GIET

In Pursuit of Innovation

GANAPATI INSTITUTE OF ENGINEERING

&

TECHNOLOGY (POLYTECHNIC)

Mathasahi , Jagatpur , Cuttack - 754200

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

Prepared By:

Miss Smaranika Moharana.

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

Python and Its History

What is Python?

Python is a **high-level, interpreted, general-purpose programming language** known for its simple syntax and readability. It is widely used in web development, data science, artificial intelligence, machine learning, automation, and software development.

Example

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

Output:

```
Hello, World!
```

History of Python

Python was created by **Guido van Rossum** in the late 1980s and officially released in **1991**.

Why Python Was Created

Guido van Rossum wanted to develop a programming language that:

- Was easy to read and write
- Reduced the complexity of programming
- Supported code reuse
- Improved programmer productivity

Origin of the Name

The name **Python** was inspired by the British comedy television show **Monty Python's Flying Circus**, not by the snake.

Evolution of Python

Python 1.0 (1994)

- First major release
- Included functions, modules, and exception handling

Python 2.0 (2000)

- Introduced list comprehensions
- Added garbage collection
- Improved memory management

Python 3.0 (2008)

- Major redesign of the language
- Removed outdated features
- Improved consistency and performance
- Became the standard version for modern development

Key Features of Python

- Simple and easy-to-read syntax
- Interpreted language
- Platform independent
- Object-oriented
- Open source and free
- Large standard library
- Supports multiple programming paradigms

Applications of Python

- Web Development
- Data Science and Analytics
- Artificial Intelligence (AI)
- Machine Learning
- Automation and Scripting
- Cybersecurity
- Scientific Computing
- Game Development

Sample Program

```
name = "Python"
year = 1991

print(name, "was released in", year)
```

Output:

```
Python was released in 1991
```

Python is a **high-level, interpreted, and general-purpose programming language** created by **Guido van Rossum** in 1991. It is known for its simple syntax and readability.

Features of Python

- Easy to learn and use
- Interpreted language (no compilation required)
- Object-Oriented Programming support
- Cross-platform (Windows, Linux, macOS)
- Large standard library
- Open-source and free

Applications of Python

- Web Development
- Data Science and Analytics
- Artificial Intelligence (AI) & Machine Learning
- Automation and Scripting
- Game Development
- Cybersecurity
- Scientific Computing

Example

```
print("Hello, Python!")
```

Output:

```
Hello, Python!
```

2. Setting Up the Python Environment

Python Installation

1. Download Python from:
   Python Official Website
2. Run the installer.
3. Select **"Add Python to PATH"**.
4. Verify installation:

```
python --version
```

Output Example:

```
Python 3.13.0
```

IDEs (Integrated Development Environments)

Popular Python IDEs:

- Visual Studio Code
- PyCharm
- Jupyter Notebook
- Spyder

These tools help write, run, and debug Python programs easily.

3. Python Syntax

Python uses **indentation (spaces)** instead of braces `{}` to define code blocks.

Example

```
age = 18

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

Output:

```
Adult
```

4. Variables

Variables store data values.

Example

```
name = "Rahul"
age = 20

print(name)
print(age)
```

Output:

```
Rahul
20
```

5. Data Types

Python has several built-in data types.

Data Type	Example
int	10
float	3.14
str	"Hello"
bool	True
list	[1,2,3]
tuple	(1,2,3)
dict	{"name":"Rahul"}

Example

```
x = 10
y = 3.14
name = "Python"
flag = True

print(type(x))
print(type(y))
```

6. Operators

Arithmetic Operators

```
a = 10
b = 5

print(a + b)
print(a - b)
print(a * b)
print(a / b)
```

Output:

```
15
5
50
2.0
```

Comparison Operators

```
print(a > b)
print(a == b)
print(a != b)
```

Output:

```
True
False
True
```

Logical Operators

```
x = True
y = False

print(x and y)
print(x or y)
print(not x)
```

Output:

```
False
True
False
```

7. Writing Python Scripts

A Python script is a file with the `.py` extension.

Example: hello.py

```
name = "Python"

print("Welcome to", name)
```

Save the file as:

```
hello.py
```

8. Executing Python Scripts

Using Terminal

```
python hello.py
```

Output:

```
Welcome to Python
```

Using an IDE

1. Open the IDE.
2. Create a Python file.
3. Write the code.
4. Click **Run**.

9. Debugging Python Scripts

Debugging means finding and fixing errors in a program.

Example of an Error

```
num = 10

print(number)
```

Output:

```
NameError: name 'number' is not defined
```

Corrected Code

```
num = 10

print(num)
```

Output:

```
10
```

Using Print Statements for Debugging

```
a = 5
b = 0

print("Value of a:", a)
print("Value of b:", b)

# print(a/b)
```

This helps identify where the problem occurs.

Complete Example Program

```
name = "Rahul"
age = 20

if age >= 18:
    print(name, "is eligible to vote")
else:
    print(name, "is not eligible to vote")
```

unit:2

Control Structures and Functions in Python

Control structures and functions are fundamental concepts in Python programming. They help control the flow of execution of a program and make code reusable, organized, and efficient. Control structures include conditional statements and loops, while functions allow programmers to divide a program into smaller manageable parts.

Conditional Statements: if, else, and elif

Conditional statements are used to make decisions in a program. They allow different blocks of code to execute depending on whether a condition is true or false.

if Statement

The `if` statement executes a block of code only when a specified condition is true.

Syntax

```
if condition:
    statement
```

Example

```
age = 18

if age >= 18:
    print("You are eligible to vote")
```

Output:

```
You are eligible to vote
```

In this example, the condition `age >= 18` is true, so the message is displayed.

if-else Statement

The `else` statement is used when an alternative action should be performed if the condition is false.

Example

```
age = 16

if age >= 18:
    print("Eligible to vote")
else:
    print("Not eligible to vote")
```

Output:

```
Not eligible to vote
```

if-elif-else Statement

When multiple conditions need to be checked, the `elif` (else if) statement is used.

Example

```
marks = 75

if marks >= 90:
    print("Grade A")
elif marks >= 70:
    print("Grade B")
elif marks >= 50:
    print("Grade C")
else:
    print("Fail")
```

Output:

```
Grade B
```

This structure helps the program choose one option from several possibilities.

Loops in Python

Loops are used to execute a block of code repeatedly. They reduce the need for writing the same code multiple times.

for Loop

A `for` loop is used when the number of iterations is known.

Syntax

```
for variable in sequence:
    statement
```

Example

```
for i in range(1, 6):
    print(i)
```

Output:

```
1
2
3
4
5
```

The loop prints numbers from 1 to 5.

while Loop

A `while` loop executes as long as a condition remains true.

Syntax

```
while condition:
    statement
```

Example

```
count = 1

while count <= 5:
    print(count)
    count += 1
```

Output:

```
1
2
3
4
5
```

The loop continues until the condition becomes false.

Nested Loops

A loop inside another loop is called a nested loop. Nested loops are useful for working with tables, matrices, and patterns.

Example

```
for i in range(1, 4):
    for j in range(1, 4):
        print(i, j)
```

Output:

```
1 1
1 2
```

```
1 3
2 1
2 2
2 3
3 1
3 2
3 3
```

The inner loop completes all its iterations for every iteration of the outer loop.

Functions in Python

A function is a reusable block of code that performs a specific task. Functions improve code readability, reduce repetition, and make programs easier to maintain.

Defining a Function

Functions are defined using the `def` keyword.

Syntax

```
def function_name():
    statements
```

Example

```
def greet():
    print("Welcome to Python")

greet()
```

Output:

```
Welcome to Python
```

Function with Parameters

Parameters allow data to be passed into a function.

Example

```
def greet(name):
    print("Hello", name)

greet("Rahul")
```

Output:

```
Hello Rahul
```

Function with Return Value

Functions can return results using the `return` statement.

Example

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

result = add(10, 20)
print(result)
```

Output:

```
30
```

Scope of Variables

The scope of a variable determines where it can be accessed in a program.

Local Variables

Variables declared inside a function are called local variables. They can only be used within that function.

Example

```
def show():
    x = 10
    print(x)

show()
```

Output:

```
10
```

Here, `x` exists only inside the function.

Global Variables

Variables declared outside all functions are called global variables. They can be accessed throughout the program.

Example

```
x = 100

def display():
    print(x)

display()
```

Output:

```
100
```

The variable `x` is available inside and outside the function.

Lambda Functions

A lambda function is an anonymous function that can have any number of arguments but only one expression. It is useful for short operations that do not require a full function definition.

Syntax

```
lambda arguments: expression
```

Example

```
square = lambda x: x * x

print(square(5))
```

Output:

```
25
```

Equivalent normal function:

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

Lambda functions are commonly used with functions like `map()`, `filter()`, and `sorted()`.

Example

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

result = list(map(lambda x: x * 2, numbers))

print(result)
```

Output:

```
[2, 4, 6, 8]
```

Recursion

Recursion is a technique in which a function calls itself to solve a problem. A recursive function must have a base case to stop the recursion; otherwise, it will continue indefinitely.

Example: Factorial Using Recursion

```
def factorial(n):
    if n == 1:
        return 1
    else:
        return n * factorial(n - 1)

print(factorial(5))
```

Output:

```
120
```

Working of Recursion

For `factorial(5)`:

```
5 × factorial(4)
5 × 4 × factorial(3)
5 × 4 × 3 × factorial(2)
5 × 4 × 3 × 2 × factorial(1)
5 × 4 × 3 × 2 × 1
= 120
```

Recursion is commonly used in mathematical calculations, tree traversal, searching algorithms, and divide-and-conquer problems.

Unit:3

Data Structures in Python: Lists, Tuples, Sets, and Dictionaries, List and Dictionary Comprehensions, String Manipulation, and Python Collections Module

Data structures are specialized formats used to organize, store, and manage data efficiently. Python provides several built-in data structures such as Lists, Tuples, Sets, and Dictionaries that help programmers handle data according to different requirements. These data structures are essential because they improve the efficiency of data storage, retrieval, and processing. Python also provides comprehensions for creating collections in a concise manner, powerful string manipulation techniques, and the Collections module for advanced data handling.

Lists

A list is an ordered and mutable collection of elements. Lists can contain elements of different data types such as integers, strings, floating-point numbers, and even other lists. Since lists are mutable, their contents can be modified after creation. Lists are enclosed within square brackets `[]`.

Lists support various operations such as insertion, deletion, updating, sorting, searching, and traversal. Common methods include `append()`, `insert()`, `remove()`, `pop()`, `sort()`, and `reverse()`. Lists allow duplicate values and maintain the order of insertion.

Example

```
numbers = [10, 20, 30, 40]

numbers.append(50)

print(numbers)
```

Output:

```
[10, 20, 30, 40, 50]
```

Applications of Lists

Lists are widely used for storing collections of items, maintaining records, processing numerical data, implementing stacks and queues, and managing dynamic data where frequent modifications are required.

Tuples

A tuple is an ordered collection of elements similar to a list, but unlike lists, tuples are immutable. Once a tuple is created, its elements cannot be modified, added, or removed. Tuples are enclosed within parentheses `()`.

Tuples are generally faster than lists because they cannot be modified. They are commonly used when data should remain constant throughout the execution of a program.

Example

```
student = ("Rahul", 101, "CSE")

print(student)
```

Output:

```
('Rahul', 101, 'CSE')
```

Applications of Tuples

Tuples are used for storing fixed information, representing coordinates, returning multiple values from functions, and protecting data from accidental modification.

Sets

A set is an unordered collection of unique elements. Duplicate values are automatically removed when stored in a set. Sets are enclosed within curly braces `{}`.

Sets support mathematical operations such as union, intersection, difference, and symmetric difference. Since sets are unordered, indexing is not supported.

Example

```
numbers = {1, 2, 3, 4, 4, 5}

print(numbers)
```

Output:

```
{1, 2, 3, 4, 5}
```

Set Operations

```
A = {1, 2, 3}
B = {3, 4, 5}

print(A.union(B))
print(A.intersection(B))
```

Output:

```
{1, 2, 3, 4, 5}
{3}
```

Applications of Sets

Sets are useful for removing duplicates from data, performing mathematical set operations, membership testing, and handling unique collections of items.

Dictionaries

A dictionary is a collection of key-value pairs. Each key in a dictionary must be unique, and values are accessed using their corresponding keys. Dictionaries are enclosed within curly braces `{}`.

Dictionaries provide very fast data retrieval because values are accessed through keys rather than indexes. They are one of the most frequently used data structures in Python.

Example

```
student = {
    "name": "Rahul",
    "age": 20,
    "course": "CSE"
}

print(student["name"])
```

Output:

```
Rahul
```

Dictionary Operations

Operations on dictionaries include adding, deleting, updating, and accessing elements using keys. Methods such as `keys()`, `values()`, `items()`, `get()`, `pop()`, and `update()` are commonly used.

Applications of Dictionaries

Dictionaries are used in databases, configuration files, user profiles, inventories, contact lists, and any situation where information needs to be stored as key-value pairs.

List Comprehensions

List comprehension is a concise and efficient way to create lists in Python. It reduces the amount of code required to generate a list from an iterable object.

The general syntax is:

```
[expression for item in iterable]
```

Example

```
squares = [x * x for x in range(1, 6)]

print(squares)
```

Output:

```
[1, 4, 9, 16, 25]
```

List comprehensions can also include conditions.

```
even_numbers = [x for x in range(1, 11) if x % 2 == 0]

print(even_numbers)
```

Output:

```
[2, 4, 6, 8, 10]
```

Advantages of List Comprehensions

List comprehensions make code shorter, more readable, and often more efficient than traditional loops.

Dictionary Comprehensions

Dictionary comprehension is a compact method for creating dictionaries. It works similarly to list comprehension but generates key-value pairs.

The general syntax is:

```
{key:value for item in iterable}
```

Example

```
squares = {x: x*x for x in range(1, 6)}

print(squares)
```

Output:

```
{1: 1, 2: 4, 3: 9, 4: 16, 5: 25}
```

Example with Condition

```
even_squares = {x: x*x for x in range(1, 11) if x % 2 == 0}

print(even_squares)
```

Output:

```
{2: 4, 4: 16, 6: 36, 8: 64, 10: 100}
```

Advantages of Dictionary Comprehensions

Dictionary comprehensions simplify the process of creating dictionaries and improve code readability and performance.

Working with Strings

A string is a sequence of characters enclosed in single quotes, double quotes, or triple quotes. Strings are immutable, meaning their contents cannot be changed after creation.

Strings are one of the most important data types in Python because they are used to store and manipulate textual data.

Example

```
text = "Python Programming"

print(text)
```

Output:

```
Python Programming
```

String Methods

Python provides numerous built-in methods for string manipulation.

Converting to Uppercase

```
text = "python"

print(text.upper())
```

Output:

```
PYTHON
```

Converting to Lowercase

```
text = "PYTHON"

print(text.lower())
```

Output:

```
python
```

Replacing Text

```
text = "Hello World"

print(text.replace("World", "Python"))
```

Output:

```
Hello Python
```

Splitting Strings

```
text = "Python is easy"

print(text.split())
```

Output:

```
['Python', 'is', 'easy']
```

Joining Strings

```
words = ["Python", "is", "easy"]

print(" ".join(words))
```

Output:

```
Python is easy
```

String Manipulation Techniques

String manipulation involves modifying or processing textual data. Common techniques include concatenation, repetition, slicing, searching, formatting, and replacement.

String Concatenation

```
first = "Hello"
second = "World"

print(first + " " + second)
```

Output:

```
Hello World
```

String Slicing

```
text = "Programming"

print(text[0:6])
```

Output:

```
Progra
```

String Repetition

```
print("Python " * 3)
```

Output:

```
Python Python Python
```

String manipulation is widely used in data processing, web development, file handling, and natural language processing applications.

Introduction to Python's Collections Module

The Collections module is a specialized library that extends Python's built-in container data types. It provides additional data structures that offer improved functionality and performance for specific tasks.

Some important classes available in the Collections module are Counter, defaultdict, deque, and namedtuple.

Counter

Counter is used for counting the frequency of elements in a collection.

```
from collections import Counter

data = ['a', 'b', 'a', 'c', 'a']

count = Counter(data)

print(count)
```

Output:

```
Counter({'a': 3, 'b': 1, 'c': 1})
```

Counter is useful in data analysis, frequency calculations, and statistical applications.

defaultdict

defaultdict automatically provides a default value when a key does not exist.

```
from collections import defaultdict

d = defaultdict(int)

d["a"] += 1

print(d)
```

Output:

```
defaultdict(<class 'int'>, {'a': 1})
```

defaultdict simplifies dictionary handling by avoiding key errors.

deque

A deque (double-ended queue) allows insertion and deletion of elements from both ends efficiently.

```
from collections import deque

dq = deque([1, 2, 3])

dq.append(4)
dq.appendleft(0)

print(dq)
```

Output:

```
deque([0, 1, 2, 3, 4])
```

Deques are commonly used in queue implementations, scheduling systems, and breadth-first search algorithms.

namedtuple

namedtuple creates tuple-like objects whose elements can be accessed using names instead of indexes.

```
from collections import namedtuple

Student = namedtuple("Student", ["name", "age"])

s = Student("Rahul", 20)

print(s.name)
```

Output:

```
Rahul
```

Unit:4

File Handling and Modules in Python: File Operations, Reading, Writing, Appending Files, Working with CSV and JSON Files, Built-In Modules, and Custom Modules

File handling and modules are important features of Python that help programmers store data permanently and organize programs efficiently. Files are used to save information so that it can be accessed even after a program terminates. Modules allow code to be divided into separate reusable components, making programs easier to manage and maintain. Python provides extensive support for file handling operations and includes many built-in modules that simplify programming tasks.

File Handling in Python

File handling is the process of creating, opening, reading, writing, and closing files. Files are used for permanent storage of data because variables store information only while the program is running.

Python provides built-in functions and methods for performing file operations. Before performing any operation on a file, it must be opened using the `open()` function.

Opening a File

The syntax for opening a file is:

```
file = open("filename", "mode")
```

The mode specifies how the file will be used.

Common file modes include:

Mode	Description
r	Read mode
w	Write mode
a	Append mode
x	Create a new file
rb	Read binary file
wb	Write binary file

Example

```
file = open("data.txt", "r")
```

This opens the file in read mode.

Reading Files

Reading a file means retrieving the contents stored in it. Python provides several methods for reading data from files.

Reading Entire File

```
file = open("data.txt", "r")

content = file.read()

print(content)

file.close()
```

Reading One Line

```
file = open("data.txt", "r")

print(file.readline())
```

```
file.close()
```

Reading All Lines

```
file = open("data.txt", "r")

print(file.readlines())

file.close()
```

Reading files is useful in applications such as processing text documents, analyzing data, reading configuration files, and handling logs.

Writing Files

Writing means storing new data in a file. When a file is opened in write mode (`w`), existing contents are overwritten.

Example

```
file = open("data.txt", "w")

file.write("Welcome to Python")

file.close()
```

After execution, the file will contain the text:

```
Welcome to Python
```

Writing files is commonly used for generating reports, storing program outputs, and saving user data.

Appending Files

Appending means adding new data to the end of an existing file without deleting its current contents. This is done using append mode (`a`).

Example

```
file = open("data.txt", "a")

file.write("\nPython is easy to learn")

file.close()
```

If the file already contains:

```
Welcome to Python
```

After appending:

```
Welcome to Python
Python is easy to learn
```

Appending is useful for maintaining logs, recording transactions, and continuously storing new information.

Using the with Statement

Python provides the `with` statement for file handling. It automatically closes the file after the operation is completed.

Example

```
with open("data.txt", "r") as file:
    content = file.read()
    print(content)
```

Using `with` is recommended because it prevents resource leaks and simplifies file management.

Working with CSV Files

CSV stands for Comma-Separated Values. CSV files are commonly used for storing tabular data such as spreadsheets, databases, and reports.

A CSV file may look like:

```
Name,Age,Course
Rahul,20,CSE
Priya,21,IT
```

Python provides the built-in `csv` module for handling CSV files.

Reading a CSV File

```
import csv

with open("students.csv", "r") as file:
    reader = csv.reader(file)

    for row in reader:
        print(row)
```

Output:

```
['Name', 'Age', 'Course']
['Rahul', '20', 'CSE']
['Priya', '21', 'IT']
```

Writing a CSV File

```
import csv

with open("students.csv", "w", newline="") as file:
    writer = csv.writer(file)

    writer.writerow(["Name", "Age", "Course"])
    writer.writerow(["Rahul", 20, "CSE"])
```

CSV files are widely used in data analysis, machine learning, business applications, and database management systems.

Working with JSON Files

JSON stands for JavaScript Object Notation. It is a lightweight format used for storing and exchanging data between applications.

Example JSON data:

```
{
    "name": "Rahul",
    "age": 20,
    "course": "CSE"
}
```

Python provides the built-in `json` module for working with JSON files.

Writing JSON Data

```
import json

student = {
    "name": "Rahul",
    "age": 20,
    "course": "CSE"
}

with open("student.json", "w") as file:
    json.dump(student, file)
```

Reading JSON Data

```
import json

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

print(data)
```

Output:

```
{'name': 'Rahul', 'age': 20, 'course': 'CSE'}
```

JSON files are extensively used in web development, APIs, cloud applications, and data exchange systems.

Python Modules

A module is a file containing Python code such as functions, variables, and classes that can be reused in other programs. Modules help organize code into separate files, making large applications easier to develop and maintain.

Python provides two types of modules:

1. Built-in Modules
2. Custom Modules

Built-In Modules

Built-in modules are pre-installed with Python and provide ready-made functionality for various tasks.

Math Module

The `math` module contains mathematical functions.

Example

```
import math

print(math.sqrt(25))
print(math.factorial(5))
```

Output:

```
5.0
120
```

Common functions include:

- sqrt()
- factorial()
- ceil()
- floor()
- sin()
- cos()
- log()

The math module is useful for scientific calculations and mathematical computations.

OS Module

The `os` module provides functions for interacting with the operating system.

Example

```
import os

print(os.getcwd())
```

Output:

```
Current working directory path
```

Common operations include:

- Creating directories

- Removing files
- Renaming files
- Accessing environment variables

The OS module is important in system programming and file management.

Datetime Module

The `datetime` module is used for handling dates and times.

Example

```
from datetime import datetime

now = datetime.now()

print(now)
```

Output:

```
2026-06-25 10:30:15
```

Common applications include:

- Scheduling tasks
- Calculating age
- Managing timestamps
- Tracking events

Importing Modules

Modules can be imported in several ways.

Import Entire Module

```
import math

print(math.sqrt(16))
```

Import Specific Function

```
from math import sqrt

print(sqrt(16))
```

Import with Alias

```
import math as m

print(m.sqrt(16))
```

Output:

```
4.0
```

Creating Custom Modules

A custom module is a Python file created by the programmer to store reusable code.

Step 1: Create a Module

Create a file named `calculator.py`

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

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

Step 2: Use the Module

Create another file:

```
import calculator

print(calculator.add(10, 20))
print(calculator.multiply(5, 4))
```

Output:

```
30
20
```

Custom modules help improve code organization, reduce duplication, and promote reusability.

Advantages of Modules

- Code reusability
- Better program organization
- Easier maintenance
- Improved readability
- Faster development
- Reduced code duplication

Object-Oriented Programming (OOP) in Python: Classes and Objects, Encapsulation, Inheritance, Polymorphism, Magic Methods, Operator Overloading, and Exception Handling

Object-Oriented Programming (OOP) is a programming paradigm that organizes software design around objects rather than functions and logic. An object represents a real-world entity and contains both data and methods that

operate on that data. OOP helps programmers create modular, reusable, scalable, and maintainable applications. Python fully supports Object-Oriented Programming and provides several important concepts such as classes, objects, encapsulation, inheritance, polymorphism, magic methods, operator overloading, and exception handling.

Classes and Objects

A class is a blueprint or template used to create objects. It defines the properties (attributes) and behaviors (methods) that objects created from the class will possess. An object is an instance of a class and represents a real-world entity.

For example, a class named Student may contain attributes such as name and roll number and methods such as display information. Individual students are considered objects of the Student class.

Creating a Class and Object

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

    def display(self):
        print("Name:", self.name)
        print("Roll:", self.roll)

s1 = Student("Rahul", 101)

s1.display()
```

Output:

```
Name: Rahul
Roll: 101
```

In the above example, `Student` is a class and `s1` is an object. The `__init__()` method is called automatically when an object is created and is used to initialize object attributes.

Advantages of Classes and Objects

- Code reusability
- Better organization of programs
- Easy maintenance
- Improved security and flexibility
- Representation of real-world entities

Encapsulation

Encapsulation is the process of binding data and methods together within a single unit, called a class. It also restricts direct access to certain data members, protecting them from accidental modification.

In Python, encapsulation is achieved using private variables and methods. Private members are prefixed with double underscores (__).

Example

```
class BankAccount:
    def __init__(self):
        self.__balance = 0

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

    def get_balance(self):
        return self.__balance

account = BankAccount()

account.deposit(5000)

print(account.get_balance())
```

Output:

```
5000
```

In this example, the variable `__balance` cannot be accessed directly from outside the class. Access is provided through methods.

Benefits of Encapsulation

- Protects data from unauthorized access
- Improves security
- Enhances maintainability
- Provides controlled access to data

Inheritance

Inheritance is a mechanism through which one class acquires the properties and methods of another class. The existing class is called the parent class or base class, while the new class is called the child class or derived class.

Inheritance promotes code reuse and reduces duplication.

Example

```
class Person:
    def show(self):
        print("I am a Person")

class Student(Person):
    pass

s = Student()

s.show()
```

Output:

```
I am a Person
```

The Student class inherits the `show()` method from the Person class.

Types of Inheritance in Python

1. Single Inheritance
2. Multiple Inheritance
3. Multilevel Inheritance
4. Hierarchical Inheritance
5. Hybrid Inheritance

Example of Multilevel Inheritance

```
class Grandparent:
    def display1(self):
        print("Grandparent")

class Parent(Grandparent):
    def display2(self):
        print("Parent")

class Child(Parent):
    def display3(self):
        print("Child")

c = Child()

c.display1()
c.display2()
c.display3()
```

Output:

```
Grandparent
Parent
Child
```

Benefits of Inheritance

- Code reusability
- Reduced redundancy
- Easy maintenance
- Improved extensibility

Polymorphism

Polymorphism means "many forms." It allows a single interface to be used for different types of objects. The same method name can perform different tasks depending on the object that invokes it.

Polymorphism increases flexibility and simplifies program design.

Method Overriding Example

```
class Animal:
    def sound(self):
        print("Animal makes a sound")

class Dog(Animal):
    def sound(self):
        print("Dog barks")

class Cat(Animal):
    def sound(self):
        print("Cat meows")

d = Dog()
c = Cat()

d.sound()
c.sound()
```

Output:

```
Dog barks
Cat meows
```

The same method `sound()` behaves differently for different objects.

Advantages of Polymorphism

- Increases flexibility
- Improves code readability
- Simplifies maintenance
- Supports dynamic behavior

Magic Methods

Magic methods, also known as special methods or dunder methods, are predefined methods in Python that begin and end with double underscores. These methods allow programmers to define the behavior of objects for built-in operations.

Some common magic methods are:

Method	Purpose
`__init__()`	Constructor
`__str__()`	String representation
`__len__()`	Length of object
`__add__()`	Addition operation
`__sub__()`	Subtraction operation
`__eq__()`	Equality comparison

Example of str()

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

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

s = Student("Rahul")

print(s)
```

Output:

```
Rahul
```

Without `__str__()`, Python would display the memory address of the object.

Importance of Magic Methods

- Customize object behavior
- Integrate with Python built-in functions
- Improve readability
- Enable operator overloading

Operator Overloading

Operator overloading allows programmers to redefine how operators behave for user-defined objects. This is achieved using magic methods.

For example, the `+` operator normally adds numbers. Using operator overloading, it can be customized to work with objects.

Example

```
class Number:
    def __init__(self, value):
        self.value = value

    def __add__(self, other):
        return self.value + other.value

n1 = Number(10)
n2 = Number(20)

print(n1 + n2)
```

Output:

```
30
```

Here, the `+` operator uses the `__add__()` magic method.

Common Operator Overloading Methods

Operator	Magic Method
+	__add__()

Operator	Magic Method
-	`__sub__()`
*	`__mul__()`
/	`__truediv__()`
==	`__eq__()`
<	`__lt__()`
>	`__gt__()`

Benefits of Operator Overloading

- Improves code readability
- Makes objects behave like built-in types
- Simplifies complex operations

Exception Handling in Python

Exception handling is the process of handling runtime errors so that a program does not terminate unexpectedly. An exception is an event that occurs during program execution and disrupts the normal flow of instructions.

Common exceptions include:

- ZeroDivisionError
- ValueError
- TypeError
- IndexError
- FileNotFoundError
- KeyError

Python uses the `try`, `except`, `else`, and `finally` blocks for exception handling.

try and except

```
try:
    result = 10 / 0
except ZeroDivisionError:
    print("Cannot divide by zero")
```

Output:

```
Cannot divide by zero
```

The program handles the error instead of crashing.

Multiple Exceptions

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

```
except ZeroDivisionError:
    print("Division by zero not allowed")
```

else Block

The `else` block executes if no exception occurs.

```
try:
    num = 10 / 2
except ZeroDivisionError:
    print("Error")
else:
    print("Operation successful")
```

Output:

```
Operation successful
```

finally Block

The `finally` block executes whether an exception occurs or not.

```
try:
    file = open("data.txt")
except FileNotFoundError:
    print("File not found")
finally:
    print("Program completed")
```

Output:

```
File not found
Program completed
```

Raising Exceptions

Python allows programmers to create exceptions using the `raise` statement.

```
age = -5

if age < 0:
    raise ValueError("Age cannot be negative")
```

Output:

```
ValueError: Age cannot be negative
```

Advantages of Exception Handling

- Prevents abnormal program termination
- Improves reliability
- Simplifies error management
- Enhances user experience
- Helps in debugging

Advanced Python and Applications: NumPy, Pandas, Matplotlib, Web Scraping, Automation Scripting, and Real-World Python Projects

Advanced Python extends the capabilities of basic Python programming by providing powerful libraries, automation tools, and techniques for handling real-world applications. Python has become one of the most popular programming languages because of its extensive ecosystem of libraries that simplify tasks such as data analysis, scientific computing, visualization, web scraping, automation, artificial intelligence, and software development. Some of the most widely used Python libraries are NumPy, Pandas, and Matplotlib. Python is also extensively used for web scraping through libraries such as requests and Beautiful Soup, and for automation using modules such as os and shutil. These tools allow programmers to develop practical solutions for real-world problems efficiently.

Introduction to Python Libraries

A library is a collection of pre-written code that provides specific functionality and can be reused in different programs. Python libraries reduce development time because programmers do not need to write every function from scratch.

Python offers thousands of libraries for different purposes, including:

- Data Analysis
- Machine Learning
- Artificial Intelligence
- Web Development
- Scientific Computing
- Automation
- Networking
- Data Visualization

Among these, NumPy, Pandas, and Matplotlib are considered essential for data processing and analysis.

NumPy Library

NumPy (Numerical Python) is a powerful library used for numerical and mathematical computations. It provides support for multidimensional arrays and a large collection of mathematical functions.

NumPy is faster than standard Python lists because it is implemented using optimized low-level code. It is widely used in scientific computing, machine learning, image processing, and data analysis.

Features of NumPy

- Efficient multidimensional arrays
- High-performance mathematical operations
- Matrix calculations

- Statistical functions
- Linear algebra support
- Random number generation

Example

```
import numpy as np

arr = np.array([10, 20, 30, 40])

print(arr)
```

Output:

```
[10 20 30 40]
```

Mathematical Operations

```
import numpy as np

arr = np.array([1, 2, 3])
print(arr * 2)
```

Output:

```
[2 4 6]
```

Applications of NumPy

- Scientific calculations
- Machine learning
- Image processing
- Data analysis
- Engineering simulations

NumPy forms the foundation of many advanced Python libraries including Pandas and Scikit-Learn.

Pandas Library

Pandas is a powerful data analysis and manipulation library built on top of NumPy. It provides data structures such as Series and DataFrame for organizing and processing structured data.

A DataFrame is similar to a spreadsheet or database table containing rows and columns.

Features of Pandas

- Data cleaning
- Data transformation
- Data filtering
- Handling missing values
- Reading and writing files
- Statistical analysis

Example

```
import pandas as pd

data = {
    "Name": ["Rahul", "Priya"],
    "Age": [20, 21]
}

df = pd.DataFrame(data)

print(df)
```

Output:

```
    Name   Age
0  Rahul   20
1  Priya   21
```

Reading CSV Files

```
import pandas as pd

df = pd.read_csv("students.csv")

print(df.head())
```

The `head()` function displays the first few rows of the dataset.

Applications of Pandas

- Data analysis
- Business intelligence
- Financial analysis
- Data cleaning
- Machine learning preprocessing

Pandas is one of the most important libraries used by data scientists and analysts.

Matplotlib Library

Matplotlib is a data visualization library used for creating charts, graphs, and plots. It helps represent data visually, making it easier to understand patterns and trends.

Features of Matplotlib

- Line charts
- Bar charts
- Pie charts
- Histograms
- Scatter plots
- Customizable visualizations

Example

```
import matplotlib.pyplot as plt

x = [1, 2, 3, 4]
y = [10, 20, 30, 40]

plt.plot(x, y)

plt.show()
```

The program displays a line graph representing the relationship between x and y values.

Bar Chart Example

```
import matplotlib.pyplot as plt

subjects = ["Math", "Science", "English"]
marks = [80, 90, 75]

plt.bar(subjects, marks)

plt.show()
```

Applications of Matplotlib

- Business reporting
- Scientific research
- Data analytics
- Machine learning visualization
- Performance monitoring

Visualization helps convert raw data into meaningful information.

Basics of Web Scraping

Web scraping is the process of extracting information from websites automatically using software programs. It allows data to be collected from web pages for analysis and processing.

Python provides several libraries for web scraping, including:

- requests
- Beautiful Soup
- Selenium

The most common combination is requests and Beautiful Soup.

Requests Library

The requests library is used to send HTTP requests and retrieve web page content.

Example

```
import requests

response = requests.get("https://example.com")
```

```
print(response.status_code)
```

Output:

```
200
```

A status code of 200 indicates that the request was successful.

Beautiful Soup

Beautiful Soup is used to parse HTML and XML documents and extract specific information.

Example

```
from bs4 import BeautifulSoup
import requests

response = requests.get("https://example.com")

soup = BeautifulSoup(response.text, "html.parser")

print(soup.title.text)
```

Output:

```
Example Domain
```

Applications of Web Scraping

- Price comparison websites
- News aggregation
- Market research
- Data collection
- Job listing analysis
- Product information gathering

Web scraping is widely used in data science and business intelligence.

Scripting for Automation

Automation refers to performing repetitive tasks automatically without human intervention. Python is one of the most popular languages for automation because of its simplicity and extensive standard library.

Automation scripts can:

- Rename files
- Organize folders
- Backup data
- Generate reports
- Send emails
- Process data automatically

Python provides modules such as os and shutil for automation tasks.

OS Module

The os module provides functions for interacting with the operating system.

Features

- Creating directories
- Renaming files
- Removing files
- Accessing environment variables
- Navigating file systems

Example

```
import os

print(os.getcwd())
```

Output:

```
Current Working Directory
```

Creating a Directory

```
import os

os.mkdir("NewFolder")
```

This creates a new folder named NewFolder.

Listing Files

```
import os

print(os.listdir())
```

This displays all files and folders in the current directory.

Shutil Module

The shutil module is used for high-level file operations such as copying, moving, and deleting files.

Copying a File

```
import shutil

shutil.copy("file1.txt", "file2.txt")
```

Moving a File

```
import shutil

shutil.move("file1.txt", "Backup/")
```

Deleting a Directory

```
import shutil

shutil.rmtree("OldFolder")
```

Applications of Shutil

- File backups
- File organization
- Data migration
- Automated maintenance

The os and shutil modules together provide powerful automation capabilities.

Mini-Project: Developing a Python Script for a Real-World Problem

One of the best ways to learn Python is through practical projects. A mini-project combines multiple concepts and demonstrates how Python can solve real-world problems.

Problem Statement

Suppose a user has hundreds of files in a folder and wants to automatically organize them according to file type.

Solution

A Python script can scan the folder, identify file extensions, create separate folders, and move files accordingly.

Example Script

```
import os
import shutil

source_folder = "Downloads"

for file in os.listdir(source_folder):

    path = os.path.join(source_folder, file)

    if os.path.isfile(path):

        extension = file.split(".")[-1]

        folder = os.path.join(source_folder, extension)

        if not os.path.exists(folder):
            os.mkdir(folder)

        shutil.move(path, os.path.join(folder, file))
```

Working of the Project

1. Read all files from the folder.
2. Determine each file's extension.
3. Create a folder for each extension.
4. Move files into their respective folders.
5. Automatically organize data without manual effort.

Benefits of the Project

- Saves time
- Reduces manual work
- Improves file management
- Demonstrates automation concepts
- Combines os and shutil modules

NumPy and **Pandas** are two of the most important Python libraries used in data analysis, scientific computing, machine learning, and data science. Pandas is built on top of NumPy and provides powerful tools for handling and analyzing structured data.

NumPy

NumPy (Numerical Python) is a library used for numerical computations and mathematical operations. It provides support for multidimensional arrays and high-performance operations on large datasets.

Features of NumPy

- Fast numerical computations
- Multidimensional arrays
- Mathematical and statistical functions
- Matrix operations
- Linear algebra support
- Random number generation

Creating a NumPy Array

```
import numpy as np

arr = np.array([10, 20, 30, 40])

print(arr)
```

Output:

```
[10 20 30 40]
```

Mathematical Operations

```
import numpy as np

arr = np.array([1, 2, 3, 4])

print(arr + 5)
print(arr * 2)
```

Output:

```
[6 7 8 9]
[2 4 6 8]
```

Two-Dimensional Array

```
import numpy as np

matrix = np.array([[1, 2],
                   [3, 4]])

print(matrix)
```

Output:

```
[[1 2]
 [3 4]]
```

Useful NumPy Functions

```
import numpy as np

arr = np.array([10, 20, 30, 40])

print(np.mean(arr))
print(np.max(arr))
print(np.min(arr))
print(np.sum(arr))
```

Output:

```
25.0
40
10
100
```

Applications of NumPy

- Scientific computing
- Machine learning
- Data analysis
- Image processing
- Engineering calculations
- Artificial Intelligence

Pandas

Pandas is a powerful library used for data manipulation and analysis. It provides two main data structures:

1. Series (One-dimensional data)
2. DataFrame (Two-dimensional tabular data)

Pandas makes it easy to work with large datasets and perform operations such as filtering, sorting, grouping, and data cleaning.

Features of Pandas

- Data cleaning and preprocessing
- Handling missing values
- Reading and writing files
- Data filtering and sorting
- Statistical analysis
- Data visualization support

Creating a Series

```
import pandas as pd

data = pd.Series([10, 20, 30, 40])

print(data)
```

Output:

```
0    10
1    20
2    30
3    40
dtype: int64
```

Creating a DataFrame

```
import pandas as pd

data = {
    "Name": ["Rahul", "Priya"],
    "Age": [20, 21]
}

df = pd.DataFrame(data)

print(df)
```

Output:

```
    Name  Age
0  Rahul   20
1  Priya   21
```

Accessing Columns

```
print(df["Name"])
```

Output:

```
0    Rahul
1    Priya
Name: Name, dtype: object
```

Reading a CSV File

```
import pandas as pd

df = pd.read_csv("students.csv")

print(df.head())
```

The `head()` function displays the first five rows of the dataset.

Statistical Functions

```
import pandas as pd

data = {
    "Marks": [80, 90, 75, 85]
}

df = pd.DataFrame(data)

print(df["Marks"].mean())
print(df["Marks"].max())
```

Output:

```
82.5
90
```

Applications of Pandas

- Data analysis
- Business intelligence
- Financial analysis
- Data cleaning
- Machine learning preprocessing
- Report generation

Object-Oriented Programming (OOP) in Python: Classes, Objects, Encapsulation, Inheritance, and Polymorphism

Object-Oriented Programming (OOP) is a programming approach in which software is designed using objects and classes. It helps in organizing complex programs into smaller, reusable, and logical components. Python supports OOP fully and allows developers to model real-world entities using classes and objects. The main concepts of OOP include classes and objects, encapsulation, inheritance, and polymorphism.

Classes and Objects

A class is a blueprint or template used to create objects. It defines properties (attributes) and behaviors (methods) that the objects created from it will have. An object is an instance of a class and represents a real-world entity.

For example, a “Car” class can define attributes like color, model, and speed, and methods like start() and stop(). Each individual car is an object of that class.

Creating a Class and Object

```
class Car:
    def __init__(self, brand, speed):
        self.brand = brand
        self.speed = speed

    def display(self):
        print("Brand:", self.brand)
        print("Speed:", self.speed)

c1 = Car("Toyota", 120)

c1.display()
```

Output:

```
Brand: Toyota
Speed: 120
```

In this example, `Car` is a class, and `c1` is an object. The `__init__` method initializes object attributes when the object is created.

Importance of Classes and Objects

- Helps in representing real-world entities
- Improves code reusability
- Makes programs modular and organized
- Enhances maintainability and scalability

Encapsulation

Encapsulation is the concept of bundling data and methods that operate on that data into a single unit called a class. It also restricts direct access to some components of an object to protect data integrity.

In Python, encapsulation is implemented using private variables, which are defined using double underscores (__).

Example of Encapsulation

```
class Account:
    def __init__(self):
        self.__balance = 0

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

    def get_balance(self):
        return self.__balance

acc = Account()

acc.deposit(1000)

print(acc.get_balance())
```

Output:

```
1000
```

Here, `__balance` is a private variable and cannot be accessed directly outside the class. It can only be modified using methods like `deposit()`.

Advantages of Encapsulation

- Data security and protection
- Controlled access to data
- Improved code maintainability
- Reduced complexity

Encapsulation ensures that internal data is hidden from outside interference and is accessed only through defined methods.

Inheritance

Inheritance is a mechanism in which one class acquires the properties and methods of another class. The class that inherits is called the child class, and the class being inherited from is called the parent class.

Inheritance promotes code reuse and reduces redundancy.

Example of Inheritance

```
class Animal:
    def sound(self):
        print("Animal makes sound")

class Dog(Animal):
    pass

d = Dog()

d.sound()
```

Output:

```
Animal makes sound
```

Here, the `Dog` class inherits the `sound()` method from the `Animal` class.

Types of Inheritance

1. Single inheritance
2. Multiple inheritance
3. Multilevel inheritance
4. Hierarchical inheritance
5. Hybrid inheritance

Example of Multilevel Inheritance

```
class A:
    def showA(self):
```

```
        print("Class A")

class B(A):
    def showB(self):
        print("Class B")

class C(B):
    def showC(self):
        print("Class C")

obj = C()

obj.showA()
obj.showB()
obj.showC()
```

Output:

```
Class A
Class B
Class C
```

Advantages of Inheritance

- Code reusability
- Reduces redundancy
- Easier maintenance
- Supports hierarchical classification

Inheritance helps build relationships between classes and allows extending existing functionality without modifying original code.

Polymorphism

Polymorphism means “many forms.” It allows the same method or operation to behave differently based on the object that is calling it. It improves flexibility and allows a single interface to represent different types of behavior.

Method Overriding Example

```
class Bird:
    def fly(self):
        print("Bird can fly")

class Eagle(Bird):
    def fly(self):
        print("Eagle flies high")

class Penguin(Bird):
    def fly(self):
        print("Penguin cannot fly")

b1 = Eagle()
b2 = Penguin()

b1.fly()
b2.fly()
```

Output:

```
Eagle flies high
Penguin cannot fly
```

Here, the same method `fly()` behaves differently depending on the object.

Types of Polymorphism

1. Compile-time polymorphism (method overloading concept)
2. Runtime polymorphism (method overriding)

Advantages of Polymorphism

- Increases flexibility
- Improves code readability
- Supports dynamic behavior
- Reduces complexity in large systems

Polymorphism allows a single interface to be used for different data types or classes.

Functions in Python A function in Python is a reusable block of code that performs a specific task. Functions help in making programs more organized, modular, and easier to maintain. Instead of writing the same code again and again, we can define a function once and use it multiple times whenever needed.

Python provides both built-in functions (like `print()`, `len()`, `sum()`) and user-defined functions created by programmers.

Types of Functions in Python

1. Built-in functions
2. User-defined functions
3. Lambda (anonymous) functions

1. User-Defined Functions

User-defined functions are created by the programmer using the `def` keyword.

Syntax

```
def function_name(parameters):
    statement
    return value
```

Example of Simple Function

```
def greet():
    print("Hello, Welcome to Python")

greet()
```

Output:

```
Hello, Welcome to Python
```

Here, `greet()` is a function that prints a message.

Function with Parameters

Functions can take input values called parameters.

Example

```
def add(a, b):
    print("Sum:", a + b)

add(10, 20)
```

Output:

```
Sum: 30
```

Function with Return Value

A function can return a value using the `return` statement.

Example

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

result = multiply(5, 4)
print(result)
```

Output:

```
20
```

Here, the function returns a value instead of directly printing it.

Default Arguments in Functions

Default arguments allow a function to use a default value if no argument is passed.

Example

```
def greet(name="Guest"):
    print("Hello", name)

greet()
greet("Rahul")
```

Output:

```
Hello Guest
Hello Rahul
```

Scope of Variables in Functions

Variable scope defines where a variable can be accessed.

1. Local Variable

Variables defined inside a function are local and can be used only inside that function.

Example

```
def show():
    x = 10
    print(x)

show()
```

2. Global Variable

Variables defined outside functions are global and can be accessed anywhere.

Example

```
x = 100

def display():
    print(x)

display()
```

Lambda Functions (Anonymous Functions)

A lambda function is a small anonymous function defined using the `lambda` keyword. It can have any number of arguments but only one expression.

Syntax

```
lambda arguments: expression
```

Example

```
square = lambda x: x * x

print(square(5))
```

Output:

```
25
```

Advantages of Functions

- Code reusability
- Reduces repetition
- Improves readability
- Easier debugging and maintenance
- Makes programs modular

Methods, Operator Overloading, and Exception Handling in Python

Python provides powerful Object-Oriented Programming (OOP) features such as methods and operator overloading, along with robust error management using exception handling. These concepts help in building flexible, reusable, and error-free programs.

Methods in Python

A method is a function defined inside a class that operates on objects of that class. Methods define the behavior of objects.

Types of methods include instance methods, class methods, and static methods.

Instance Methods

Instance methods use `self` and work with object data.

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

    def display(self):
        print("Name:", self.name)

s1 = Student("Rahul")
s1.display()
```

Output:

```
Name: Rahul
```

Class Methods

Class methods work on class variables and use `cls`.

```
class Student:
    school = "ABC School"

    @classmethod
    def show_school(cls):
        print(cls.school)

Student.show_school()
```

Output:

```
ABC School
```

Static Methods

Static methods do not use `self` or `cls`.

```
class Math:
    @staticmethod
    def add(a, b):
        return a + b

print(Math.add(10, 20))
```

Output:

```
30
```

Operator Overloading in Python

Operator overloading allows operators to work with user-defined objects using magic methods.

Example

```
class Number:
    def __init__(self, value):
        self.value = value

    def __add__(self, other):
        return self.value + other.value

n1 = Number(10)
n2 = Number(20)

print(n1 + n2)
```

Output:

```
30
```

Comparison Operator Example

```
class Number:
    def __init__(self, value):
        self.value = value

    def __gt__(self, other):
        return self.value > other.value

a = Number(10)
b = Number(5)

print(a > b)
```

Output:

```
True
```

Common Magic Methods

Operator	Method
+	__add__()
-	__sub__()
*	__mul__()
/	__truediv__()
==	__eq__()
>	__gt__()

Exception Handling in Python

Exception handling is used to handle runtime errors and prevent program crashes.

try and except

```
try:
    x = 10 / 0
except ZeroDivisionError:
    print("Cannot divide by zero")
```

Output:

```
Cannot divide by zero
```

try, except, else

```
try:
    x = 10 / 2
except ZeroDivisionError:
    print("Error")
else:
    print("No error")
```

Output:

```
No error
```

finally Block

```
try:
    f = open("file.txt")
except FileNotFoundError:
    print("File not found")
finally:
    print("Execution completed")
```

Output:

```
File not found
Execution completed
```

Raising Exceptions

```
age = -5

if age < 0:
    raise ValueError("Age cannot be negative")
```

Advantages of Exception Handling

- Prevents program crashes
- Improves reliability
- Helps in debugging
- Provides better user experience
- Handles unexpected errors

File Operations in Python

File handling in Python is used to store data permanently in a file. Unlike variables, which lose data when the program ends, files keep data saved even after execution. Python provides built-in functions to perform file operations such as reading, writing, and appending files.

A file must be opened before performing any operation using the `open()` function.

```
file = open("data.txt", "mode")
```

Common file modes are:

- r → read mode
- w → write mode
- a → append mode

Reading Files

Reading means accessing data stored in a file.

Reading entire file

```
file = open("data.txt", "r")

content = file.read()
print(content)

file.close()
```

If the file contains text, it will be displayed completely.

Reading one line

```
file = open("data.txt", "r")

print(file.readline())

file.close()
```

Reading all lines

```
file = open("data.txt", "r")

print(file.readlines())

file.close()
```

Reading files is used in data analysis, log processing, and configuration reading.

Writing Files

Writing means storing new data into a file. In write mode (`w`), existing data is overwritten.

Example

```
file = open("data.txt", "w")

file.write("Hello Python")

file.close()
```

If the file already has content, it will be replaced.

Writing is used for saving reports, output data, and user information.

Appending Files

Appending means adding data to an existing file without deleting previous content.

Example

```
file = open("data.txt", "a")

file.write("\nWelcome to Python")

file.close()
```

Appending is used in logs, records, and continuous data storage.

Using with Statement

The `with` statement automatically closes the file.

```
with open("data.txt", "r") as file:
    print(file.read())
```

This is the safest way to handle files.

Working with CSV Files

CSV (Comma Separated Values) files are used to store tabular data such as spreadsheets.

Python provides a built-in `csv` module.

Reading CSV file

```
import csv

with open("students.csv", "r") as file:
    reader = csv.reader(file)

    for row in reader:
        print(row)
```

Output:
Each row is displayed as a list.

Writing CSV file

```
import csv

with open("students.csv", "w", newline="") as file:
    writer = csv.writer(file)

    writer.writerow(["Name", "Age"])
    writer.writerow(["Rahul", 20])
```

CSV files are used in data analysis, Excel sheets, and databases.

Working with JSON Files

JSON (JavaScript Object Notation) is a lightweight format used to store and exchange data.

Python provides the `json` module.

Writing JSON file

```
import json

data = {
    "name": "Rahul",
    "age": 20
}

with open("data.json", "w") as file:
    json.dump(data, file)
```

Reading JSON file

```
import json

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

print(data)
```

Output:

```
{'name': 'Rahul', 'age': 20}
```

Python Modules: Built-In Modules and Custom Modules

A module in Python is a file that contains Python code such as variables, functions, and classes. Modules help in organizing code into reusable parts, making programs easier to manage, maintain, and understand. Python provides many built-in modules and also allows users to create their own custom modules.

Built-In Modules in Python

Built-in modules are pre-installed with Python and can be used directly without installation. Some commonly used built-in modules are `math`, `os`, and `datetime`.

Math Module

The `math` module provides mathematical functions for calculations such as square root, factorial, power, and logarithms.

Example

```
import math

print(math.sqrt(25))
print(math.factorial(5))
```

Output:

```
5.0
120
```

Uses of math module

- Scientific calculations
- Engineering problems
- Mathematical operations

OS Module

The `os` module is used to interact with the operating system. It allows file and directory operations.

Example

```
import os

print(os.getcwd())
```

This prints the current working directory.

Common OS operations

```
import os

os.mkdir("NewFolder")   # Create folder
print(os.listdir())     # List files and folders
```

Uses of OS module

- File management
- Directory handling
- System-level operations

Datetime Module

The `datetime` module is used to work with dates and times.

Example

```
from datetime import datetime

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

Output:

```
2026-06-25 10:30:00
```

Uses of datetime module

- Time and date tracking
- Scheduling tasks
- Logging events

Importing Modules

Modules can be imported in different ways.

Import entire module

```
import math

print(math.sqrt(16))
```

Import specific function

```
from math import sqrt

print(sqrt(16))
```

Import with alias

```
import math as m

print(m.sqrt(16))
```

Creating Custom Modules

A custom module is a user-defined Python file that contains reusable functions, variables, or classes.

Step 1: Create a module file

Create a file named `my_module.py`

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

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

Step 2: Use the module in another file

```
import my_module

print(my_module.add(10, 20))
print(my_module.multiply(5, 4))
```

Output:

```
30
20
```

Advantages of Custom Modules

- Code reusability
- Better organization
- Easy maintenance
- Avoid repetition
- Improves readability

Why Modules are Important

Modules help in dividing large programs into smaller parts. This makes development faster and debugging easier. Built-in modules provide ready-made functionality, while custom modules allow programmers to create their own reusable code.

Mini-Project: Developing a Python Script for a Real-World Problem

A mini-project in Python is a small real-world application that combines different programming concepts such as file handling, loops, functions, conditionals, and modules to solve a practical problem. It helps in understanding how Python is used beyond theory in real-life situations like automation, data management, and task handling.

Problem Statement

Many users store large numbers of files in a folder (downloads, documents, images, etc.), and it becomes difficult to manage them manually. The goal is to create a Python script that automatically organizes files into different folders based on their file types.

Objective of the Project

The main objective of this project is to:

- Automatically organize files in a directory
- Group files based on extensions (like .jpg, .pdf, .txt)
- Reduce manual effort
- Improve file management efficiency

Concepts Used

This project uses:

- os module (for file handling and directory operations)
- shutil module (for moving files)
- loops (for iterating files)
- conditionals (for checking file types)

Python Script for File Organizer

```
import os
import shutil

folder_path = "Downloads"

files = os.listdir(folder_path)

for file in files:
    file_path = os.path.join(folder_path, file)

    if os.path.isfile(file_path):

        extension = file.split(".")[-1]

        new_folder = os.path.join(folder_path, extension)
```

```
        if not os.path.exists(new_folder):
            os.mkdir(new_folder)

    shutil.move(file_path, os.path.join(new_folder, file))
```

Explanation of the Script

1. The program imports `os` and `shutil` modules.
2. It reads all files from the given folder using `os.listdir()`.
3. It checks whether each item is a file using `os.path.isfile()`.
4. It extracts the file extension (like jpg, pdf, txt).
5. It creates a folder for that extension if it does not exist.
6. It moves the file into the respective folder using `shutil.move()`.

Example Working

Before running the script:

Downloads folder contains:

- photo.jpg
- report.pdf
- notes.txt

After running the script:

Downloads folder becomes:

- jpg/photo.jpg
- pdf/report.pdf
- txt/notes.txt

Real-World Applications

This project can be used in:

- File management systems
- Desktop automation
- Data organization tools
- Cloud storage organization
- Office file sorting systems

Advantages of the Project

- Saves time
- Reduces manual work
- Organizes files efficiently
- Easy to expand and modify
- Demonstrates real automation

Possible Enhancements

- Sorting by file size
- Sorting by date
- Adding GUI using Tkinter
- Creating logs of moved files
- Handling duplicate files

Python Modules: Built-In Modules and Custom Modules

A module in Python is a file containing Python code such as functions, variables, and classes. Modules are used to organize programs into smaller, reusable parts. They help improve readability, reduce repetition, and make large programs easier to manage.

Python provides many built-in modules and also allows programmers to create their own custom modules.

Built-In Modules in Python

Built-in modules are pre-installed with Python and can be used directly without any installation. Some commonly used built-in modules are `math`, `os`, and `datetime`.

Math Module

The `math` module provides mathematical functions for performing calculations like square root, factorial, power, and trigonometric operations.

Example

```
import math

print(math.sqrt(25))
print(math.factorial(5))
```

Output:

```
5.0
120
```

Uses of math module

- Scientific calculations
- Engineering computations
- Mathematical problem solving

OS Module

The `os` module is used to interact with the operating system. It helps in file handling, directory management, and system operations.

Example

```
import os

print(os.getcwd())
```

Common OS operations

```
import os

os.mkdir("NewFolder")
print(os.listdir())
```

Uses of OS module

- File and folder management
- System-level operations
- Automation tasks

Datetime Module

The `datetime` module is used to work with date and time in Python.

Example

```
from datetime import datetime

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

Output:

```
2026-06-25 10:30:00
```

Uses of datetime module

- Date and time tracking
- Scheduling tasks
- Event logging
- Time-based calculations

Importing Modules in Python

Modules can be imported in different ways:

1. Import entire module

```
import math
print(math.sqrt(16))
```

2. Import specific function

```
from math import sqrt
print(sqrt(16))
```

3. Import with alias

```
import math as m
print(m.sqrt(16))
```

Creating Custom Modules

A custom module is a user-defined Python file that contains reusable code such as functions or variables.

Step 1: Create a module file

Create a file named `my_module.py`

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

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

Step 2: Use the module in another program

```
import my_module

print(my_module.add(10, 20))
print(my_module.subtract(30, 10))
```

Output:

```
30
20
```

Advantages of Custom Modules

- Code reusability
- Better program organization
- Easy maintenance
- Reduces repetition
- Improves readability

Importance of Modules in Python

Modules make programming easier by dividing large programs into smaller parts. Built-in modules provide ready-made functionality, while custom modules allow programmers to create reusable code for specific tasks.

Basic Programs in Python

Python is a simple and beginner-friendly programming language. Basic programs help understand syntax, variables, input/output, and logic building.

Below are some commonly used basic Python programs.

1. Hello World Program

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

Output:

```
Hello, World!
```

2. Add Two Numbers

```
a = 10
b = 20
sum = a + b

print("Sum =", sum)
```

Output:

```
Sum = 30
```

3. User Input Program

```
name = input("Enter your name: ")

print("Hello", name)
```

4. Even or Odd Number

```
num = int(input("Enter a number: "))

if num % 2 == 0:
    print("Even Number")
else:
    print("Odd Number")
```

5. Find Largest Number

```
a = 10
b = 20

if a > b:
    print("A is largest")
else:
    print("B is largest")
```

6. Factorial of a Number

```
num = 5
fact = 1
```

```
for i in range(1, num + 1):
    fact = fact * i

print("Factorial =", fact)
```

Output:

```
Factorial = 120
```

7. Simple Calculator

```
a = int(input("Enter first number: "))
b = int(input("Enter second number: "))

print("Addition:", a + b)
print("Subtraction:", a - b)
print("Multiplication:", a * b)
print("Division:", a / b)
```

8. Print Numbers from 1 to 10

```
for i in range(1, 11):
    print(i)
```

9. Reverse a Number

```
num = 1234
rev = 0

while num > 0:
    digit = num % 10
    rev = rev * 10 + digit
    num = num // 10

print("Reversed Number =", rev)
```

10. Check Prime Number

```
num = 7
flag = 0

for i in range(2, num):
    if num % i == 0:
        flag = 1
        break

if flag == 0:
    print("Prime Number")
else:
    print("Not Prime Number")
```