|-- module1.py
|-- module2.py
|-- subpackage1/
|  |-- __init__.py
|  |-- module3.py
```

|-- subpackage2/
| |-- __init__.py
| |-- module4.py

Writing the __init__.py : - Even if a directory is empty, it must have a "__init__.py" file. When a package or submodule is imported, this file is run and may include initialization code, variable definitions, or import statements that will be used when the package is used.

Using the Package
You may use dot notation to import the required modules or submodules into other Python scripts or modules to use your package. For example:

```
# Importing modules from a package
import my_package.module1
from my_package.subpackage1 import module3

# Using the imported modules
my_package.module1.function_name()
module3.some_function()
```

3.4 Namespace and "__name__"
A namespace is a collection of identifiers that links them to the associated objects (variable names, function names, class names, etc.). Namespaces give a program's entities a method to be organized and distinguished while also preventing name conflicts. Names serve as keys in each namespace's equivalent of a dictionary, while objects serve as values.

The name of the current module or script is stored in a special variable called "__name__" in Python. The "__name__" property of a Python script is set to "__main__" when it is executed. When determining if a Python file is being used as the primary program or imported as a module into another program, this characteristic is especially helpful. You can only conditionally execute certain code by looking at the value of "__name__" when the script is run directly, not when it is imported as a module.

For Example:

```
# module.py
def some_function():
print("Hello from the module!")

if __name__ == "__main__":
print("This will only execute when module.py is run directly.")
```

Both messages will be printed when you execute "module.py" directly. However, the code beneath (if __name__ == "__main__":) won't run if "module.py" is imported into another script, preventing unwanted behaviour.

3.5 Sub Packages

Code can be further organized and structured within a bigger package using sub-packages. Within the main package, a nested hierarchy is created by sub-packages, which are subdirectories that each contains their own

collection of modules. This hierarchical structure improves administration, reuse, and concern separation in increasingly complicated projects.
Subpackages utilize __init__.py files inside each subpackage directory and adhere to the same rules as packages. When the subpackage is imported, these files may include an initialization code or import instructions that make the items within the subpackage available.
For example, consider a main package named "utils" with two subpackages, "math" and "string," each containing their respective modules:

```
utils/
|-- __init__.py
|-- math/
| |-- __init__.py
| |-- calculations.py
|-- string/
| |-- __init__.py
| |-- manipulation.py
```

To access elements from subpackages, you use dot notation:

```
# Importing modules from subpackages
import utils.math.calculations
from utils.string import manipulation

# Using the imported modules
utils.math.calculations.add(5, 3)
manipulation.capitalize("hello")
```

3.6 Module Aliases

Python's module aliases feature lets you give imported modules other names that are either shorter or more descriptive. This can lessen typing, improve the readability of the code, and prevent naming conflicts. To create an alias for a module during import, you use the as keyword followed by the desired alias name. For example:

```
# Using 'm' as an alias for the 'math' module
import math as m
result = m.sqrt(25)
```

You can also use aliases for specific components within a module:

```
# Using 'dt' as an alias for 'datetime'
from datetime import datetime as dt
current_time = dt.now()
```

3.7 Standard Library or Third-Party Module

Both the standard library and third-party modules are essential to Python's ability to expand its functionality and streamline program development.

Standard Library

A selection of modules and packages called the standard library are supplied with Python installs. It offers a wide range of features, including regular expressions (re), string manipulation, mathematical operations (math), file handling (os, io), and more. By providing tools for common tasks, these modules let developers produce effective code without having to constantly discover new

techniques. All Python contexts and versions provide a uniform user experience thanks to the standard library. For example, you may use the datetime module from the standard library to interact with dates and times:

```
import datetime
current_time = datetime.datetime.now()
print(current_time)
```

Third-Party Module

Developers outside the Python core development team produce third-party modules. They target particular requirements or domains, extending Python's capability beyond what the standard library provides. To save time and effort, developers use third-party modules, using pre-built solutions for specific tasks. These modules cover a wide range of topics, such as machine learning (scikit-learn, TensorFlow), data analysis (pandas, numpy), and web programming (requests, Flask).

Third-party modules must often be installed using package management tools like "pip" before being used. For example, to set up the well-liked HTTP requests module:

```
pip install requests
```

Then, you can use the module in your code:

```
import requests
response = requests.get("https://www.example.com")
print(response.status_code)
```

Important: -

Python's flexibility and capacity for quick development are enhanced by both the standard library and third-

modules. Developers may produce effective, feature-rich, and maintainable software solutions that are suited to particular needs by using these resources. Python's applicability for a variety of application domains is improved by the availability of a large selection of modules, which also supports a strong ecosystem of tools and libraries.

Data Structures

4.1 Introduction

In order to support effective manipulation, storage, and retrieval of data, data structures are essential constructions in computer science that organize and store data in a certain fashion. They act as the building blocks for creating and putting into practice algorithms, enabling efficient information management within a computer program.

Data structures offer a methodical technique to organise different operations-optimizing data items, such as numbers, language, or complicated objects. Regarding access speed, memory utilisation, and ease of change, various data structures provide a variety of trade-offs. Hash tables, trees, graphs, queues, stacks, lists, and arrays are a few examples of typical data structures which we will learn in the Chapter of Advance Data Structures.

The particular issue at hand and the kinds of operations that must be often carried out determine the data structure to be used. For example, hash tables excel in speedy data retrieval based on keys, whereas arrays are great for quick indexing and sequential access. Trees are utilized to retain hierarchical relationships, whereas linked lists are best for frequent insertions and removals.

4.2 Lists

A list is a functional and basic data structure that lets you organize and store collections of objects. Lists are ordered series that can include components of many

by the language, lists are essential to many programming tasks.

The process of making a list is simple. You use square brackets to surround a list of values that are separated by commas. For example:
my_list = [1, 2, 3, 'hello', True]

Since lists are zero-indexed, you can access elements by utilizing the locations of the index keys. For the first element, indexing begins at 0, for the second at 1, and so on. The list's end is where negative indices start counting.
For Example:
print(my_list[0]) # Outputs: 1
print(my_list[-1]) # Outputs: True

Lists support various operations, including:
Append: - Add an element to the end of the list.
my_list.append('new element')

Insert: - Insert an element at a specific index.
my_list.insert(2, 'inserted')

Remove: - Remove the first occurrence of a specific value.
my_list.remove(2)

Pop: - Remove and return an element at a given index. If no index is provided, the last element is removed.
popped_element = my_list.pop(1)

Len: - Get the number of elements in the list.
length = len(my_list)

Concatenation: - Combine two or more lists.
new_list = my_list + [4, 5, 6]

Slicing: - Extract a portion of the list using a start and end index.
sub_list = my_list[1:4] # Includes elements at indices 1, 2, and 3

4.3 Tuples
An ordered and immutable group of elements is represented by a data structure called a tuple. Lists and tuples are similar, yet there are some significant distinctions. Their immutability makes them useful for situations where you want to guarantee that the data remains intact throughout its existence. They are frequently used to group relevant data together.
A comma-separated list of values enclosed in brackets forms a tuple. For example:

my_tuple = (1, 2, 'hello', 3.14)

Like lists, tuples are indexed, allowing you to access elements by their position:
print(my_tuple[0]) # Outputs: 1
print(my_tuple[2]) # Outputs: 'hello'

Tuples, however, cannot have their elements changed once they are created since they are immutable. To maintain the accuracy of the data within the tuple, this

immutability is required. Individual members cannot be changed, however the full tuple can be changed:
my_tuple = (4, 5, 6)

Tuples are often used in various contexts, such as:
Function return Value: - Functions can return multiple values as a tuple, which can then be unpacked by the caller.
def get_coordinates():
return 2, 3

x, y = get_coordinates()

Data Grouping: - Tuples can be used to group related data, creating a compact way to represent records.
person = ('Pawan', 20, 'Book Writer')

Dictionary Keys: - Since tuples are immutable, they can be used as keys in dictionaries, whereas lists cannot.
my_dict = {('Pawan', 20): 'Book Writer'}

Function Arguments: - Tuples can be used to pass multiple arguments to a function as a single argument.
def process_data(*args):
print(args)

process_data(1, 2, 3)

Multiple Assignments: - Tuples allow you to assign multiple variables in a single line.
a, b, c = 1, 2, 3

4.4 Dictionaries

A dictionary is a strong and flexible data structure that enables key-value pair storage and retrieval of data. In certain computer languages, dictionaries are also referred to as hash maps or associative arrays. They offer a mechanism to depict the connections between different data sets and are especially effective for quick data retrieval based on unique keys.

Curly braces are used to form dictionaries, which are lists of comma-separated key-value pairs containing a key for each value. For example:

```
my_dict = {'name': 'Pawan', 'age': 20, 'occupation': 'Book Writer'}
```

A dictionary's keys must be distinct, and while they are frequently strings or integers, other immutable types may also be utilised. Any data type, including lists, tuples, other dictionaries, and even functions, may be used as a value.

You can access values in a dictionary using their associated keys:

```
print(my_dict['name']) # Outputs: 'Pawan'
print(my_dict['age']) # Outputs: 20
```

Dictionaries support various operations, including:

Adding and Modifying Entries: - You can add new key-value pairs or modify existing ones.

```
my_dict['city'] = 'India'
my_dict['age'] = 20
```

Removing Entries:- You can delete entries using the "del"

statement.
del my_dict['occupation']

Dictionary Methods: - Python provides methods like "keys()", "values()", and "items()" to retrieve various aspects of the dictionary.

Dictionary Comprehensions: - Similar to list comprehensions, you can create dictionaries using comprehensions for concise and efficient creation.
squared_dict = {x: x**2 for x in range(5)}

4.5 Sets
A set is a basic data structure made to hold an unordered group of distinct items. When you need to effectively execute operations like membership checking, intersection, union, and difference, sets are very helpful. They offer a simple method for controlling data gathering while guaranteeing that each piece only appears once.

Creating a set is simple – you enclose a comma-separated sequence of values within curly braces {}. For example:
my_set = {1, 2, 3, 4, 5}
Sets make sure that only unique data are saved by automatically removing duplicate values. A duplicate value won't be added to the set if you try to do so.

Sets support various set operations, including:
Adding Elements: - You can add elements to a set using the "add()" method.
my_set.add(6)

Removing Elements: - Elements can be removed using the "remove()" or "discard()" methods.

```
my_set.remove(2)
```

Set Operations: - Sets can be combined using operations like union, intersection, and difference.

```
set_a = {1, 2, 3}
set_b = {3, 4, 5}
union_set = set_a | set_b
intersection_set = set_a & set_b
difference_set = set_a - set_b
```

Set Comprehensions: - Like list comprehensions, you can create sets using comprehensions for concise creation.

```
squared_set = {x ** 2 for x in range(5)}
```

4.6 List Comprehensions

By applying an expression to each element in an existing sequence (such as a list, tuple, or range), list comprehensions in Python offer a concise and effective approach to generate lists. They do away with the requirement for explicit loops by condensing the process of making a new list into a single line of code.

A list comprehension's fundamental syntax starts with an expression and ends with a loop that iterates through the sequence's components. To make a list of squares from 0 to 4, for example:

```
squared_numbers = [x ** 2 for x in range(5)]
```

Conditions may also be added to list comprehensions using an optional if clause. This enables you to filter the

items according to a circumstance. To make a list of even integers from 0 to 9, for example:
even_numbers = [x for x in range(10) if x % 2 == 0]

4.7 Nested Data Structures
Python uses the term "nested data structures" to describe the idea of nesting data structures. In order to describe advanced relationships between data pieces, you may use this to construct and explain hierarchical arrangements. Lists of lists, dictionaries of dictionaries, and combinations of other data structures are typical examples.
For example, you may have a list of dictionaries, where each one has details on a certain person:
people = [
{'name': 'Pawan', 'age': 20},
{'name': 'Preet', 'age': 21}
]
Similar to nesting lists within dictionaries or dictionaries within lists, various data structures may be used to build multi-level structures that mimic real-world situations.

Using many levels of indexing or key lookup is necessary to access inner items while navigating and working with nested data structures. For example:
print(people[0]['name']) # Outputs: 'Pawan'

4.8 Iterating Over Data Structures
When iterating through data structures in Python, each element of a collection, such as a list, tuple, dictionary,

set, or even a string, is accessed sequentially. You can process, filter, or transform the data using this procedure to conduct operations on each element.
The most common loops used to iterate over a collection are "for" and "while" loops. The "for" loop is frequently used with data structures since it takes care of the iteration process automatically. For example, to repeat a list:

```
numbers = [1, 2, 3, 4, 5]
for num in numbers:
print(num)
```

You may use methods like "items()" to iterate through keys, values, or both in dictionaries:

```
person = {'name': 'Pawan', 'age': 20}
for key, value in person.items():
print(key, value)
```

Additionally, you may iterate through the components of arranged data structures using loops. To further increase your control over the process, Python has methods like "enumerate()" that allow you to obtain both the index and the value throughout the iteration.

4.9 Sorting and Searching

For effectively organising and retrieving data from collections like lists, arrays, and more sophisticated data structures, sorting and searching are key programming techniques. Working with data is made simpler by Python's built-in functions and techniques for sorting and searching.

Sorting:- When items are sorted, they are put in a certain

order, generally ascending or descending. Python provides the "sorted()" method to order lists and other iterable objects:

```
numbers = [5, 2, 9, 1, 5]
sorted_numbers = sorted(numbers) # Outputs: [1, 2, 5, 5, 9]
```

For lists, you can also use the "sort()" method, which sorts the list in place:

```
numbers.sort()
```

Using the key parameter, you may modify the sorting, and using the reverse option, you can even reverse the order.

Searching: - Finding a particular piece inside a collection is the task of searching. Python has functions like "index()" to determine an element's position in a list:

```
numbers = [1, 2, 3, 4, 5]
index = numbers.index(3) # Outputs: 2
```

However, be cautious when using index(), as it raises an error if the element is not found.

The Python "in" operator is used to determine whether an element is present in a collection for advanced searching:

```
if 3 in numbers:
print("3 is present in the list.")
```

File Handling

5.1 Introduction

The act of reading from and writing to files on the computer's storage system is referred to as file handling in Python. Data storage and retrieval rely heavily on files, and Python offers an extensive set of tools and methods for managing files well.

File opening, reading, writing, and closing functions are included in Python. The crucial operations are "open()", "read()", "write()", and "close()". By providing the file's location and the preferred mode ('r' for reading, 'w' for writing, 'a' for appending, and more), the "open()" function can establish a connection to a file. Once a file has been opened, you may use the "read()" and "write()" functions to add or alter the file's content. The "read()" method returns the file's contents as strings or lines. The connection to the file is finally correctly closed using the "close()" method.

Various file types, including text files, binary files, and even specialised formats like CSV and JSON, are supported by Python's file handling features. Text files may be edited using common string operations and contain text that is readable by humans. For multimedia data like photos and audio, binary files are the best option since they contain information in a non-human readable manner. The handling of binary files in Python is helpful for operations like transferring files, data archiving, and processing unprocessed binary data.

function work together to make sure that files are correctly closed after usage, lowering the possibility of resource leaks. When interacting with files, this is particularly crucial since open files may behave unexpectedly and may damage data.

5.2 Reading Data From Files

Opening a file, extracting its contents, and then processing or using that data within your program is the common operation of reading data from files. To establish a reading connection to a file, use the "open()" method. Here is a detailed explanation with an example:

Opening a file: - Use the "open()" function with the file's path and mode "r" (read) to open the file for reading. For instance:

```
file_path = 'example.txt'
with open(file_path, 'r') as file:
# File handling code
```

Reading Content: - After the file has been opened, there are several ways to read its content. While "readline()" reads a single line at a time, "read()" reads the whole file's content as a string. All lines are read and returned as a list via the "readlines()" function.

```
with open(file_path, 'r') as file:
content = file.read() # Read the entire content as a string
line = file.readline() # Read a single line
lines = file.readlines() # Read all lines into a list
```

5.3 Working Data to Files

In Python, adding content to existing files or generating new ones is known as writing data to files. To open a file for writing, use the "open()" method together with the "w" (write) mode. Here is a simple explantion:

Open a file for Writing: - Use the "open()" method with the file path and the mode 'w' to start writing data to a file. The file will be generated if it doesn't already exist. If it does, previous information will be replaced.

```
file_path = 'output.txt'
with open(file_path, 'w') as file:
# Writing data to the file
```

Writing Data: - You may use the "write()" function to add content to the file after it has been opened. A string can be sent as an argument to the method, which appends it to the file.

```
with open(file_path, 'w') as file:
file.write("Hello, world!\n")
file.write("This is a new line.")
```

5.4 Using "with" Statement

Python's "with" statement makes it easier to handle resources like files that need to be correctly obtained and released. Even if exceptions arise within the block, it makes sure that resources are immediately cleaned away once they are no longer required.

The received resource is connected to a context manager by the "with" statement, which specifies how to acquire and release the resource. As a result, resource clean-up no longer requires explicit "try" and "finally" blocks.

For example, the "with" statement simplifies file management while working with files and shuts the file immediately once its block is finished:

```
with open("example.txt", "r") as file:
content = file.read()
# File is automatically closed outside the block
```

5.5 Working With Binary Files

Reading and writing data that is not in a human-readable text format is a part of working with binary files in Python. Images, audio, video, and serialized objects are just a few of the forms of data that may be included in binary files. You may properly handle and analyse such data using Python's support for handling binary files.
To work with binary files:

Opening Binary File: - To open a binary file, use the "open()" function with the proper file path and mode (for example, "rb" for reading binary and "wb" for writing binary).

```
with open("image.jpg", "rb") as file:
binary_data = file.read()
```

Reading Binary Data: - The "read()" function returns the data as a series of bytes when reading binary files. This binary data can be processed and modified to suit your needs.

Writing Binary Data: - The "write()" function expects bytes as its input for writing binary data to a file. Before writing your data, it must first be encoded into bytes.

```
binary_data = b'\x00\x01\x02\x03'
```

```
with open("output.bin", "wb") as file:
file.write(binary_data)
```

5.6 File Pointer Position

The next read or write operation will take place at the current location indicated by the file pointer position. The file pointer is a key idea in file management since it enables data navigation and manipulation inside of files. When a file is opened, the file pointer begins at the beginning and moves as data is read or written.

Here's how to work with the file pointer position using the "seek()" and "tell()" methods, along with an example:

```
with open("example.txt", "r") as file:
position = file.tell()
print("Current position:", position)
```

"tell()" Method :- The "tell()" method returns the current position of the file pointer within the file.

```
with open("example.txt", "r") as file:
position = file.tell()
print("Current position:", position)
```

"seek() Method : - The file pointer is moved to a given location within the file using the "seek(offset, whence)" technique. The "offset" denotes the number of bytes to relocate, while the "whence" specifies the offset's starting point. The three most frequent "whence" values are 0 (file start), 1 (current position), and 2 (file end).

```
with open("example.txt", "r") as file:
file.seek(10, 0) # Move to 10 bytes from the start
data = file.read()
print(data)
```

The file pointer is moved to the 10th byte from the

beginning of the file, and then the content is read from that point.

5.7 Exception Handling

The method of exception handling in Python enables you to handle and react to unknown issues or exceptions that may arise during the execution of your program. These can include runtime difficulties like division by zero or file not found, as well as syntax mistakes. Exception handling gives you a regulated approach to dealing with mistakes while also preventing your program from crashing.

The "try", "except", "else", and "finally" blocks are the core components of Python's exception handling:

"try" - This block contains the code that might raise an exception.

"except" - If an exception occurs within the "try" block, the code in the "except" block is executed. You can specify which exception types to catch.

"else" - This block is executed if no exceptions occur in the "try" block.

"finally" - This block is always executed, whether an exception occurred or not. It's used for clean-up operations.

Example:

```
try:
num = int(input("Enter a number: "))
result = 10 / num
```

```
except ZeroDivisionError:
print("Cannot divide by zero.")
except ValueError:
print("Invalid input. Please enter a number.")
else:
print("Result:", result)
finally:
print("Exception handling complete.")
```

The program tries to divide and responds to probable errors with the appropriate error messages, such as zero division or incorrect input. Whether or whether an exception arises, the "finally" block makes sure that any necessary cleaning is carried out.

5.8 CSV Files

Tabular data can be stored in plain text using the CSV (Comma-Separated Values) file format. Values are separated by commas or other delimiters inside each line of the file, which forms a row. Python has built-in libraries that can read and write CSV files quickly and effectively, making working with structured data simple. To work with CSV files in Python:

Reading CSV Files: -For reading CSV files, the "csv" module provides methods like "csv.reader()". You may retrieve data by index or column name as you traverse through the rows.

```
import csv
```

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

Writing CSV Files: - To generate and write data to CSV files, use the "csv.writer()" method. Values are separated from one another by the designated delimiter when data is written row by row.

```
import csv

data = [
["Name", "Age", "Country"],
["Pawan", 20, "Punjab"],
["Preet", 21, "Delhi"]
]
with open("output.csv", "w", newline="") as file:
writer = csv.writer(file)
writer.writerows(data)
```

5.9 JSON Files

The JSON (JavaScript Object Notation) format is a simple and popular way to convey structured data. It is a text-based format that is easily parsed and generated by both people and machines. JSON files are frequently used for data interchange between systems, configuration files, and data storage.

The "json" module provides functions to work with JSON data:

Reading JSON File: - The "json.load()" method may be

used to read data from a JSON file since it parses the JSON content and returns it as a Python data structure.

```
import json

with open("data.json", "r") as file:
data = json.load(file)
```

Writing JSON Files: - The "json.dump()" method is used to output data to a JSON file. It writes a JSON-formatted version of a Python data structure to the file.

```
import json

data = {"name": "Pawan", "age": 20, "country": "India"}
with open("output.json", "w") as file:
json.dump(data, file)
```

Object-Oriented Programming

6.1 Introduction

A strong and popular programming paradigm called object-oriented programming (OOP) requires structuring and organizing code in a more logical and modular fashion. OOP is a fundamental strategy that makes it easier to create and manage big software systems in Python, a flexible and dynamically-typed programming language.

Python's OOP is built around classes and objects. An outline or template for an object's properties (data) and methods (functions) is known as a class. The instances of these classes that make up objects, on the other hand, each have their own distinct data and the capacity to carry out operations via methods.

The basic principle of OOP in Python is encapsulation. It involves combining information and operations on that information into a single class. This encourages data hiding, the practice of concealing an object's internal information from the outside world and controlling access to the object's properties and operations through well-specified interfaces. This improves code maintainability since code that depends on a class's public interface won't be impacted by changes to the internal implementation of the class.

Another essential component of OOP in Python is inheritance. It enables a class to inherit properties and functions from another class (the superclass or base class) (also known as a subclass or derived class). As general functionality may be described in a base class

structure in complicated software systems.
The third pillar of Python's OOP is polymorphism. Through a shared interface, it enables objects of several classes to be considered as instances of a single base class. As a result, functions and methods may be created to operate with a range of objects without requiring knowledge of those objects' exact types, making programming more versatile and adaptable.

6.2 Classes and Objects

The basic components of object-oriented programming (OOP) are classes and objects. They offer a systematic, disciplined manner to describe and deal with complex data structures and behaviours. Let's explore the ideas of classes and objects in more detail with some examples.

Classes

An object's structure and behaviour are specified by a class, which serves as a blueprint or template. It includes methods (variables) that operate on those attributes as well as data attributes (variables). A class acts as a blueprint that specifies the qualities and abilities that an item of that class may possess. For Example:

```
class Dog:
def __init__(self, name, age):
self.name = name
self.age = age

def bark(self):
```

return "Wooof!"

Name and age are the two attributes we've given the "Dog" class. When a new object is formed, the "__init__" method is a unique method (constructor) that initializes the attributes. The dog's behaviour when it barks is described by the bark method.

Objects

A class's instances are objects. It is an actual representation of the blueprint that the class created. Each object has a unique collection of attribute values and is capable of carrying out actions that are defined by the class's methods. For Example:

```
dog1 = Dog("Tommy", 4)
dog2 = Dog("Jio", 6)
print(dog1.name) # Output: "Tommy"
print(dog2.age) # Output: 6

print(dog1.bark()) # Output: "Wooof!"
```

The objects "dog1" and "dog2" belong to the "Dog" class. Name and age are different property values. "Woof!" is the response that is returned when the class' method, bark, is called on "dog1".

6.3 Constructor and Destructor

To initialize and destroy objects, respectively, constructors and destructors are special procedures used in object-oriented programming (OOP). They are essential in controlling the life cycle of things. Let's investigate constructors and destructors in further detail

through explanations and examples.

Constructor

When an object of a class is formed, a constructor, a specific method, is invoked. The attributes are initialised, and the object is made ready for usage. The constructor function in Python is known as "__init__()". For Example:

```
class Person:
def __init__(self, name, age):
self.name = name
self.age = age

person1 = Person("Pawan", 20)
person2 = Person("Preet", 21)

print(person1.name) # Output: "Pawan"
print(person2.age) # Output: 21
```

The "name" and "age" arguments are sent to the "__init__()" function, which initializes the necessary Person class properties. The constructor is automatically called when 'person1' and 'person2' are formed to set their characteristics.

Destructor

When an object is ready to be destroyed or trash collected, a particular procedure called a destructor is invoked. The destructor function in Python is called "__del__()". For Example:

```
class Book:
def __init__(self, title):
self.title = title
```

```
def __del__(self):
print(f"Book '{self.title}' is being destroyed")

book1 = Book("Python Three Levels")
book2 = Book("JavaScript SooN")

del book1
```

In the Book class, the "_ _del_ _()" function is defined. When the book1 object is explicitly removed using the "del" command, the destructor is executed and an associated message is produced.

6.4 Instance and Class Attributes

In Object-Oriented Programming (OOP), instance attributes and class attributes are two different types of attributes used to specify the traits and properties of objects and classes, respectively. They are essential for organising code and simulating real-world ideas. Let's investigate instance and class characteristics using justifications and illustrations.

Instance Attribute

Each object of a class has its own unique properties called instances. They are declared in the methods of the class, frequently in the constructor ("_ _init_ _()" function), and store distinct data values for each object. An object's state is captured via its instance attributes, which enable the object to have unique features. For Example:

```
class Car:
def __init__(self, make, model):
self.make = make
```

```
self.model = model

car1 = Car("Toyota", "Camry")
car2 = Car("Honda", "Civic")

print(car1.make) # Output: "Toyota"
print(car2.model) # Output: "Civic"
```
The make and model attributes are instance attributes. Each instance of the "Car" class (car1 and car2) has its own values for these attributes.

Class Attribute

All instances of a class have the same class features. Outside of any methods, they are declared in the class itself. Class attributes indicate traits or qualities that apply to all members of the class and are constant across instances. For Example:

```
class Circle:
pi = 3.14159

def __init__(self, radius):
self.radius = radius

circle1 = Circle(5)
circle2 = Circle(7)
print(circle1.pi) # Output: 3.14159
print(circle2.radius) # Output: 7
```

Here, "pi" is a class attribute shared by all instances of the Circle class. It remains the same for all circles.

6.5 Inheritance and Polymorphism

Two essential ideas in Object-Oriented Programming (OOP)—inheritance and polymorphism—allow for the reuse, extension, and flexibility of code. These ideas allow you to design complex systems in Python by using pre-existing code and adapting behavioural differences. Let's explore polymorphism and inheritance with explanations and examples.

Inheritance

A new class (subclass or derived class) can inherit properties and methods from an existing class (superclass or base class) using the technique of inheritance. Because common properties and behaviours may be defined in the base class while specialized features or existing methods can be added by derived classes, this encourages code reuse. For Example:

```
class Animal:
def __init__(self, name):
self.name = name

def speak(self):
pass

class Dog(Animal):
def speak(self):
return "Woof!"

class Cat(Animal):
def speak(self):
return "Meow!"
```

```
dog = Dog("Tommy")
cat = Cat("wosho")

print(dog.speak()) # Output: "Woof!"
print(cat.speak()) # Output: "Meow!"
```

The "Dog" and "Cat" classes come from the "Animal" basic class. Both derived classes add their own functionality by overriding the "speak()" method and inheriting the "name" property.

Polymorphism

Through a shared interface, polymorphism enables objects of several classes to be considered as instances of a single base class. By enabling functions and methods to operate on a variety of object types without having to be aware of their precise kinds, this encourages flexibility. According to the type of the real object, polymorphism makes sure the right method is called. For Example:

```
def make_animal_speak(animal):
return animal.speak()

dog = Dog("Tommy")
cat = Cat("wosho")

print(make_animal_speak(dog)) # Output: "Woof!"
print(make_animal_speak(cat)) # Output: "Meow!"
```

The "make_animal_speak()" function uses the "speak()" method of a "Animal" object as an argument. Due to polymorphism, this method works with both "Dog" and "Cat" objects.

6.6 Encapsulation

The core idea of encapsulation in object-oriented programming (OOP) focuses on grouping together into a single entity known as a class the data (attributes) and the methods (functions) that operate on that data. This unit serves as a protective barrier around an object's internal state, preventing others from having direct access to the object's properties. Encapsulation encourages data hiding, which means that an object's implementation details are hidden and that access to its characteristics is managed by well-defined interfaces.

For Example:

Let's create a simple program of Encapsulation which is used for encapsulated attribute.

```
class BankAccount:
def __init__(self, account_number, balance):
# Encapsulated attribute
self.__account_number = account_number
self.__balance = balance # Encapsulated attribute

def get_balance(self):
return self.__balance

def deposit(self, amount):
if amount > 0:
self.__balance += amount

def withdraw(self, amount):
if 0 < amount <= self.__balance:
self.__balance -= amount

# Creating a BankAccount object
```

```
account = BankAccount("123456", 1000)

# Accessing attributes through methods
print(account.get_balance()) # Output: 1000

# Attempting to access encapsulated attribute directly (not recommended)
# print(account.__balance) # This will raise an AttributeError

# Using methods to modify encapsulated attributes
account.deposit(500)
print(account.get_balance()) # Output: 1500

account.withdraw(200)
print(account.get_balance()) # Output: 1300
```

"__account_number" and "__balance" are characteristics that are contained in the "BankAccount" class. Direct access to these characteristics is restricted to members of the class. Instead, well-defined functions like "get_balance()," "deposit()," and "withdraw()" are used to grant access. This encapsulation makes sure that only authorized actions may change the account balance, preserving its integrity.

6.7 Method Overriding

A basic concept in object-oriented programming (OOP) is method overriding, which enables a subclass to offer a particular implementation for a method that is already defined in its superclass. By doing so, the subclass can preserve the same method signature while modifying the

behaviour of the inherited method. In derived classes, method overriding makes it easier to provide more specialized behaviour, encouraging flexibility and extensibility. For Example:

```
class Animal:
def speak(self):
return "Animal speaks"

class Dog(Animal):
def speak(self):
return "Woof!"

class Cat(Animal):
def speak(self):
return "Meow!"

dog = Dog()
cat = Cat()

print(dog.speak()) # Output: "Woof!"
print(cat.speak()) # Output: "Meow!"
```

A general "speak()" function is defined by the basic class "Animal". The "Cat" and "Dog" subclasses each have their own implementations of this function that they override. A customized output ("Woof!" and "Meow!") is produced when the overridden methods of the corresponding subclasses are called on instances of "Dog" and "Cat" when the "speak()" method is called on them. This exemplifies how derived classes may replace inherited methods with their own specialized versions thanks to method overriding.

6.8 Access Modifiers

The visibility and accessibility of class members (attributes and methods) from outside the class are managed via access modifiers, sometimes referred to as access specifies, in Object-Oriented Programming (OOP) languages like Python. Public, protected, and private are the three access modifier levels offered by Python. It's important to remember that unlike some other languages, Python uses conventions rather than rigid access control.

Public Access Modifiers: - All class members are public by default, meaning they can be accessed from anywhere, both within and outside the class. For Example:

```
class Car:
def __init__(self, make, model):
self.make = make # Public attribute
self.model = model # Public attribute

def start(self): # Public method
return f"{self.make} {self.model} is starting"
```

Both the 'make' and 'model' attributes, as well as the "start()" method, are public. They can be accessed from outside the class using dot notation.

Protected Access Modifiers: - A single leading "underscore (_)" designates a protected member. It's a convention, not a rule that says certain members need to be considered as protected and accessible only by members of the class and its subclasses. For Example:

```
class Vehicle:
def __init__(self, brand):
self._brand = brand # Protected attribute

def display_brand(self): # Protected method
return self._brand

class Car(Vehicle):
def start(self):
return f"{self._brand} is starting"
```

A protected attribute is "_brand". The text suggests that this characteristic should be protected; however it does not enforce it.

Private Access Modifiers: - Members who are private are denoted by two leading "underscores (__)". This is a tradition; therefore remember that it means that certain members shouldn't be directly accessed from outside the class. For Example:

```
class BankAccount:
def __init__(self, account_number):
self.__account_number = account_number # Private
attribute

def display_account_number(self): # Private method
return self.__account_number

account = BankAccount("12345678")

# Direct access to __account_number will raise an
AttributeError
# print(account.__account_number)
```

```
# Indirect access using a method is possible
print(account.display_account_number()) # Output:
"12345678"
```

The property "__account_number" is private. We've built a function called "display_account_number()" to access it. This technique conforms to the naming mangling practice while providing indirect access to the private attribute.

6.9 Abstract Class

A class that cannot be created directly but acts as a guide for other classes is known as an abstract class. It may include abstract methods, which must be specified in its subclasses because they lack implementation. The "ABC" (Abstract Base Class) module from the "abc" module in Python may be used to construct abstract classes.

```
from abc import ABC, abstractmethod

class Shape(ABC): # Abstract class
@abstractmethod
def area(self):
pass

class Circle(Shape):
def __init__(self, radius):
self.radius = radius

def area(self):
return 3.14159 * self.radius ** 2
```

```
class Rectangle(Shape):
def __init__(self, width, height):
self.width = width
self.height = height

def area(self):
return self.width * self.height

# Creating objects
circle = Circle(5)
rectangle = Rectangle(4, 6)

print(circle.area()) # Output: 78.53975
print(rectangle.area()) # Output: 24
```

The abstract function "area()" is part of the class "Shape" in coding. The "Circle" and "Rectangle" subclasses provide practical implementations of the "area()" function and derive from "Shape." This upholds the agreement that all "Shape" subclasses must include the "area()" function.

6.10 Class Method and Static Method

The two types of methods that may be written within a class to carry out operations that are connected to the class but don't necessarily need an instance of the class are class methods and static methods. The code becomes more modular and adaptable as a result of these techniques' additional levels of organization and abstraction.

Class Method

A class method is one that is tied to the class rather than the class instance. The first parameter is often called "cls" and is the class itself. Class methods can be used to carry out actions using class attributes or offer class-specific functionality. For Example:

```
class MathOperation:
pi = 3.14159 # Class attribute

@classmethod
def circle_area(cls, radius):
return cls.pi * radius ** 2

area = MathOperation.circle_area(5)
print(area) # Output: 78.53975
```

The class method "circle_area()" belongs to the "MathOperation" category. Using the supplied radius and the "pi" attribute of the class, it determines the area of a circle. The class name (MathOperation) is used to invoke class methods rather than an instance.

Static Method

Similar to a class method, a static method is one that is connected to the class but does not accept the class or instance as an argument. However being specified within a class, it is not immediately accessible to any of the class's properties or methods. Utility functions that are connected to the class but don't need class-specific data are frequently implemented using static methods. For Example:

```
class Utility:
@staticmethod
def multiply(a, b):
```

```
return a * b

result = Utility.multiply(4, 5)
print(result) # Output: 20
```

A static function found in the "Utility" class is "multiply()". It returns its product after receiving two parameters. Static methods, in contrast to class methods, cannot access a class's properties or methods unless such arguments are explicitly passed in.

6.11 Operator Overloading

You may specify the behaviour of operators for objects of custom classes in Python by using operator overloading, commonly referred to as operator improvised polymorphism. You may modify the default behaviour of built-in operators like '+', '−', '*', and '/' by implementing custom methods within your class, enabling your objects to meaningfully interact with these operators. For Example:

```
class Vector:
def __init__(self, x, y):
self.x = x
self.y = y

def __add__(self, other):
return Vector(self.x + other.x, self.y + other.y)

def __str__(self):
return f"({self.x}, {self.y})"

v1 = Vector(1, 2)
```

v2 = Vector(3, 4)

v3 = v1 + v2
print(v3) # Output: (4, 6)
The "__add__()" special function, which is used for changing the behaviour of the "+" operator, is defined by the Vector class. The "__add__()" function is invoked when two "Vector" objects are combined using the "+" operator, returning a new "Vector" object that contains the total of the original objects' components.

Error Handling and Exceptions

7.1 Introduction

The robustness, dependability, and stability of programs are influenced by important software development principles such as exception management and error handling. Errors are inescapable in the dynamic and intricate world of programming, whether brought on by user input, hardware problems, or unanticipated circumstances. Developers use exception mechanisms and error-handling strategies to lessen the effect of these problems. These strategies allow software to gracefully manage failures, avoid crashes, and provide users with helpful feedback. We examine error handling and exception definitions, theories, and recommended practices in this post.

Understanding Errors and their Consequences

Unintended or unexpected behaviours that happen when a program is being executed are referred to as errors in software development. These can include grammatical mistakes, logical mistakes, and runtime problems like division by zero; null pointer dereferences, and network failures. Failure to fix these issues may result in program failures, data loss, weakened security, and irate users.

The Role of Error Handling

A methodical technique for controlling and resolving mistakes in software programs is known as error handling. Its main objective is to make sure the program keeps running even when mistakes occur. Anticipating

Understanding Exceptions

An occurrence that prevents a program from running normally is considered an exception. When an extraordinary circumstance occurs, the program throws an exception to show that something went wrong. A suitable exception-handling mechanism then detects and processes this exception.

Exception Handling

Modern programming languages have an essential notion called exception handling. It offers a methodical approach to handling unexpected events and runtime problems. The following are the main elements of exception handling: Throwing exceptions, Catching exceptions, and Exception Propagation.

7.2 Syntax Error VS Exceptions

While both syntax errors and exceptions are categories of problems that might happen during program execution, they are caused by different things and need various techniques for detection and repair.

Syntax Error

The Python interpreter encounters syntax errors when it detects a grammatical problem in the language. These mistakes happen when the program is digesting data before running. Misplaced punctuation, misspelled words, and improper indentation are examples of common syntax mistakes. The program won't execute unless the syntax problems are corrected since they prohibit the code from being built or interpreted. During the development stage, they are frequently found using syntax highlighting or compiler warnings. For Example:

print("Hello, World!"
The missing closing parenthesis causes a syntax error. The interpreter will point to the location of the error, making it relatively straightforward to identify and correct.

Exceptions

On the other hand, exceptions happen throughout a program's execution when a circumstance develops that obstructs the usual flow. These circumstances may be brought on by outside variables, user behaviour, or faulty internal logic. Python has a built-in exception-handling system that enables developers to foresee and skilfully handle unforeseen circumstances. For Example:

```
try:
num = int(input("Enter a number: "))
result = 10 / num
print("Result:", result)
except ZeroDivisionError:
print("Error: Cannot divide by zero.")
except ValueError:
print("Error: Please enter a valid number.")
```

The computer program tries to divide 10 by a user-supplied integer. A "ZeroDivisionError" exception is triggered if the user enters zero, and the first except block is then invoked to address the error. A "ValueError" exception is thrown and the second "except" block is triggered if the user inputs a non-numeric value.

7.3 “try” – “except” Blocks

The "try" and "except" blocks are crucial building blocks for handling exceptions and dealing with unexpected events while a program is running. Developers may use

this approach to create code that can manage exceptions and stop programs from crashing.

How “try” – “except” Blocks

The code that can cause exceptions is contained within the "try" section. The program's control is instantly moved to the matching "except" block if an exception occurs within the "try" block. This enables developers to specify precise actions to be taken in response to particular exceptions. For Example:

```
try:
num = int(input("Enter a number: "))
result = 10 / num
print("Result:", result)
except ZeroDivisionError:
print("Error: Cannot divide by zero.")
except ValueError:
print("Error: Please enter a valid number.")
```

The "try" block makes an effort to accept user input, divide, and report the outcome. A "ZeroDivisionError" happens if the user enters zero, and the program goes to the first "except" block, producing an error message. A "ValueError" is raised if the user provides a value that is not numeric, and the second "except" block deals with the problem.

7.4 Handling Multiple Exceptions

It's essential to effectively handle several exceptions while writing robust and dependable programs. Python's exception-handling system makes it possible for programmers to catch and handle several exception

types with just one "try" block, improving the readability and maintainability of their code.

Using Multiple except Blocks

Python enables you to handle different exception kinds individually by allowing you to use numerous "except" blocks inside of a single "try" block. Developers can offer specialized replies for various error conditions using this method.

For Example:

```
try:
num = int(input("Enter a number: "))
result = 10 / num
print("Result:", result)
except ZeroDivisionError:
print("Error: Cannot divide by zero.")
except ValueError:
print("Error: Please enter a valid number.")
except Exception as e:
print("An unexpected error occurred:", e)
```

The "try" block makes an effort to accept user input, divide, and report the outcome. Specific exceptions like "ZeroDivisionError" and "ValueError" are handled by the first two "except" blocks with unique error messages. Any other exceptions that could occur are caught in the last "except" block, which also outputs a general error message and the exception's description.

7.5 “else” and “finally” Clauses

The "try-except" block may be expanded with the "else" and "finally" clauses in addition to allowing for the capturing and handling of exceptions. These clauses

provide more flexibility and control when it comes to controlling code execution in error-handling contexts.

“else” Clauses

If no exceptions are thrown within the "try" block, the "else" clause, which is used after the "try" and "except" blocks, is then performed. It is helpful when you wish to carry out specific tasks only in the event that the code included in the "try" block is successful. For example:

```
try:
num = int(input("Enter a number: "))
result = 10 / num
except ZeroDivisionError:
print("Error: Cannot divide by zero.")
else:
print("Result:", result)
```

The "try" block's function runs successfully if the user provides a valid number. The "else" block's code, which prints the division result, will likewise be performed because no exceptions are triggered.

“finally” Clauses

After the "try" and "except" blocks, the "finally" clause is used, and regardless of whether an exception occurred or not, it is performed. This is helpful for actions like resource release, file closure, or operation completion.

For Example:

```
try:
file = open("data.txt", "r")
# perform operations on the file
except FileNotFoundError:
print("Error: File not found.")
```

```
finally:
file.close()
print("File closed.")
```

Whether an exception occurs or not, the "finally" block will be executed. It ensures that the file is properly closed, preventing resource leaks.

Combining "else" and "finally" Clauses

Both the "else" and "finally" clauses can be used together to achieve comprehensive error handling. For Example:

```
try:
# code that may raise exceptions
except SomeException:
# handling code for SomeException
else:
# code to execute if no exceptions occur
finally:
# code to execute regardless of exceptions
```

7.6 Raising Exceptions

When specific criteria are satisfied, the programmer has the option of directly raising an exception in addition to being able to catch and manage them. This feature enables developers to manage the error handling process and design unique error situations.

Raising Exception using "raise"

To manually raise exceptions, use the "raise" keyword. Developers can select the kind of exception and, if desired, provide a message or other details. For Example:

```
def divide(a, b):
if b == 0:
```

```
raise ZeroDivisionError("Cannot divide by zero")
return a / b
try:
result = divide(10, 0)
except ZeroDivisionError as e:
print("Error:", e)
```

If the denominator is 0, the "divide" function throws a "ZeroDivisionError" error. The program switches to the "except" block to deal with the exception when the function is invoked with "divide(10, 0)".

7.7 Custom Exception Classes

Custom exception classes provide programmers the power to design specialized error kinds that suit the particular requirements of their applications. These customised exceptions provide better mistake detection, code organization, and user experience.

Creating Custom Exception Class

To create a custom exception class, developers typically create a new class that inherits from Python's built-in "Exception" class or its subclasses.

For Example:

```
class InvalidInputError(Exception):
def __init__(self, message="Invalid input"):
self.message = message
super().__init__(self.message
def process_data(data):
if not data:
raise InvalidInputError("Empty data received")
```

try:
data = None
process_data(data)
except InvalidInputError as e:
print("Custom error occurred:", e)
Iterating from the basic "Exception" class yields the "InvalidInputError" class. The error message may be altered by using the "_ _init_ _" function. When empty data is received, the "process_data" method raises this specific exception.

7.8 Handling Unhandled Exceptions
Unexpected behaviour and program crashes might result from unhandled exceptions. Developers may, however, put mechanisms in place to gracefully manage these issues, avoiding interruptions and giving consumers useful feedback.
Global Exception with “sys.excepthook”
The "sys.excepthook" method in Python offers a way to catch unhandled exceptions. Developers can specify how unhandled errors are reported or handled globally by specifying a custom exception handler. For Example:
import sys

def custom_exception_handler(exc_type, exc_value, exc_traceback):
print(f"Unhandled exception: {exc_type}: {exc_value}")

sys.excepthook = custom_exception_handler

def calculate_division(a, b):
return a / b

result = calculate_division(10, 0)
The "sys.excepthook" is given the "custom_exception_handler" function. The "calculate_division" function's division by zero causes an unhandled exception, which is subsequently captured and handled by the custom handler, outputting a helpful message.

7.9 “assert” Statement
The "assert statement" is a helpful tool for checking programming assumptions. It enables programmers to add checks to their code to make sure certain criteria are met. An "AssertionError" exception is produced whenever an assertion fails, signifying that something unexpected happened.
Usage of “assert” Statements
The “assert” statement takes an expression that should evaluate to “True”. If the expression evaluates to “False”, the assertion fails, and an exception is raised. For Example:

```
def divide(a, b):
assert b != 0, "Denominator cannot be zero"
return a / b

try:
result = divide(10, 0)
except AssertionError as e:
print("Assertion error:", e)
```

Before completing the division, the "divide" function utilises a "assert" statement to verify that the denominator is not zero. The assertion fails in this instance because the denominator is in fact 0, resulting

in an "AssertionError" that is addressed in the "except" block.

7.10 Exception Chaining

Python's exception-chaining feature enables programmers to keep a detailed log of all the exceptions that are thrown while the program is running. When exceptions occur within exception-handling blocks, it makes debugging and comprehending the series of actions that led to an error easier.

Using Exception Chaining

The process of raising a new exception while keeping the context of the previous exception is known as exception chaining. The "from" term, which denotes that the current exception is brought about by the previous exception, is utilized to do this. For Example:

```
try:
num = int("abc")
except ValueError as ve:
raise RuntimeError("Error converting string to integer")
from ve
```

A "ValueError" is raised while trying to convert the text "abc" to an integer. The "from" keyword is used to chain the original "ValueError" to a new "RuntimeError" raised by the code in the "except" block. This keeps the two exceptions connected.

Function Programming

8.1 Introduction

A programming paradigm known as "functional programming" places an emphasis on using functions as the fundamental units of software construction. Functional programming is cantered on the idea of seeing computing as the evaluation of mathematical functions, in contrast to imperative programming, which places emphasis on statements that alter the state of a program. This paradigm promotes immutability, declarative programming, and higher-order functions, which eventually results in code that is shorter, more modular, and easier to maintain.

Python is a flexible and dynamic programming language that supports both object-oriented and functional programming paradigms. Python's functional features give developers a strong and expressive means to build clear and effective code, even though they are not entirely functional.

First-class functions are the cornerstone of functional programming in Python. Functions can be supplied as arguments to other functions, returned as values from functions, and assigned to variables in Python since they are first-class citizens. As a result, it is possible to create higher-order functions—that is, functions that interacts with or returns other functions. Developers may create more abstract and reusable code because of this freedom.

Another key idea in functional programming is

immutable. Immutable objects cannot be changed after creation, which aids in the avoidance of unanticipated effects and facilitates the reasoning process in programming.

The usage of higher-order functions like "map," "filter," and "reduce" is supported by functional programming. By eliminating common repetition, filtering, and transformation patterns, these methods enable developers to define actions on collections clearly. The "filter" function chooses items based on a condition, the "map" function applies a certain function to each element in a collection, and the "reduce" function combines elements using a specific operation.
Functional programming concepts are adhered to by Python's support for lambda functions, sometimes referred to as anonymous functions. When providing straightforward actions to higher-order functions, lambdas' ability to build short, temporary functions inline might be handy.
List comprehensions and generator expressions in Python make it possible to generate new collections quickly and expressively by applying current ones to new ones. These features encourage functional coding by putting more emphasis on data transformation than on explicit loops and mutations.
8.2 Pure Function
In Python, a "pure function" is a function that consistently returns the same result for a set of inputs and has no side effects, i.e., it doesn't change any external variables or state. The function does not result in any observable changes outside of its domain, and the

outcome exclusively depends on the input. Pure functions are easy to learn, test, and reason about because of their predictability and lack of side effects. For example:

```
def square(x):
return x * x
```

When given an "x" as an input, the "square" function returns the square of that input. It never modifies any external variables or state; instead, it continually delivers the same outcome for the same input. No matter when or where it is called, the result of "square(5)" is always "25".

Pure functions can be more readily parallelized, memoized (caching outcomes for the same inputs), and optimized since they don't depend on or change external state. Additionally, because they don't introduce unforeseen modifications or hidden dependencies, they increase the stability of the code. Contrast this with an impure function:

```
total = 0
def add_to_total(x):
global total
total += x
return total
```

The "add_to_total" function uses a global variable in addition to changing the internal state of "total". Due to its output being dependent on an external state and having side effects from changing that state, this renders the function impure. Therefore, running "add_to_total(3)" twice in distinct program states might result in different outcomes, making it more challenging to reason about and test.

8.3 Immutability

Python uses the term "immutability" to describe the property of things that cannot be changed after they are created. Any effort to alter an immutable object leads to the production of a new object, and they maintain their original value the whole time they exist. Code that adheres to this idea is more stable, predictable, and reliable because it avoids unexpected shifts and side effects. For Example:

name = "Pawan"

The string value "Pawan" is set to the variable "name". Python's immutability means that if you do an action that appears to change the text, like:

new_name = name.upper()

The "upper()" function only wraps the existing string. Instead, a new string is created with all the characters in uppercase. The initial string "Pawan" is still present. Immutability makes sure that this action won't have an impact on other sections of the code that used the original "name" variable.

Python supports immutable objects like integers, strings, tuples, and frozen sets. Since individual members of composite objects like tuples cannot be changed after the object has been constructed, this idea also applies to them. For example:

point = (3, 5)

new_point = point + (1, 2)

A new tuple called "new_point," which includes items from "point" and (1, 2), is created by the addition operation. Since the initial tuple (3, 5) is still valid, the immutability criteria are fulfilled.

8.4 HOC (Higher-Order Function)

A function that either accepts one or more functions as inputs or produces another function is referred to as a higher-order function. By recognizing functions as first-class citizens, this fundamental idea in functional programming enables the development of more abstract, adaptable, and reusable code. For example:

```
def apply_operation(func, x, y):
return func(x, y)

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

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

result1 = apply_operation(add, 5, 3)
result2 = apply_operation(subtract, 10, 4)
```

Given that it accepts the other function "func" as an input and applies it to the supplied parameters "x" and "y," the "apply_operation" function is a higher-order function. As a consequence, you may send several functions (such as add or subtract) to "apply_operation" and get various outcomes depending on the operation.

Another common example of a higher-order function in Python is the built-in 'map' function:

```
numbers = [1, 2, 3, 4, 5]

def square(x):
return x * x
```

squared_numbers = map(square, numbers)
The higher-order function "map" applies the function "square" to each member of an iterable called "numbers" and then returns an iterator of the results. Each integer in the list is squared in this instance.

8.5 Lambda Function
An anonymous function, commonly referred to as a lambda function, is an easy way to construct short, one-line functions without the formality of a 'def' declaration. When you need a quick and easy function but don't want to use the 'def' keyword to define a whole function, lambda functions come in useful. For Example:
square = lambda x: x * x
result = square(5)
The square of the input 'x' is calculated via the lambda function "lambda x: x * x". Now that the "square" variable has a pointer to this lambda function, calling "square(5)" yields the value "25".

Higher-order functions like "map," "filter," and "sorted" are frequently used in combination with lambda functions. For instance, to perform an action on each member of an iterable, you may use a lambda function with "map":
numbers = [1, 2, 3, 4, 5]
squared_numbers = map(lambda x: x * x, numbers)
The lambda function is used to square each number in the 'numbers' list, producing a new iterator containing the squared values.

Important: -

It's important to remember that named functions created with "def" may often handle situations where lambda functions cannot. However, when used wisely, lambda functions may increase code readability and maintainability, especially in circumstances when a brief, inline function is preferable to a full function specification.

8.6 Map, Filter, Reduces

Map, Filter, and Reduce are three strong functions in Python that are crucial for functional programming. With the help of these functions, you can, in turn, perform operations to collections; conditionally filter items, and aggregate values. You may adhere to functional programming concepts and produce clear, expressive code by using these functions.

Map: Transforming Elements

An iterator holding the results is returned by the "map" function after applying a specified function to each member of an iterable (such as a list). It is very helpful when you need to perform a specific operation on each element in a collection. For example:

numbers = [1, 2, 3, 4, 5]

def square(x):
return x * x
squared_numbers = map(square, numbers)

Each item in the "numbers" list is given the "square" function, which generates an iterator called "squared_numbers" that contains the squared values [1, 4, 9, 16, 25].

Filtering: Selecting Elements
By choosing components from an existing iterable that meet the criteria defined by a function, the "Filter" function creates a new iterable. It is a useful tool for extracting items that satisfy particular requirements. For Example:

```
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

def is_even(x):
return x % 2 == 0

even_numbers = filter(is_even, numbers)
```

A number's equality can be determined using the "is_even" function. The "filter" function then chooses an even number iterator [2, 4, 6, 8, 10] from the initial list and returns it.

Reduce: Aggregating Values
The "reduce" function, found in the "unctools" module, successively applies a binary function to each element of a sequence, thus condensing it to a single result. For tasks like summarising, producing, or determining the maximum/minimum, it is helpful. For example:

```
from functools import reduce

numbers = [1, 2, 3, 4, 5]

def multiply(x, y):
return x * y

product = reduce(multiply, numbers)
```

The "multiply" function is applied to pairs of elements in

the "numbers" list, resulting in the product of all elements: 1 * 2 * 3 * 4 * 5 = 120.
Important: -
While strong tools, map, filter, and reduce may frequently be substituted with more understandable and Pythonic list comprehensions or generator expressions. For instance, list comprehensions are frequently chosen because of their clarity and readability. However, since map, filter, and reduce correspond nicely with the ideas of functional programming and are helpful in more complicated situations, learning and being at ease with them is still helpful.

8.7 Closures
Python has a strong feature called closures that enables functions to keep their creation environment even after the outer function has done running. When a variable from a nested function's containing (enclosing) function is referenced, a closure is established. As a result, even after the outer function's execution is complete, the inner function can continue to access those variables and arguments. For Example:

```
def outer_function(x):
def inner_function(y):
return x + y
return inner_function

closure = outer_function(10)
result = closure(5) # Returns 15
```

The inner function "inner_function" is defined by "outer_function" and uses the input "x" in its computation. The "inner_function" is returned as a

closure when "outer_function(10)" is invoked. Even after "outer_function" has done running, this closure still allows access to the value of "x," which is 10. The outcome of calling "closure(5)" is "10 + 5", which gives the number 15.

Important: -
When using closures, it's important to exercise caution since if they're not understood correctly, they might result in unexpected behaviour. Closures provide the ability to collect and keep references to variables, which if improperly managed might result in erroneous behaviour or unnecessary memory use.

8.8 Decorators
Python's decorators are a strong and versatile feature that let you change or improve a function's or method's behaviour without altering the source code. In essence, decorators are functions that wrap over other functions, allowing you to extend the original function's capabilities or behaviour. They support code reuse and maintainability by being extensively utilized for activities like logging, authentication, memoization, and more. For Example:

```
def uppercase_decorator(func):
def wrapper(*args, **kwargs):
result = func(*args, **kwargs)
return result.upper()
return wrapper

@uppercase_decorator
def greet(name):
```

return f"Hello, {name}!"

greeting = greet("Pawan") # Returns "HELLO, PAWAN!"
The decorator function "uppercase_decorator" accepts the input "func" from another function. It constructs an inner "wrapper" function that calls the original "func" and raises the case of the outcome. The decorator is used to decorate the greet function using the "@uppercase_decorator" syntax. When greet("Pawan") is used, the greeting is transformed to uppercase after going through the "uppercase_decorator"

Decorators can also take arguments, enabling you to customize their behavior based on different parameters:
def repeat_decorator(n):
def decorator(func):
def wrapper(*args, **kwargs):
results = [func(*args, **kwargs) for _ in range(n)]
return results
return wrapper
return decorator

@repeat_decorator(n=3)
def roll_dice():
import random
return random.randint(1, 6)

dice_rolls = roll_dice() # Returns a list of three random dice rolls
The decorator factory "repeat_decorator" accepts the input "n" and outputs a decorator function. This decorator generates a list of results by iteratively calling

the decorated function "n" times.

8.9 Generators

Python offers a simple and memory-saving method for building iterators called generators. Instead of processing and storing all of the values at once, they enable you to produce values instantly as needed. Since it is impossible to retain all values in memory when processing big datasets or infinite sequences, generators are very helpful in these situations. For Example:

```
def fibonacci_generator():
a, b = 0, 1
while True:
yield a
a, b = b, a + b

fibonacci = fibonacci_generator()
for _ in range(10):
print(next(fibonacci))
```

Fibonacci numbers are defined in an infinite series using the "fibonacci_generator" function. It utilizes the "yield" keyword to generate one value at a time rather than computing and storing the complete series. The first 10 Fibonacci numbers are then printed once the "fibonacci" generator has been iterated through using a loop.

Advanced-Data Structures

9.1 Introduction

Python's advanced data structures are specialized containers made to store, handle, and process data effectively for a variety of challenging computational tasks. Although simple data structures like lists, tuples, sets, and dictionaries give the bare minimum in terms of storage and retrieval, advanced data structures enable more complex functionality to meet certain programming difficulties. These structures are essential for lowering immediate complexity, improving readability, and optimizing algorithms.

The Heap, a binary tree-based structure utilized for effective priority queue operations, is one example of an advanced data structure. Heaps can be implemented as min-heaps or max-heaps, depending on whether the parent node is bigger or smaller than its offspring. Heaps are advantageous for tasks like scheduling, graph algorithms (Dijkstra's algorithm), and more because they provide constant-time access to the smallest (or biggest) member.

The Graph is a crucial advanced data structure that depicts a group of nodes (vertices) linked by edges. Graphs can be weighted or unweighted, as well as directed or undirected. They are essential for network and connection modelling, having uses in recommendation systems, social networks, and transportation planning. The "networkx" package of Python offers graph manipulation features.

The Trie is a tree-like structure designed specifically for

implementations. Prefix searches are made possible by the “trie's” representation of each character in the string at each level.
When managing a group of disjoint sets and carrying out operations like union and search, the Disjoint-Set (Union-search) data structure is employed. It is necessary for tasks like applying “Kruskal's” approach to find a minimal spanning tree or identifying linked components in a network.

9.2 Stacks
In Python, a stack is a linear data structure that adheres to the Last-In-First-Out (LIFO) tenet. It operates like a set of things piled on top of one another, where the last thing added is the first thing taken away. In many algorithms, stacks are frequently used to control function calls, handle expressions, and keep track of state. Lists' capacity for dynamic scaling makes it simple to create stacks. Items are added to the top of the stack using the "append()" function, and the top item is removed and returned using the "pop()" method.

For Example:
Consider a scenario where you need to evaluate a mathematical expression containing parentheses. You can use a stack to ensure that the parentheses are balanced before proceeding with the evaluation:

```
def is_balanced(expression):
stack = []
for char in expression:
if char == '(':
```

```
stack.append(char)
elif char == ')':
if not stack: # If the stack is empty, there's an imbalance
return False
stack.pop()
return len(stack) == 0 #If the stack is empty, all
parentheses are balanced

expression1 = "((2 + 3) * 5)"
expression2 = "(3 * 5) + 7)"
print(is_balanced(expression1)) # Output: True
print(is_balanced(expression2)) # Output: False
```

The open brackets found in the expression are recorded on a stack by the "is_balanced()" function. When the stack is empty and a closing parenthesis is met, an imbalance has occurred. If the stack is empty after processing the full expression, all brackets are balanced.

9.3 Queues

A queue is a basic data structure that conforms to the First-In-First-Out (FIFO) rule. The first person to join is the first to be served, much like a queue of people waiting. For handling processes that need sequential processing, such as scheduling, work distribution, and breadth-first search methods, queues are frequently utilised.

Lists can be used to build queues, however, the "queue" module, which offers the "Queue" class, is more effective. The "get()" function removes and returns items from the front of the queue, whereas the "put()" method adds things to the rear of the queue.

For Example:

Let's explore an example of using a queue to perform a breadth-first search (BFS) traversal on a graph:

```
from queue import Queue

def bfs(graph, start):
visited = set()
queue = Queue()
queue.put(start)
visited.add(start)

while not queue.empty():
node = queue.get()
print(node, end=' ')

for neighbor in graph[node]:
if neighbor not in visited:
queue.put(neighbor)
visited.add(neighbor)

# Example graph represented as an adjacency list
graph = {
'A': ['B', 'C'],
'B': ['D', 'E'],
'C': ['F'],
'D': [],
'E': ['F'],
'F': []
}

bfs(graph, 'A') # Output: A B C D E F
```

The "bfs()" method performs a breadth-first traversal of the graph using a queue. When the queue is empty, the

procedure repeats itself starting from the designated "start" node, exploring its neighbours and adding them to the queue if they haven't already been visited. This effectively completes a BFS traverse by ensuring that nodes at the same level are visited before going to the next level.

9.4 Linked Lists

A basic data structure in Python for organising and storing a set of elements is a linked list. Linked lists don't need contiguous memory allocation, unlike arrays or lists. Instead, they are made up of nodes, each of which contains a reference (or pointer) to the following node in the sequence as well as the data. This makes it possible to insert and remove items from the list quickly and effectively.

Custom classes can be used to implement linked lists. Each node has a link to the next node and the data itself. Typically, the last node refers to none to denote the conclusion of the list.

For Example: Here's a simple example of a singly linked list,

```
class Node:
def __init__(self, data):
self.data = data
self.next = None

class LinkedList:
def __init__(self):
self.head = None

def append(self, data):
```

```
new_node = Node(data)
if not self.head:
self.head = new_node
else:
current = self.head
while current.next:
current = current.next
current.next = new_node

def display(self):
current = self.head
while current:
print(current.data, end=' -> ')
current = current.next
print("None")

# Creating a linked list and adding elements
llist = LinkedList()
llist.append(1)
llist.append(2)
llist.append(3)

# Displaying the linked list: 1 -> 2 -> 3 -> None
llist.display()
```

The "LinkedList" class has methods for displaying and appending data to the linked list. The various list items are represented by the "Node" class. A linked list is formed when entries are added by adding additional nodes that are connected to one another.

9.5 Trees

In Python and other programming languages, trees are

popular hierarchical data structures used to depict hierarchical connections between components. A tree is made up of nodes, which are joined together by edges. The top node, known as the root, and successive nodes branch out to form subtrees. Each node can have one parent node and one or more child nodes. For many different applications, such as file systems, hierarchical data representation, and search algorithms, trees are essential.
Implementing trees using custom classes is possible. Each node normally contains information and pointers (references) to its sibling nodes. There are several sorts of trees, including balanced trees like AVL trees or Red-Black trees, binary search trees, and balanced trees.
For Example: Here's an example of a binary tree implementation,

```
class TreeNode:
def __init__(self, data):
self.data = data
self.left = None
self.right = None

class BinaryTree:
def __init__(self, root_data):
self.root = TreeNode(root_data)

def insert(self, data):
self._insert_recursive(self.root, data)

def _insert_recursive(self, node, data):
if data < node.data:
if node.left is None:
```

```
node.left = TreeNode(data)
else:
self._insert_recursive(node.left, data)
else:
if node.right is None:
node.right = TreeNode(data)
else:
self._insert_recursive(node.right, data)

def inorder_traversal(self, node):
if node:
self.inorder_traversal(node.left)
print(node.data, end=' ')
self.inorder_traversal(node.right)

# Creating a binary tree and inserting elements
tree = BinaryTree(10)
tree.insert(5)
tree.insert(15)
tree.insert(3)
tree.insert(7)

# Performing an inorder traversal: 3 5 7 10 15
tree.inorder_traversal(tree.root)
```

The "BinaryTree" class contains methods to insert data and perform an in order traversal of the tree. The "TreeNode" class represents the individual nodes of the tree.

9.6 Graphs

Graphs are flexible data structures that can be used to show connections and interactions between different

types of items. Graphs in Python are made up of nodes (vertices) and edges that link adjacent nodes together. A variety of real-world contexts, including social networks, transportation systems, computer networks, and more, are frequently modelled using graphs. They are important in resolving complex connection and route difficulties.

Graphs may be implemented using a number of different methods. One typical method is to employ adjacency lists or dictionaries, where each node serves as a key and the value is a list of nearby nodes (adjacent vertices). Adjacency matrices, which display the graph as a 2D matrix and provide a value to each edge between nodes, are an alternative. For Example:

```
class Graph:
def __init__(self):
self.adjacency_list = {}

def add_vertex(self, vertex):
if vertex not in self.adjacency_list:
self.adjacency_list[vertex] = []

def add_edge(self, vertex1, vertex2):
if vertex1 in self.adjacency_list and vertex2 in
self.adjacency_list:
self.adjacency_list[vertex1].append(vertex2)
self.adjacency_list[vertex2].append(vertex1)

def display(self):
for vertex, neighbors in self.adjacency_list.items():
print(f"{vertex}: {neighbors}")
```

```
# Creating a graph and adding vertices and edges
graph = Graph()
graph.add_vertex("A")
graph.add_vertex("B")
graph.add_vertex("C")
graph.add_edge("A", "B")
graph.add_edge("B", "C")

# Displaying the graph's adjacency list
graph.display()
```

The "Graph" class contains methods to add vertices and edges, as well as to display the adjacency list representation of the graph.

9.7 Hashing and Hash Maps

Hashing and hash maps are fundamental ideas for effective data storage and retrieval. Utilizing a hash function, hashing entails converting data (keys) into fixed-size values (hash codes). Data structures called hash maps, commonly referred to as dictionaries, use hashing to store key-value pairs and provide quick access to items depending on their keys.

Hash maps are implemented using the built-in "dict" class. Hashing is integral to many Python data structures, including sets and dictionaries.

For Example: Here's a simple example of using a hash map to store and retrieve student grades,

```
class StudentGrades:
def __init__(self):
self.grades = {}

def add_grade(self, student, grade):
```

```
self.grades[student] = grade

def get_grade(self, student):
return self.grades.get(student, "Grade not found")

# Creating a student grades hash map
grades_map = StudentGrades()

# Adding grades to the hash map
grades_map.add_grade("Alice", 90)
grades_map.add_grade("Bob", 85)
grades_map.add_grade("Carol", 92)

# Retrieving grades from the hash map
print(grades_map.get_grade("Alice")) # Output: 90
print(grades_map.get_grade("David")) # Output: Grade
not found
```

Student names are used as keys and their associated grades are used as values in the "StudentGrades" class's hash map. The "get_grade()" function fetches grades based on the student's name, whereas the "add_grade()" method adds grades to the hash map.

9.8 Trie

For effective storing and access of a dynamic set of strings or keys, a trie (pronounced "try") is a specialized tree-like data structure. In a trie, each node is a character, and a string is formed by the path from the root to each node. Tries are more advantageous than other data structures for activities like autocomplete, prefix matching, and dictionary implementations because they offer quick lookups and less memory use.

Custom classes can be utilized to implement a trie. Each node is made up of a character, a Boolean indicating whether or not it completes a word and pointers to any child nodes it has.
For Example: Here's an example of a simple trie implementation for storing and searching words,

```
class TrieNode:
def __init__(self):
self.children = {}
self.is_end_of_word = False

class Trie:
def __init__(self):
self.root = TrieNode()

def insert(self, word):
node = self.root
for char in word:
if char not in node.children:
node.children[char] = TrieNode()
node = node.children[char]
node.is_end_of_word = True

def search(self, word):
node = self.root
for char in word:
if char not in node.children:
return False
node = node.children[char]
return node.is_end_of_word

# Creating a trie and inserting words
```

```
trie = Trie()
trie.insert("apple")
trie.insert("app")
trie.insert("banana")

# Searching for words in the trie
print(trie.search("apple")) # Output: True
print(trie.search("app")) # Output: True
print(trie.search("banana")) # Output: True
print(trie.search("orange")) # Output: False
```

The "Trie" class contains methods to insert words and search for them in the trie. The "TrieNode" class represents individual nodes in the trie.

9.9 Disjoint-Set (Union-Find)

A disjoint set, commonly referred to as a union-find structure, is a type of data structure used to efficiently carry out operations like a union (combining sets) and find (identifying the set to which an element belongs). It is frequently employed to address connectivity-related issues, such as identifying linked elements in a network or applying "Kruskal's algorithm" to create a minimal spanning tree.

There are several ways to implement a Disjoint Set. The use of a list or an array, where each element represents a set, and the values indicate the element's parent (or a representative element), is a typical strategy. In the union process, the parent of the representation from one set is updated to point to the representative from the other set.

For Example: Here's an example of a simple Disjoint Set implementation using lists,

```
class DisjointSet:
def __init__(self, size):
self.parent = [i for i in range(size)]

def find(self, element):
if self.parent[element] == element:
return element
self.parent[element] = self.find(self.parent[element])
return self.parent[element]

def union(self, set1, set2):
parent_set1 = self.find(set1)
parent_set2 = self.find(set2)
if parent_set1 != parent_set2:
self.parent[parent_set1] = parent_set2

# Creating a Disjoint Set and performing union
operations
ds = DisjointSet(5)
ds.union(0, 1)
ds.union(2, 3)
ds.union(3, 4)

# Checking if elements are in the same set
print(ds.find(0) == ds.find(1)) # Output: True
print(ds.find(2) == ds.find(4)) # Output: False
```

The "DisjointSet" class contains methods for finding the representative of a set and performing union operations. The "parent" list represents the disjoint sets.

Database

10.1 Introduction

A structured collection of data that is organized and kept in a systematic way, allowing for effective data retrieval, modification, and management, is referred to as a database in Python. A wide range of applications, from straightforward data storage to intricate data processing and reporting, depend on databases. Python has a number of frameworks and packages that make it easier to create, maintain, and interface with databases, enabling developers to work with data in a fluid manner. For processing massive amounts of data in a systematic manner, databases are essential. They offer a mechanism to manage data integrity and consistency when saving, accessing, updating, and deleting information. Databases are frequently used in Python for a variety of tasks, including online applications, desktop programs, scholarly research, and data-driven decision-making.

SQLite is one of Python's most popular database management systems (DBMS). The relational database engine SQLite is small, serverless, and self-contained. Since it is already included in the Python standard library, using it is simple and doesn't call for any further installs. SQLite databases are appropriate for small to medium-sized applications since they are stored as files on the local file system.

To work with "SQLite" databases in Python, you need to import the sqlite3 module, which provides functions and classes for interacting with SQLite databases.

applications, SQLite is a self-contained, serverless, open-source, and lightweight relational database management system (RDBMS). Because SQLite doesn't require a separate server process or configuration, unlike conventional database systems, it is simple to install into a variety of platforms. It streamlines deployment and management by operating directly on a single database file that is kept on the local filesystem.

Data integrity and dependability are ensured by SQLite's support for standard SQL syntax and ACID (Atomicity, Consistency, Isolation, Durability) compliance. Due to its compact footprint and low resource needs, it is especially well suited for small to medium-sized projects, embedded systems, mobile apps, and desktop software.

SQL commands are used by developers to communicate with SQLite using libraries or APIs offered by different programming languages, including Python. The creation and maintenance of tables, indexing for quick data retrieval, support for intricate queries, and transaction management are some of its important features. SQLite enables concurrent access by several users and can manage heavy workloads whilst being lightweight.

This RDBMS is visible in situations where efficiency, quickness, and little overhead are important. The simplicity, portability, and convenience of the use of SQLite make it a great option for projects requiring a self-contained, file-based database solution, even while it might not be appropriate for large-scale applications demanding numerous concurrent writes or high throughput.

10.3 Database Connection with SQLite

In Python, the term "database connectivity" refers to the power of a Python program to connect to a relational database management system (RDBMS), such as MySQL, PostgreSQL, SQLite, or Oracle. Python programs can interact with databases through this connection, carrying out data collection, manipulation, and storage tasks as well as facilitating the seamless integration of application logic and data storage.

Python programmers frequently utilize packages like sqlite3, mysql-connector, psycopg2, and pyodbc to accomplish database connectivity. These libraries include classes and methods for developing, managing, and running SQL queries against the database.

Here's a simple explanation of how to establish database connectivity in Python using the sqlite3 library, which is included in Python's standard library:

i. Import Required Modules: - Begin by importing the "sqlite3" library, which provides functions for SQLite database connectivity.

```
import sqlite3
```

ii. Establish a Connection: - To connect to the SQLite database, use the "connect()" method. The database will be created if it doesn't already exist.

```
conn = sqlite3.connect('my_database.db')
```

iii. Create a Cursor: - A cursor is a control structure that enables interaction with the database. It allows you to execute SQL queries and fetch results.

```
cursor = conn.cursor()
```

iv. Execute SQL queries: - You can execute SQL queries using the cursor's "execute()" method. Here's an example of creating a table and inserting data into it.

```
cursor.execute('''CREATE TABLE IF NOT EXISTS users (
id INTEGER PRIMARY KEY,
username TEXT,
email TEXT)''')

cursor.execute("INSERT INTO users (username, email)
VALUES (?, ?)", ('john_doe', 'john@example.com'))
```

v. Commit Changes and Close Connection: - After executing queries, commit the changes to the database and close the connection.

```
conn.commit()
conn.close()
```

A new user record is placed into the 'users' table when a connection to a SQLite database has been made and built. For other databases, using their respective libraries, the same procedures can be used.

Database Connection Example

Example: Let's create a simple example of a Python script that connects to a SQLite database, creates a table, inserts data, and retrieves it:

```
import sqlite3

# Establish connection
connection = sqlite3.connect("example.db")
cursor = connection.cursor()

# Create table
```

```
create_table_query = "CREATE TABLE IF NOT EXISTS
books (id INTEGER PRIMARY KEY, title TEXT, author
TEXT)"
cursor.execute(create_table_query)
connection.commit()

# Insert data
insert_query = "INSERT INTO books (title, author)
VALUES (?, ?)"
data = [("The Great Gatsby", "F. Scott Fitzgerald"), ("To
Kill a Mockingbird", "Harper Lee")]
cursor.executemany(insert_query, data)
connection.commit()

# Retrieve and print data
select_query = "SELECT * FROM books"
cursor.execute(select_query)
books = cursor.fetchall()
for book in books:
print(book)

# Close connection
cursor.close()
connection.close()
```

We import the “sqlite3” library, create a connection to an SQLite database named "example.db," create a "books" table, insert two books, retrieve and print the data, and finally close the connection.

10.4 Performing Database Operations

Utilizing specialized libraries and methods to connect with databases, database operations in Python enable

data storage, retrieval, change, and analysis within your applications. Python provides a number of capabilities for managing databases, including the ability to connect to databases, run queries, manipulate data, and guarantees data integrity. Let's examine these features using an example to show how Python handles database operations.
Example: Let's use the straightforward example of writing a Python program to handle the book records of a library using SQLite, a compact and embedded relational database.

Step – 1 Importing Required Libraries
Start by importing the necessary libraries. In this case, we'll use the built-in sqlite3 module to work with SQLite.

```
import sqlite3
```

Step – 2 Connecting to Database
Establish a connection to the SQLite database. If the database doesn't exist, SQLite will create it.

```
connection = sqlite3.connect("library.db")
cursor = connection.cursor()
```

Step – 3 Creating a Table
Define the structure of the table to store book information. We'll create a table named "books" with columns for book ID, title, author, and year of publication.

```
create_table_query = """
CREATE TABLE IF NOT EXISTS books (
id INTEGER PRIMARY KEY,
title TEXT,
```

```
author TEXT,
year INTEGER
)
"""
cursor.execute(create_table_query)
connection.commit()
```

Step – 4 Inserting Data

Insert book records into the "books" table using the "executemany()" method.

```
insert_query = "INSERT INTO books (title, author, year)
VALUES (?, ?, ?)"
books_data = [
("Python – Three Levels", "Pawanpreet Singh", 2023),
("JavaScript", "Pawanpreet Singh", 2023)
]
cursor.executemany(insert_query, books_data)
connection.commit()
```

Step – 5 Querying the Database

Retrieve and display the list of books using a "SELECT" query.

```
select_query = "SELECT * FROM books"
cursor.execute(select_query)
books = cursor.fetchall()
for book in books:
print(book)
```

Step – 6 Updating Data

Perform an update operation to change the author of a book.

```
update_query = "UPDATE books SET author = ? WHERE
```

title = ?"
new_author = "Python – Backend Developing (Edited)"
book_title = "Pawanpreet Singh"
cursor.execute(update_query, (new_author, book_title))
connection.commit()

Step – 7 Closing the Connection
Finally, close the cursor and the database connection to release resources.
cursor.close()
connection.close()

10.5 Database Security
In order to safeguard sensitive data contained in databases from unauthorized access, manipulation, and malicious actions, database security in Python is an important part of application development. Strong database security procedures include a variety of safeguards for the confidentiality and integrity of data, including authentication, authorization, encryption, and input validation. Let's examine these ideas in more detail and give an example of how to improve database security in a Python program.
For Example:
Let's have a look at a Python program that controls user profiles kept in a MySQL database. To improve database security, we'll concentrate on adding authentication, authorization, and input validation.

Step – 1 Authentication and Authorization
import mysql.connector

```
# Establish database connection
connection = mysql.connector.connect(
host="localhost",
user="app_user",
password="strong_password",
database="user_profiles"
)
cursor = connection.cursor()


# Authenticate user
def authenticate(username, password):
query = "SELECT id FROM users WHERE username = %s
AND password = %s"
cursor.execute(query, (username, password))
user_id = cursor.fetchone()
return user_id

# Authorize user based on role
def authorize(user_id, required_role):
query = "SELECT role FROM users WHERE id = %s"
cursor.execute(query, (user_id,))
user_role = cursor.fetchone()
if user_role and user_role[0] == required_role:
return True
return False

Step -2 Input Validations and Sanitization
def get_user_profile(user_id):
query = "SELECT * FROM profiles WHERE user_id = %s"
cursor.execute(query, (user_id,))
user_profile = cursor.fetchone()
```

return user_profile

def create_user_profile(user_id, profile_data):
query = "INSERT INTO profiles (user_id, bio) VALUES (%s, %s)"
cursor.execute(query, (user_id, profile_data))
connection.commit()

Using the right information, we create a secure connection to the MySQL database. The "authenticate" function verifies user credentials and, if successful, returns the user's ID. The "authorize" function determines if a user is authorized to do a certain activity. The “get_user_profile” and “create_user_profile” methods employ parameterized queries to perform input validation in order to avoid SQL injection.

Web Development with Python

11.1 Introduction

Python is a programming language that may be used to generate dynamic, interactive web pages and online applications. Python is a great choice for creating multiple components of a web project, from the server-side logic to the user interface, due to its adaptability, simplicity of use, and multiple libraries. Python's use in web development has motivated the creation of strong frameworks and tools that speed up the creation process.

The frontend and the backend are the two fundamental parts of web development. Everything that users directly interact with, such as the layout, design, and user interface components, is considered to be part of the frontend. The web application's backend, on the other hand, controls the server-side logic, database interactions, and business logic.

A variety of frameworks and tools are available in Python for front-end and back-end development. Frameworks like Flask, Django, and FastAPI can be utilized for front-end development. These frameworks include URL routing, template engines, and other tools that make it easier to create user-friendly interfaces. By combining HTML templates with Python code, developers may construct dynamic websites that provide content specific to user demands.

Python's capabilities in the backend are highlighted by frameworks like Django and Flask. Django, a powerful and feature-rich framework, makes it easier to create

eliminating the requirement for unstructured SQL queries. While keeping simplicity, Flask is a lightweight framework that offers greater flexibility, allowing developers to select the components they want.
The capabilities of Python for web development go more than just building websites. Python's easy communication with several databases, including SQLite, PostgreSQL, and MySQL, is important for web applications since they frequently need to store and access data. Python libraries can be used to handle these databases, offering effective data management for online applications.
The wide ecosystem of third-party packages available for Python is required for web development. Data manipulation, analysis, and visualization are made possible by libraries like NumPy, Pandas, and Matplotlib, enabling the introduction of data-driven features into web applications. Additionally, Selenium-like tools enable automated testing, confirming the reliability and error-free operation of online applications.
Using external services and APIs (Application Programming Interfaces) is another aspect of web development. The Requests package in Python makes it simple to send and receive HTTP requests for data from and to external sources. Developers may include third-party services in their web applications with this functionality, including payment gateways, social networking platforms, and data sources.

11.2 Popular Frameworks in Python

Python includes a number of well-known web frameworks that improve and speed up the web development process. These frameworks offer pre-made elements,

tools, and standards that let programmers concentrate on creating functionality rather than taking care of small details. The most well-known Python web frameworks are listed below:

i. Django

A high-level, full-stack framework called Django is known for its "batteries-included" philosophy. It offers everything required to create complicated online applications, including the ability to handle routing, authentication, database management, and user session management. Django emphasizes the DRY (Don't Repeat Yourself) architectural pattern and the Model-View-Controller (MVC) architectural pattern.

Imagine if we developed a blogging platform. You may define models for blog articles, users, and comments using Django. To manage posts and users and to develop views and templates to show the blog's content, you would utilize Django's built-in admin interface. Data from blogs could be easily stored and retrieved thanks to Django's ORM, which would manage database interactions.

ii. Flask

With a few limitations, Flask is a compact and adaptable micro-framework that offers the tools needed to create online applications. For smaller projects or when developers want greater control over the components they utilize, it is a fantastic option. Flask is sometimes referred to as a "micro" framework since it has a limited set of pre-built features and lets developers pick and select any additional features they require.

Consider making a straightforward to-do list application. You would define routes in Flask to deal with adding, updating, and removing tasks. To present the jobs on the frontend, you would create templates. Due to Flask's simplicity, you can concentrate on the app's essential features without being distracted by extraneous overhead.

11.3 Setting Up a Web Application

Installing, building, and launching your application are just a few of the tasks involved in setting up a web application in Python using the Django framework. Let's go over the procedure while constructing a straightforward "Task List" web application as an example.

Step – 1 Install Django

Make sure Python is set up on the system before you begin. Next, run the following command to install Django:
pip install Django

Step – 2 Create a New Django Project

Run the following command in your terminal after navigating to the directory where you wish to create your project:
django-admin startproject tasklist
This creates a new directory named "tasklist" containing the initial project structure.

Step – 3 Create a Django App

Make a new Django app by navigating to the "tasklist" directory. A Django project's app is one of its parts that

are modular.
cd tasklist
python manage.py startapp tasks

Step – 4 Define Models
Open the "models.py" file in the "tasks" app directory to specify the models for your application. In our example, we'll make a straightforward "Task" model:

```
from django.db import models

class Task(models.Model):
title = models.CharField(max_length=200)
completed = models.BooleanField(default=False)

def __str__(self):
return self.title
```

Step – 5 Creating and Apply Migration
Django manages your database structure through migrations. To generate and implement migrations, execute the following commands:

```
python manage.py makemigrations
python manage.py migrate
```

Step – 6 Create Views and Templates
Create views in the "tasks" app's "views.py" file. Views process requests and produce results. To render HTML material, create templates. We'll develop a simple task list view as an example:

```
from django.shortcuts import render
from .models import Task
```

```
def task_list(request):
tasks = Task.objects.all()
return render(request, 'tasks/task_list.html', {'tasks':
tasks})
```

Step – 7 Create Templates

Make a "templates" directory in the "tasks" app directory.

Create the "task_list.html" file in the "templates" directory:

```
<!DOCTYPE html>
<html>
<head>
<title>Task List</title>
</head>
<body>
<h1>Task List</h1>
<ul>
{% for task in tasks %}
<li>{{ task.title }} - {% if task.completed %}Completed{%
else %}Pending{% endif %}</li>
{% endfor %}
</ul>
</body>
</html>
```

Step – 8 Configure URLs

Define URL patterns in the "urls.py" file of the "tasks" app.

```
from django.urls import path
from . import views

urlpatterns = [
```

```
path('', views.task_list, name='task_list'),
]
```

Step – 9 Include App URLs in Project URLs
In the "urls.py" file of the project "tasklist" directory, include the app's URL.

```
from django.contrib import admin
from django.urls import path, include

urlpatterns = [
path('admin/', admin.site.urls),
path('', include('tasks.urls')),
]
```

Step – 10 Run the Development Server
Finally, run the development server to see your web application in action.

```
python manage.py runserver
```

Open your web browser and navigate to "http://127.0.0.1:8000/" to view your Task List web application.

11.4 Handling HTTP Requests
Data is sent and received through the Hypertext Transfer Protocol (HTTP), the basic internet protocol while handling HTTP requests in Python. To make this procedure easier, Python has libraries like "http.client", "urllib", and third-party libraries like "requests".

"Sending Request and Receiving Responses"
Sending Request: - You can generate and send HTTP requests to web servers using Python. This procedure is

made easier by the "requests" package, which offers a high-level API for sending HTTP requests. Your requests can include data, headers, parameters, and an HTTP method (GET, POST, PUT, or DELETE).
Receiving Responses: - The web server replies to a request with an HTTP response. Status codes, headers, and response content can all be handled and processed by Python. You can use the answer to extract and modify data for later processing.
Example: Fetching Data from Rest API

```
import requests

def fetch_data_from_api():
url = "https://jsonplaceholder.typicode.com/posts/1"

try:
response = requests.get(url)
response.raise_for_status() # Raise an exception for HTTP errors
data = response.json() # Parse JSON response

title = data['title']
body = data['body']

print(f"Title: {title}")
print(f"Body: {body}")

except requests.exceptions.RequestException as e:
print(f"Error: {e}")

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

The GET request is sent to a 'JSONPlaceholder' API endpoint using the "requests" library. The reply is obtained and converted into JSON. The post's title and body are printed after being derived from the JSON data. Any HTTP problems (like 404 Not Found) will raise an exception for correct error handling thanks to the "raise_for_status()" method.

11.5 APIs and Web Services

In order to improve communication and data exchange across various software applications and systems, web services and APIs are needed. They provide programmers access to particular features or information offered by outside platforms, services, or databases. Powerful frameworks and tools for communicating with APIs and using web services are available in Python.

APIs: - A collection of rules and processes known as an API describes how various software components may interact and communicate with one another. APIs hide the complexity at the core and expose particular functionality and data. They act as middlemen, allowing programmers to create apps that make use of the features of other services without having to learn how they operate from the inside.

Web Services: - An API called a web service operates particularly via the internet utilizing established protocols like HTTP. Applications can use them to request for and communicate data in a structured format, such JSON or XML. Web services provide remote access to server-hosted functionality or data, enabling the development of distributed and networked applications.

Example: Fetching Weather data from a Web Services

```
import requests

def get_weather_data(city_name):
api_key = "YOUR_API_KEY"
base_url =
f"http://api.openweathermap.org/data/2.5/weather?q=
{city_name}&appid={api_key}"

try:
response = requests.get(base_url)
response.raise_for_status()
weather_data = response.json()

temperature = weather_data["main"]["temp"]
weather_description = weather_data["weather"][0]
["description"]

print(f"Temperature: {temperature} K")
print(f"Weather: {weather_description}")

except requests.exceptions.RequestException as e:
print(f"Error: {e}")

if __name__ == "__main__":
city = "India"
get_weather_data(city)
```

We submit an HTTP GET request to the "OpenWeatherMap" API using the "requests" package, supplying the city name and an API key as inputs. The reply is obtained and converted into JSON. From the JSON data, we extract the temperature and a

description of the weather before printing the findings.

Machine Learning and Data Science

12.1 Introduction

With the help of Python, the areas of machine learning and data science are able to derive useful predictions and conclusions from data. Python, a flexible and popular programming language, acts as the foundation for these fields by providing a robust ecosystem of libraries and tools that allow practitioners to effectively manage, analyse, and model data.

Machine Learning

A branch of artificial intelligence known as machine learning (ML) deals with creating models and algorithms that enable computers to learn from data and make predictions or judgments without being explicitly programmed. Python's simplicity, readability, and rich library support make it an important instrument in machine learning.

In the field of machine learning, Python libraries like TensorFlow, Scikit-Learn, and Keras offer a strong toolbox for developing, testing , and comparing various kinds of machine-learning models. For example, the user-friendly interface provided by Scikit-Learn makes it easy to do tasks like classification, regression, clustering, and dimensionality reduction. TensorFlow and Keras, on the other hand, are geared towards deep learning, a branch of machine learning that uses neural networks for tasks like audio and picture recognition, natural language processing, and more.

fall under the umbrella of data science and are used to provide valuable insights and inform decisions. Data scientists are given the tools they need to efficiently edit and visualize data thanks to Python's comprehensive libraries, including Pandas, Matplotlib, and Seaborn.

12.2 Data Collection and Preparation

The collecting, organization, and refinement of raw data into a useful and organized format for analysis and modeling are important steps in data science and machine learning workflows. These essential tasks are made easier by a range of libraries and Python-specific approaches.

Data collection involves collecting data from a variety of sources, including files, websites, databases, and APIs. Web scraping, or the automated extraction of data from webpages, is made possible by Python's libraries, such as "requests," "BeautifulSoup," and "Selenium." Using libraries like "requests" or specialized packages made for certain APIs, like "tweepy" for Twitter data, one may access APIs. Libraries like "SQLAlchemy" improve database interfaces by enabling queries and data retrieval. Additionally, the "open()" function of the Python programming language makes it simple to read data from local files, and "pandas" offers a variety of data structures for storing and manipulating information.

To assure the quality and fit of the obtained data for analysis, data preparation includes preprocessing and cleaning. With tools for data translation and manipulation, libraries like "pandas" play an important part. Functions like "dropna()" or "fillna()" are used to manage missing data, while "duplicated()" and

"drop_duplicates()" are used to find and get rid of duplicates. Utilizing techniques like string manipulation or regular expressions, inconsistent data may be standardized.

12.3 Data Analysis and Exploration

The methodical investigation, visualization, and interpretation of data using Python constitute data analysis and exploration, two essential parts of the data science workflow. By exposing hidden patterns, connections, and insights, these procedures promote creativity and well-informed decision-making.

Multiple components and tools are available in Python that make it easy to explore and analyze data. The cornerstone for manipulating and analyzing data is the fundamental library, pandas. It provides DataFrames and Series data structures, allowing users to import, process, and filter data effectively. Particularly DataFrames enable tabular data format, similar to spreadsheets, which makes managing and analyzing datasets simple.

A important first stage in the data analysis process is exploratory data analysis (EDA). The ability to depict data distributions, relationships, and anomalies is made possible by Python's "matplotlib" and "seaborn" modules. Heatmaps and pair plots show relationships between variables, while histograms, scatter plots, and box plots assist grasp the properties of the data. These visualizations offer perceptions into the organization of the data as well as possible research topics.

12.4 Data Visualization

The technique of displaying complicated data in

graphical or visual representations to derive insightful conclusions is known as data visualization in Python. The variety of modules available in Python makes it possible to build meaningful visualizations that effectively communicate trends, patterns, and connections in data. Libraries like "matplotlib" offer flexible charting features, enabling the generation of a variety of plots including line charts, scatter plots, and histograms. "matplotlib" is built upon by Seaborn, which provides higher-level functions for attractive visualizations. Interactive web-based visualizations made possible by "Plotly" increase user engagement. "geopandas" and "Folium" are useful for efficiently visualizing geographic data. Additionally, the Jupyter Notebooks and Python integration makes it possible for inline visualizations, which makes it simple to integrate charts with explanatory text.

12.5 NLP (Natural Language Processing)

Python's Natural Language Processing (NLP) uses algorithms to interact with and study human language. Python's extensive library ecosystem enables NLP operations and the extraction of knowledge from text input. Tools for tokenization, part-of-speech tagging, and sentiment analysis are provided by "NLTK" and "spaCy." These packages simplify text preparation, stemming, and lemmatization. While "TextBlob" makes sentiment analysis and language translation easier, "Gensim" makes topic modelling and document similarity analysis more convenient. Modern pre-trained models for activities like text production and language recognition are offered by Transformers and Hugging Face. Text categorization and clustering are made possible by Python's interaction with

machine learning packages like "scikit-learn". Python is a key component of contemporary language-driven technology and is used to build efficient NLP-driven applications, such as chatbots and sentiment analysis engines.

Concurrency and Multi-Threading

13.1 Introduction

Modern programming requires an understanding of concurrency and multithreading, two ideas that help programmers create effective, responsive systems that can carry out several activities at once. Python offers implementation methods for both notions, which each take a different approach to the problem of managing numerous jobs concurrently.

Concurrency

Concurrency is the capacity of a program to manage several activities or processes at once, enabling them to advance separately. It doesn't always indicate real parallel execution because the tasks might be carried out concurrently via sharing resources and context switching. Applications having user interfaces that react to user inputs while carrying out background processes, such as web servers managing many client requests, require concurrency in order to manage several activities at once.

Multithreading

Previously developed, multithreading uses threads to accomplish concurrency. Because threads are lightweight and may transition between jobs fast, they are appropriate for I/O-bound activities (where the thread often waits for external resources like file I/O or network operations). Developers can easily build and

restriction.
The term "parallelism" describes the actual simultaneous execution of activities using several processing cores to carry out calculations. While Python's GIL restricts parallelism within a single process, employing many processes can lead to genuine parallelism. The "multiprocessing" module offers a user interface for setting up and controlling many processes, each with a dedicated Python interpreter instance and memory space. By allowing each process to execute independently on a different core, gets around the GIL restriction and enables true parallel execution. For CPU-bound activities that can be broken up into smaller pieces and carried out in parallel, parallelism is acceptable.

13.2 Thread and Process

Programs may do several tasks at once because of concurrency and parallelism techniques like threads and processes. Although they both aim to increase efficiency, their implementations and use cases are different.

Thread

A process's lightweight execution unit is called a thread. Although communication and data exchange are facilitated by the fact that threads share the same memory space, the Global Interpreter Lock (GIL) prevents the true parallel execution of threads inside a single process. For I/O-bound operations, when waiting for outside resources (such as file I/O or network requests) dominates the execution time, threads are a good fit. A web server managing several client connections

simultaneously, each connection maintained by a different thread, is an example.

```
import threading

def print_numbers():
for i in range(1, 6):
print(f"Thread 1: {i}")

def print_letters():
for letter in 'abcde':
print(f"Thread 2: {letter}")

thread1 = threading.Thread(target=print_numbers)
thread2 = threading.Thread(target=print_letters)

thread1.start()
thread2.start()

thread1.join()
thread2.join()

print("Threads have finished")
```

Process

On the other hand, a process is a standalone instance of the software with its own memory and Python interpreter. By using many CPU cores, processes may achieve true parallelism since they are not impacted by the GIL. Processes are therefore appropriate for CPU-bound jobs that need large amounts of processing. A data processing program that breaks a sizable dataset into pieces processes each in parallel in a different

process, and then merges the outcomes is an example.

```
import multiprocessing

def process_data(data_chunk):
result = [x * 2 for x in data_chunk]
return result

if __name__ == '__main__':
data = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
num_processes = 4

pool = multiprocessing.Pool(processes=num_processes)
chunk_size = len(data) // num_processes
chunks = [data[i:i + chunk_size] for i in range(0,
len(data), chunk_size)]

results = pool.map(process_data, chunks)
pool.close()
pool.join()

final_result = [item for sublist in results for item in
sublist]
print(final_result)
```

13.3 GIL (Global Interpreter Lock)

A key feature of the Python programming language that significantly affects its concurrency paradigm is the Global Interpreter Lock (GIL). The Python interpreter uses the GIL, a mutex or lock, to make sure that only one thread is executing Python bytecode at once within a single process. As a result, only one thread may execute Python code in a multi-threaded Python program while

the others are essentially put on hold. This restriction hinders the real parallel execution of threads within a single process, which might have an impact on performance, especially in jobs that are CPU-bound. Past is the main justification for the GIL's existence. Python was initially intended to be a single-threaded language, and the GIL was added to make memory management easier and prevent the complicated problems that may occur when several threads access and alter the same object at the same time. Even though the GIL makes memory management simpler, it limits the potential performance benefits of multi-threading, especially on multi-core CPUs.
The GIL has a greater impact on CPU-bound processes since the Python interpreter does a large portion of the computational effort in these jobs. On the other hand, I/O-bound activities might profit from multi-threading since threads frequently spend time waiting for outside resources, during which other threads can work. Python threads are so ideally suited for situations like controlling several client connections in a web server.
For example: Here's a simplified example illustrating the effect of the GIL:

```
import threading

counter = 0

def increment():
global counter
for _ in range(1000000):
counter += 1
```

```
# Create two threads that increment the counter
thread1 = threading.Thread(target=increment)
thread2 = threading.Thread(target=increment)

thread1.start()
thread2.start()

thread1.join()
thread2.join()

print("Counter:", counter)
```

Given that two threads add "1000000" to the counter each, you could anticipate that the ultimate value of the "counter" will be "2000000". However, because of race situations and the interleaved execution of threads, the GIL may cause the final number to be less than "2000000". This demonstrates how Python's multi-threading is limited for jobs that are CPU-bound.

13.4 Concurrent Future Model

Python's “concurrent.futures” module offers a high-level interface for executing asynchronous functions concurrently and in parallel. It makes it simpler to build concurrent code without having to explicitly deal with lower-level threading or multiprocessing concerns by abstracting the underlying complexity of maintaining threads or processes. This module is very helpful for activities that may be performed in parallel, such as performing asynchronous I/O operations or running several separate calculations.

The "ThreadPoolExecutor" and "ProcessPoolExecutor" classes are the two primary ones in the

"concurrent.futures" package. These classes include functions like "map()" and "submit()" that let you submit tasks for concurrent execution and obtain the results as soon as they are complete.
For Example: Here's an example demonstrating the use of “ThreadPoolExecutor” to parallelize the execution of a function,

```
import concurrent.futures

def process_data(data_chunk):
return [x * 2 for x in data_chunk]

if __name__ == '__main__':
data = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
num_threads = 4

with
concurrent.futures.ThreadPoolExecutor(max_workers=num
as executor:
chunk_size = len(data) // num_threads
chunks = [data[i:i + chunk_size] for i in range(0,
len(data), chunk_size)]

results = executor.map(process_data, chunks)

final_result = [item for sublist in results for item in
sublist]
print(final_result)
```

The "ThreadPoolExecutor" is used to handle data chunks in a multi-threaded manner. Each element in a data chunk is doubled by the "process_data()" method. The executor's "map()" method is used to simultaneously

apply the "process_data()" function to each chunk. The results are gathered and delivered automatically in the order of the input chunks.

13.5 Asynico

With the help of the potent Python package Asyncio, developers may create concurrent and asynchronous code in a way that is more organised and legible. It works particularly well for network- and I/O-dependent applications where processes must wait a long time for resources like network requests or file I/O operations. Because Asyncio allows for non-blocking execution, one thread may perform several tasks at once without needing to wait for one to finish before going on to the next.

Coroutines, particular kinds of functions that may be interrupted and restarted, are the basic idea behind asyncio. The async keyword is used to build coroutines, which make working with asynchronous processes more natural by enabling developers to write asynchronous code that resembles synchronous code.

For Example:

```
import asyncio

async def greet(name):
print(f"Hello, {name}!")
await asyncio.sleep(1) # Simulate an I/O-bound operation
print(f"Goodbye, {name}!")

async def main():
await asyncio.gather(
greet("Alice"),
```

```
greet("Bob"),
greet("Charlie")
)

asyncio.run(main())
```

The "greet()" coroutine is designed to first print a greeting, then use await "asyncio.sleep(1)" to simulate an I/O-bound activity, and finally display a farewell message. Asyncio.gather() is used by the "main()" coroutine to run numerous coroutines simultaneously. The program publishes hellos and good-byes from many names simultaneously when "asyncio.run(main())" is called, instead of waiting for each I/O-bound action to finish.

API Integration

14.1 Introduction

One of the most important concepts in the current digital world is API integration, which makes it possible for various software programs, platforms, and systems to connect and communicate with one another in an effortless manner. An API, short for "Application Programming Interface," is a collection of established guidelines, procedures, and resources that allow various software components to communicate and exchange data successfully. The functionality, effectiveness, and user experience of software programs across several fields, from e-commerce and banking to healthcare and social networking, are all significantly improved by this integration.

Basically, API integration serves as a link between various software components, allowing them to communicate and exchange data without requiring costly manual intervention. Similar to a language interpreter, APIs facilitate communication between software systems that may otherwise find it difficult to do so because of differences in programming languages, data formats, or operating processes.

14.2 RESTFul APIs

The architectural design approach known as Representational State Transfer (REST) is used to create networked systems, especially web services, in order to promote effective communication between various

them accessible, scalable, and simple to integrate. The flexible programming language Python offers tools and frameworks that make it easier to build and use RESTful APIs.

RESTful APIs in Python follow mainly four principles,
i. Resource Oriented: - Unique URLs serve as identifiers for resources like data items and services. Each URL stands for a particular resource, which may be accessed by utilizing GET, POST, PUT, and DELETE, among other common HTTP methods.
ii. Stateless: - Information about client sessions is not kept on the server. Every request made by the customer comes with all the information required to process it independently.
iii. Client-Sever Separation: - Because the client and server are separate objects, they may develop separately. This division makes maintenance easier and increases scalability.
iv. Cacheable: - Caching server responses can improve performance for frequently used resources.

For Example: Here's a simple example of creating and using a RESTful API in Python using the Flask framework,

```
from flask import Flask, request, jsonify

app = Flask(__name__)

# Sample data
tasks = [
{'id': 1, 'title': 'Buy groceries', 'done': False},
{'id': 2, 'title': 'Clean the house', 'done': True}
```

```
]

# Define a route to get all tasks
@app.route('/tasks', methods=['GET'])
def get_tasks():
return jsonify({'tasks': tasks})

# Define a route to get a specific task
@app.route('/tasks/<int:task_id>', methods=['GET'])
def get_task(task_id):
task = next((task for task in tasks if task['id'] == task_id),
None)
if task is None:
return jsonify({'error': 'Task not found'}), 404
return jsonify({'task': task})

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

A straightforward RESTful API is created using the Flask framework. The definition of two routes includes getting all tasks and getting a single task by ID. The information about the tasks is returned by the API when it is invoked and accessible using a web browser or an HTTP client.

14.3 Making API Requests

Python's API requests include the transfer of data to and from other services, such web APIs, in order to get or modify data. A popular utility that makes it easier to send HTTP requests and handle results is the "requests" package, which enables frictionless communication with many APIs.

For Example: Here is the step for making API requests

Step – 1 Installation

Before making API requests, ensure you have the "requests" library installed. You can install it using pip.

```
pip install requests
```

Step – 2 Making GET Requests

The HTTP GET method is often used to get data from a web API. Here is an example of how to send a GET call to the "JSONPlaceholder" API, which offers test data in place of real data.

```
import requests

response =
requests.get('https://jsonplaceholder.typicode.com/posts/1
if response.status_code == 200:
data = response.json()
print(data)
else:
print('Request failed with status code:',
response.status_code)
```

Using the "requests.get()" method, you may retrieve information from the given URL. The "json()" function is then used to retrieve the JSON data from the response.

Step – 3 Making POST Requests

The HTTP POST method is used to provide data to an API, such as when establishing new resources. Here is an example of how to use the "JSONPlaceholder" API to create a new post.

```
import requests
```

```
new_post = {'title': 'New Post', 'body': 'This is a new post
content.', 'userId': 1}
response =
requests.post('https://jsonplaceholder.typicode.com/posts'
json=new_post)
if response.status_code == 201:
created_post = response.json()
print('New post created with ID:', created_post['id'])
else:
print('Request failed with status code:',
response.status_code)
```

14.4 Handling Responses

To extract the appropriate data, find mistakes, and provide that users are happy after executing API queries, it is essential to handle and analyse the answers with care. Python offers methods and tools for intelligently analysing, modifying, and reacting to API results.

For Example:

i. Parsing JSON Responses: - JSON is a compact and understandable data transfer format that is typically returned by modern APIs. Python's "json" package makes parsing JSON answers simple. Example of parsing JSON response using the requests library,

```
import requests

response = requests.get('https://api.example.com/data')
if response.status_code == 200:
data = response.json()
print(data)
```

```
else:
print('Request failed with status code:',
response.status_code)
```

ii. Error Handling: - Different status codes can be returned by APIs to denote success or failure. Error handling correctly ensures that unexpected outcomes or problems are handled graciously. Example for error handling,

```
import requests

response = requests.get('https://api.example.com/data')
if response.status_code == 200:
data = response.json()
# Process data
else:
print('Request failed with status code:',
response.status_code)
if response.status_code == 404:
print('Resource not found.')
elif response.status_code == 500:
print('Internal server error.')
# Handle other cases
```

iii. Pagination: - To reduce the number of results sent for each request, many APIs paginate their returns. To retrieve all the data, you might have to submit many queries. Example for handling pagination response,

```
import requests

page = 1
while True:
```

```
response = requests.get(f'https://api.example.com/data?
page={page}')
if response.status_code == 200:
data = response.json()
# Process data
page += 1
else:
print('Request failed with status code:',
response.status_code)
break
```

iv. Response Metadata: - APIs frequently provide extra metadata in their answers, such as pagination information or headers providing rate limits. These specifics are accessible via the "response.headers" element. Example for response metadata,

```
import requests

response = requests.get('https://api.example.com/data')
if response.status_code == 200:
data = response.json()
rate_limit = response.headers.get('X-RateLimit-Limit')
print(f'Rate limit: {rate_limit}')
else:
print('Request failed with status code:',
response.status_code)
```

14.5 Rate Limiting

The method of rate limitation is used to regulate and govern the rate at which users or clients can submit requests to an API. It guarantees the stability and

accessibility of the API, guards against misuse, and encourages proper usage. Python provides libraries and methods for implementing efficient rate restriction systems.

For Example: Here's an explanation and an example of implementing rate limiting in Python,
i. Token Bucket Algorithm: - A popular method of rate limitation is the token bucket algorithm. It operates by keeping track of a virtual bucket of tokens, each of which stands for a request. The bucket gets depleted of a token each time a request is made. Further requests are postponed until fresh tokens are added at a set pace if the bucket is empty.
For Example: Example of rate limiting using the "ratelimit" library,

```
from ratelimit import limits, sleep_and_retry
import requests

# Define rate limits: 100 requests per hour
@sleep_and_retry
@limits(calls=100, period=3600) # 100 requests per 3600 seconds (1 hour)
def make_request():
response = requests.get('https://api.example.com/data')
return response

# Make API requests
for _ in range(150):
response = make_request()
print(response.status_code)
```

The "make_request" method is rate limited using the

"ratelimit" library. When the rate cap is reached, the function stops making new requests and instead waits until there are more tokens available.

ii. Exponential Backoff: - Another strategy uses exponential backoff, where clients gradually lengthen the time between retries after receiving replies that exceed the rate limit. This lessens the burden on the API server when it is busiest.
For Example: Example of exponential backoff using the "backoff" library,

```
import requests
from requests.exceptions import RequestException
import backoff


# Define exponential backoff decorator
@backoff.on_exception(backoff.expo, RequestException, max_tries=5)
def make_request():
response = requests.get('https://api.example.com/data')
return response

# Make API requests with exponential backoff
try:
response = make_request()
print(response.status_code)
except RequestException as e:
print('Request failed:', str(e))
```

If a "RequestException" occurs, the "backoff" library is used to apply exponential backoff to the "make_request" method. In the event of failures, the library automatically

extends the time between retries.