UNIT-IV:

Files: Types of Files, Creating and Reading Text Data, File Methods to Read and Write Data, Reading and Writing Binary Files, Pickle Module, Reading and Writing CSV Files, Python os and os.path Modules.

Object-Oriented Programming: Classes and Objects, Creating Classes in Python, Creating Objects in Python, Constructor Method, Classes with Multiple Objects, Class Attributes Vs Data Attributes, Encapsulation, Inheritance, Polymorphism.

PART A: FILE HANDLING IN PYTHON

1. What is a File?

A **file** is a collection of data stored permanently on a computer.

Python provides file-handling functions to:

- Create files
- Open files
- Read data
- Write data
- Append data
- Close files

General Syntax

file_object = open("filename", "mode")

Example:

file = open("student.txt", "r")

Here:

- student.txt → file name
- "r" → read mode
- file → file object

---

2. Types of Files

There are mainly two types of files:

1. Text Files

Store data as readable characters.

Examples:

.txt

.csv

.py

.html

Example content:

Ravi

20

CSE

2. Binary Files

Store data in binary form (bytes).

Examples:

.jpg

.png

.mp3

.mp4

.pdf

.dat

Binary data is generally not meant to be read directly as ordinary text.

---

3. File Opening Modes

Python provides several file modes.

Mode	Meaning
r	Read
w	Write
a	Append

Mode	Meaning
x	Create a new file
r+	Read and write
w+	Write and read
a+	Append and read
rb	Read binary
wb	Write binary

---

4. Creating a Text File

We can create a file using w mode.

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


file.write("Ravi\n")
file.write("Computer Science\n")
file.write("Python Programming")


file.close()
```

This creates:

student.txt

with:

Ravi

Computer Science

Python Programming

Important

If the file already exists, w mode **overwrites its existing contents**.

---

5. Reading Text Data

Using read()

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

data = file.read()

print(data)

file.close()
```

Output

```
Ravi

Computer Science

Python Programming
```

read() reads the entire file.

---

6. Reading a Specific Number of Characters

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


data = file.read(10)


print(data)


file.close()
```

read(10) reads approximately the first **10 characters**.

---

7. Reading One Line

The readline() method reads one line at a time.

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


line = file.readline()
```

```
print(line)

file.close()
```

---

8. Reading All Lines

The readlines() method returns all lines as a list.

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

lines = file.readlines()

print(lines)

file.close()
```

Example output:

```
['Ravi\n', 'Computer Science\n', 'Python Programming']
```

---

9. Writing Data to a File

The write() method writes a string into a file.

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

file.write("Welcome to Python\n")
file.write("File Handling")

file.close()
```

---

10. Writing Multiple Lines

Use writelines().

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

lines = [
    "Python\n",
    "Java\n",
    "C++\n"
]

file.writelines(lines)

file.close()
```

---

11. Append Data to a File

The a mode adds data to the end of an existing file.

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

file.write("\nArtificial Intelligence")

file.close()
```

Existing data is not deleted.

---

12. Using with open()

A better approach is to use the with statement.

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

The file is automatically closed after the with block.

Recommended approach

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

---

13. File Methods

Important file methods:

Method	Purpose
read()	Reads entire file
read(n)	Reads n characters
readline()	Reads one line
readlines()	Reads all lines
write()	Writes data
writelines()	Writes multiple lines
seek()	Changes file position
tell()	Returns current position
close()	Closes file

---

14. tell() Method

tell() returns the current position of the file pointer.

```
with open("data.txt", "r") as file:
    print(file.tell())
    print(file.read(5))
    print(file.tell())
```

---

15. seek() Method

seek() moves the file pointer to a particular position.

```
with open("data.txt", "r") as file:
    file.seek(0)
```

```
print(file.read())
```

---

16. Reading and Writing Binary Files

Binary files are handled using b with the mode.

Read Binary

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


print(data[:20])
```

Here:

"rb"

means:

Read Binary

Write Binary

```
with open("copy.jpg", "wb") as file:
    file.write(data)
```

wb means:

Write Binary

---

17. Pickle Module

The **pickle module** is used to store Python objects in a binary file and retrieve them later.

It is useful for storing:

- Lists
- Dictionaries
- Tuples
- Objects
- Other Python data structures

---

pickle.dump()

Used to write an object to a file.

```
import pickle

student = {
    "name": "Ravi",
    "age": 20,
    "branch": "CSE"
}

with open("student.dat", "wb") as file:
    pickle.dump(student, file)
```

---

pickle.load()

Used to read the object.

```
import pickle

with open("student.dat", "rb") as file:
    student = pickle.load(file)

print(student)
```

Output:

```
{'name': 'Ravi', 'age': 20, 'branch': 'CSE'}
```

Important

Never unpickle data from an untrusted source, because loading a malicious pickle can execute code.

---

18. CSV Files

CSV stands for:

Comma-Separated Values

Example:

Name,Age,Branch

Ravi,20,CSE

Sita,21,AI

Python provides the csv module for working with CSV files.

---

19. Writing a CSV File

```
import csv


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


    writer.writerow(["Name", "Age", "Branch"])
    writer.writerow(["Ravi", 20, "CSE"])
    writer.writerow(["Sita", 21, "AI"])
```

---

20. 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', 'Branch']

['Ravi', '20', 'CSE']

['Sita', '21', 'AI']

---

21. os Module

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

```
import os
```

Get Current Directory

```
import os


print(os.getcwd())
```

Create Directory

```
import os


os.mkdir("PythonFiles")
```

List Files and Directories

```
import os


print(os.listdir())
```

Rename a File

```
import os


os.rename("old.txt", "new.txt")
```

Delete a File

```
import os


os.remove("data.txt")
```

---

22. os.path Module

os.path provides functions for working with file and directory paths.

Check Whether File Exists

```
import os


print(os.path.exists("student.txt"))
```

Output:

True

or

False

Check Whether It Is a File

```
print(os.path.isfile("student.txt"))
```

Check Whether It Is a Directory

```
print(os.path.isdir("PythonFiles"))
```

Get File Size

```
print(os.path.getsize("student.txt"))
```

PART B: OBJECT-ORIENTED PROGRAMMING

23. What is Object-Oriented Programming?

Object-Oriented Programming (OOP) is a programming approach in which programs are designed using **classes and objects**.

Important OOP concepts include:

1. Class
2. Object
3. Constructor
4. Encapsulation
5. Inheritance
6. Polymorphism

---

24. Classes and Objects

Class

A **class** is a blueprint or template for creating objects.

For example:

```
class Student:
    pass
```

Here Student is a class.

Object

An **object** is an instance of a class.

```
student1 = Student()
```

Here:

- Student → class
- student1 → object

---

25. Creating a Class

```
class Student:


    def display(self):
        print("This is a student")
```

---

26. Creating an Object

```
class Student:


   def display(self):
      print("This is a student Tej")
         print("This is PYTHON  class")
         print("This is a AI&DS Dept")
student1 = Student()


student1.display()
```

Output

This is a student

---

27. What is self?

self refers to the **current object**.

Example:

```
class Student:


   def display(self):
      print("Student object")



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

Python internally passes s1 as the self parameter.

---

28. Constructor Method

A constructor is a special method that is automatically called when an object is created.

In Python, the constructor is:

__init__()

Example:

```
class Student:

    def __init__(self):
        print("Constructor called")



s1 = Student()
```

Output:

Constructor called

---

29. Constructor with Data

```
class Student:

    def __init__(self, name, age):
        self.name = name
        self.age = age

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

s1 = Student("Ravi", 20)
s1.display()
```

Output:

Name: Ravi

Age: 20

---

30. Classes with Multiple Objects

A class can be used to create many objects.

```
class Student:

    def __init__(self, name, age):
        self.name = name
        self.age = age

    def display(self):
        print(self.name, self.age)



s1 = Student("Ravi", 20)
s2 = Student("Sita", 21)
s3 = Student("Kiran", 19)

s1.display()
s2.display()
s3.display()
```

Output:

Ravi 20

Sita 21

Kiran 19

Each object has its own data.

---

31. Class Attributes vs Data Attributes

This is an important B.Tech examination topic.

Class Attribute

A class attribute is shared by all objects.

```
class Student:
    college = "ABC Engineering College"
    def __init__(self, name):
        self.name = name
```

Here:

college

is a **class attribute**.

Data/Instance Attribute

An instance attribute belongs to a particular object.

self.name = name

name is an **instance attribute**.

Example

```
class Student:
    college = "ABC Engineering College"
    def __init__(self, name):
        self.name = name
s1 = Student("Ravi")
s2 = Student("Sita")
print(s1.name)
print(s2.name)
print(s1.college)
print(s2.college)
```

Output:

Ravi

Sita

ABC Engineering College

ABC Engineering College

Difference

Class Attribute	**Instance/Data Attribute**
Shared by objects	Different for each object
Defined inside class	Usually defined using self
One common value	Object-specific value

---

32. Encapsulation

Encapsulation means combining data and methods inside a class and controlling access to the internal data.

Python commonly uses:

- Public members
- Protected convention: _variable
- Private members: __variable

Example of Encapsulation

```
class BankAccount:

    def __init__(self, balance):
        self.__balance = balance

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

```
    def get_balance(self):
        return self.__balance


account = BankAccount(1000)

account.deposit(500)

print(account.get_balance())
```

Output:

1500

Here:

self.__balance

is a **private attribute**.

It is intended to be accessed through methods such as get_balance() rather than directly.

---

33. Inheritance

Inheritance allows one class to acquire properties and methods from another class.

The existing class is called the:

Parent/Base class

The new class is called the:

Child/Derived class

Example

```
class Animal:

    def eat(self):
        print("Animal eats")
```

```
class Dog(Animal):

    def bark(self):
        print("Dog barks")


d = Dog()

d.eat()
d.bark()
```

Output:

Animal eats

Dog barks

Dog inherits the eat() method from Animal.

---

34. Types of Inheritance

Common types include:

1. Single Inheritance

A

↓

B

2. Multilevel Inheritance

A

↓

B

↓

C

3. Multiple Inheritance

A B

\ /

C

4. Hierarchical Inheritance

A

/ \

B C

5. Hybrid Inheritance

Combination of multiple inheritance types.

---

35. Polymorphism

Polymorphism means **one interface/name with different behavior**.

For example, different classes can have a method with the same name but perform different operations.

```
class Dog:


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



class Cat:


    def sound(self):
        print("Cat meows")
```

```
d = Dog()
c = Cat()

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

Output:

```
Dog barks
Cat meows
```

The method name is the same:

```
sound()
```

but its behavior is different.

---

36. Polymorphism Through Inheritance

```
class Animal:

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



class Dog(Animal):

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



class Cat(Animal):

    def sound(self):
```

```
        print("Cat meows")



animals = [Dog(), Cat()]


for animal in animals:
    animal.sound()
```

Output:

Dog barks

Cat meows

The same statement:

animal.sound()

produces different behavior depending on the object.

Quick Revision Tables

File Handling

Concept	**Python**
Open file	open()
Read	read()
Read line	readline()
Read all lines	readlines()
Write	write()
Write multiple lines	writelines()
Current position	tell()
Move position	seek()
Close	close()
Binary file	rb, wb

Concept	Python
Python objects	pickle
CSV	csv
OS operations	os
Path operations	os.path

OOP

Concept	Meaning
Class	Blueprint/template
Object	Instance of class
__init__()	Constructor
self	Current object
Class attribute	Shared by objects
Instance attribute	Object-specific
Encapsulation	Bundling/protecting data and methods
Inheritance	Reusing parent class features
Polymorphism	Same interface, different behavior

One Complete OOP Example

This combines **class, object, constructor, class attribute, instance attributes, and method**:

```
class Student:

    # Class attribute
    college = "ABC Engineering College"

    # Constructor
    def __init__(self, name, age, branch):
```

```
        # Instance attributes
        self.name = name
        self.age = age
        self.branch = branch

    # Method
    def display(self):
        print("Name    :", self.name)
        print("Age     :", self.age)
        print("Branch  :", self.branch)
        print("College :", self.college)



# Creating objects
s1 = Student("Ravi", 20, "CSE")
s2 = Student("Sita", 21, "AI")

# Calling methods
s1.display()
print()
s2.display()
```

Output

```
Name    : Ravi
Age     : 20
Branch  : CSE
College : ABC Engineering College

Name    : Sita
```

Age : 21

Branch : AI

College : ABC Engineering College

Sample Experiments:

1. Write a program to sort words in a file and put them in another file. The output file should have only lower-case words, so any upper-case words from source must be lowered.

2. Python program to print each line of a file in reverse order.

3. Python program to compute the number of characters, words and lines in a file.

4. Write a program to create, display, append, insert and reverse the order of the items in the array.

5. Write a program to add, transpose and multiply two matrices.

6. Write a Python program to create a class that represents a shape. Include methods to calculate its area and perimeter. Implement subclasses for different shapes like circle, triangle, and square.

1. Sort Words in a File and Store Them in Another File

Requirements

- Read words from input.txt
- Convert all words to lowercase
- Sort them alphabetically
- Store the result in output.txt

Program

```
# Read words from input file

with open("input.txt", "r") as file:
    data = file.read()

# Convert to lowercase and split into words
words = data.lower().split()

# Sort the words
words.sort()

# Write sorted words into another file
with open("output.txt", "w") as file:
    for word in words:
        file.write(word + "\n")

print("Words sorted and written to output.txt")
```

Example input.txt

Python Java C Programming

Apple banana Orange

output.txt

apple

banana

c

java

orange

programming

python

Important concepts

data.lower()

Converts uppercase letters to lowercase.

words.sort()

Sorts the list alphabetically.

---

2. Print Each Line of a File in Reverse Order

Here, **the characters in each line are reversed**.

Program

```
with open("input.txt", "r") as file:


    for line in file:
        line = line.rstrip("\n")
        print(line[::-1])
```

Example input.txt

Python

Computer

Programming

Output

nohtyP

retupmoC

gnimmargorP

Explanation

line[::-1]

reverses the string.

For example:

"Python"[::-1]

gives:

nohtyP

---

3. Count Characters, Words and Lines in a File

Program

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

# Number of characters
characters = len(data)

# Number of words
words = len(data.split())

# Number of lines
lines = len(data.splitlines())

print("Number of characters:", characters)
print("Number of words:", words)
print("Number of lines:", lines)
```

Example input.txt

Python is easy

Python is powerful

Output

Number of characters: 34

Number of words: 6

Number of lines: 2

Note: Character count includes spaces and newline characters because it counts the raw file contents.

---

4. Create, Display, Append, Insert and Reverse an Array

In Python, a **list** is commonly used for this type of exercise.

Program

```
# Create an array/list
arr = [10, 20, 30, 40]

print("Original array:", arr)

# Append an item
arr.append(50)
print("After append:", arr)

# Insert an item at index 2
arr.insert(2, 25)
print("After insertion:", arr)

# Reverse the array
arr.reverse()
```

```
print("After reverse:", arr)
```

Output

Original array: [10, 20, 30, 40]

After append: [10, 20, 30, 40, 50]

After insertion: [10, 20, 25, 30, 40, 50]

After reverse: [50, 40, 30, 25, 20, 10]

Important methods

Method	Purpose
append()	Adds item at the end
insert()	Adds item at a specified position
reverse()	Reverses the list

---

5. Add, Transpose and Multiply Two Matrices

We will use nested lists to represent matrices.

Suppose:

A = 1 2

3 4

B = 5 6

7 8

Complete Program

```
# Define two matrices


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

```
B = [
    [5, 6],
    [7, 8]
]

# Matrix Addition

addition = [
    [A[i][j] + B[i][j] for j in range(2)]
    for i in range(2)
]

# Transpose of A

transpose = [
    [A[j][i] for j in range(2)]
    for i in range(2)
]

# Matrix Multiplication

multiplication = [
    [
        sum(A[i][k] * B[k][j] for k in range(2))
        for j in range(2)
    ]
    for i in range(2)
```

```
]

# Display results

print("Matrix A:")
for row in A:
    print(row)

print("\nMatrix B:")
for row in B:
    print(row)

print("\nAddition:")
for row in addition:
    print(row)

print("\nTranspose of A:")
for row in transpose:
    print(row)

print("\nMultiplication:")
for row in multiplication:
    print(row)
```

Output

```
Matrix A:
[1, 2]
[3, 4]
```

Matrix B:

[5, 6]

[7, 8]

Addition:

[6, 8]

[10, 12]

Transpose of A:

[1, 3]

[2, 4]

Multiplication:

[19, 22]

[43, 50]

Matrix addition

1+5 = 6

2+6 = 8

3+7 = 10

4+8 = 12

Therefore:

6 8

10 12

Matrix multiplication

First element:

(1×5) + (2×7)

= 5 + 14

= 19

Second element:

(1×6) + (2×8)

= 6 + 16

= 22

Therefore the result is:

19 22

43 50

---

6. Shape Class with Circle, Triangle and Square

This experiment demonstrates several important **OOP concepts**:

- Class
- Objects
- Constructor
- Inheritance
- Method overriding
- Polymorphism

We'll create a base class called Shape.

Then create:

- Circle
- Triangle
- Square

as subclasses.

Program

```
import math



# Base class
```

```
class Shape:

    def area(self):
        pass

    def perimeter(self):
        pass



# Circle class
class Circle(Shape):

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

    def area(self):
        return math.pi * self.radius * self.radius

    def perimeter(self):
        return 2 * math.pi * self.radius



# Square class
class Square(Shape):

    def __init__(self, side):
        self.side = side
```

```
    def area(self):
        return self.side * self.side

    def perimeter(self):
        return 4 * self.side


# Triangle class
class Triangle(Shape):

    def __init__(self, a, b, c):
        self.a = a
        self.b = b
        self.c = c

    def perimeter(self):
        return self.a + self.b + self.c

    def area(self):
        s = self.perimeter() / 2
        return math.sqrt(
            s * (s - self.a) *
            (s - self.b) *
            (s - self.c)
        )


# Create objects
```

```
circle = Circle(5)
square = Square(4)
triangle = Triangle(3, 4, 5)


# Display results
print("Circle")
print("Area =", circle.area())
print("Perimeter =", circle.perimeter())

print("\nSquare")
print("Area =", square.area())
print("Perimeter =", square.perimeter())

print("\nTriangle")
print("Area =", triangle.area())
print("Perimeter =", triangle.perimeter())
```

Output

Approximately:

Circle

Area = 78.53981633974483

Perimeter = 31.41592653589793

Square

Area = 16

Perimeter = 16

Triangle

Area = 6.0

Perimeter = 12

Understanding the OOP Program

1. Base Class

class Shape:

This is the parent/base class.

It provides the general structure for shapes.

---

2. Inheritance

class Circle(Shape):

means Circle inherits from Shape.

Similarly:

class Square(Shape):

and:

class Triangle(Shape):

inherit from Shape.

---

3. Constructor

For example:

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

The constructor runs automatically when we create:

circle = Circle(5)

So:

radius = 5

is stored in the object.

---

4. Method Overriding

The parent class has:

```
def area(self):
    pass
```

Each child class defines its own area():

Circle → area()

Square → area()

Triangle → area()

Each one calculates area differently.

---

5. Polymorphism

We can use the same method name:

shape.area()

for different types of shapes, and each object performs its own calculation.

For example:

circle.area()

square.area()

triangle.area()

The method name is the same, but the implementation is different.

Quick Lab-Exam Revision

Experiment	**Main Concepts**
1. Sort words in file	File handling, lower(), sort()
2. Reverse each line	File handling, slicing
3. Count characters/words/lines	read(), split(), splitlines()
4. Array operations	append(), insert(), reverse()
5. Matrix operations	Lists, nested loops/comprehensions

Experiment	Main Concepts
6. Shape hierarchy	Classes, objects, inheritance, polymorphism

Most important syntax to remember

File reading:

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

File writing:

```
with open("output.txt", "w") as file:
    file.write(data)
```

Append to list:

```
arr.append(value)
```

Insert into list:

```
arr.insert(index, value)
```

Reverse list:

```
arr.reverse()
```

Create a class:

```
class Student:
    pass
```

Create an object:

```
s1 = Student()
```

Inheritance:

```
class Child(Parent):
    pass
```

Constructor:

```
def __init__(self):
    pass
```

These six programs together provide good coverage of **File Handling, Lists, Matrix Operations, Classes, Inheritance, and Polymorphism**

Object-Oriented Programming (OOP) in Python is centered around organizing code into reusable blueprints called Classes to create specific instances called Objects. [1]

Python cleanly implements the **four core pillars of OOP**: **Inheritance**, **Polymorphism**, **Encapsulation**, and **Abstraction**. Here is a comprehensive guide to their technical implementation. [1]

---

1. The Foundation: Classes and Objects

A **Class** serves as the blueprint, while an **Object** is a concrete instance of that class. [1, 2]

- __init__: The constructor method used to initialize instance variables.
- self: Represents the specific instance of the object being modified or accessed. [1, 2, 3]

python

```
class Smartphone:
    # Class attribute (shared by all instances)
    platform = "Mobile"

    # Constructor method
    def __init__(self, brand, model):
        self.brand = brand      # Instance attribute
        self.model = model      # Instance attribute

    # Instance method
    def display_info(self):
        return f"{self.brand} {self.model} runs on {self.platform}"

# Object Instantiation
phone1 = Smartphone("Apple", "iPhone 15")
print(phone1.display_info())  # Output: Apple iPhone 15 runs on Mobile
```

---

2. Inheritance

Inheritance allows a **child class** to inherit attributes and methods from a **parent class**, promoting code reusability. [1, 2]

- Python supports single, multiple, multilevel, and hierarchical inheritance.
- Use super() to safely call the parent class's constructor or methods. [1, 2]

```
python

# Parent Class

class Vehicle:

    def __init__(self, brand):

        self.brand = brand


    def start_engine(self):

        return "Engine started."


# Child Class inheriting from Vehicle

class ElectricCar(Vehicle):

    def __init__(self, brand, battery_capacity):

        # Forward parent attributes using super()

        super().__init__(brand)

        self.battery_capacity = battery_capacity


    # Unique child method

    def charge(self):

        return f"Charging the {self.battery_capacity}kWh battery."


tesla = ElectricCar("Tesla", 75)

print(tesla.start_engine())  # Inherited method: Engine started.

print(tesla.charge())        # Child method: Charging the 75kWh battery.
```

---

3. Polymorphism

Polymorphism means "many forms". It allows different classes to have **methods with the same name** but unique execution logic. [1, 2]

- **Method Overriding:** A child class changes the behavior of a method inherited from a parent class. [1]

- **Method Overloading:** Python does not support compile-time method overloading natively, but achieves it via optional default arguments or argument lists (*args, **kwargs). [1]

python

```
class Dog:
    def make_sound(self):
        return "Woof!"

class Cat:
    def make_sound(self):
        return "Meow!"

# A unified interface handling different object types
def animal_sound_handler(animal_object):
    print(animal_object.make_sound())

dog = Dog()
cat = Cat()

animal_sound_handler(dog)  # Output: Woof!
animal_sound_handler(cat)  # Output: Meow!
```

---

4. Encapsulation

Encapsulation wraps data and the methods that handle it inside a single unit while **restricting direct access** to prevent accidental modifications. [1, 2]

- **Protected (_attribute):** A single underscore flags an attribute for internal use within the class or subclasses. It operates entirely on developer convention (the "consenting adults" rule). [1]
- **Private (__attribute):** A double underscore triggers Python's internal **name mangling**, rewriting the variable name to obscure direct external access. [1]

- Use @property decorators to safely access and validate hidden fields via getters and setters. [1, 2]

python

```
class BankAccount:
    def __init__(self, owner, balance):
        self.owner = owner
        self.__balance = balance  # Private attribute (Name mangled)

    # Getter: Expose private attribute cleanly
    @property
    def balance(self):
        return self.__balance

    # Setter: Modify private attribute with validation rules
    @balance.setter
    def balance(self, amount):
        if amount >= 0:
            self.__balance = amount
        else:
            raise ValueError("Balance cannot be negative.")

account = BankAccount("Alice", 1000)

# Direct access to private attribute fails
# print(account.__balance) -> Throws AttributeError

# Accessing and modifying through the encapsulated property layer
account.balance = 1500
```

```
print(account.balance)  # Output: 1500
```

---

5. Data Abstraction

Abstraction **hides complex implementation details** and exposes only essential interfaces. [1, 2]

- Python implements abstraction using the built-in abc (Abstract Base Class) module.
- An abstract class **cannot be instantiated** directly. It forces any child class to implement its designated @abstractmethod primitives. [1]

```
python

from abc import ABC, abstractmethod


# Abstract Base Class acting as a strict template

class PaymentGateway(ABC):


    @abstractmethod

    def process_payment(self, amount):

        pass


# Concrete implementation inheriting from the abstract blueprint

class PayPalPayment(PaymentGateway):


    def process_payment(self, amount):

        return f"Processing ${amount} securely via PayPal API."


# gateway = PaymentGateway() -> Throws TypeError: Can't instantiate abstract class


payment = PayPalPayment()

print(payment.process_payment(250)) # Output: Processing $250 securely via PayPal API.
```



YouTube·Patrick Loeber

20m

Object Oriented Programming (OOP) In Python



YouTube·Bro Code

7m

Python Object Oriented Programming Full Course 🐍

NareshIT

Python OOPs (Object-Oriented Programming) Concepts

Table_title: 1️⃣Key OOP Concepts in Python Table_content: | Concept | Description | | --- | --- | | Class | A blueprint for creat...

Programiz

Python Object Oriented Programming - Programiz

Encapsulation is one of the key features of object-oriented programming. Encapsulation refers to the bundling of attributes and me...

WsCube Tech

OOPs Concepts in Python: All Types With Examples

Table of Contents * Introduction * What Is Object-Oriented Programming in Python? * OOPs Concepts in Python * Python Class * Pytho...

Dev Genius

Mastering Advanced OOP Concepts in Python: Theory Behind OOP | by Raman Bazhanau | Dev Genius

OOP Pillars: Abstraction (focus on behavior via ABC), Inheritance ("is-a", C3 linearization MRO via `super()`), Encapsulation (pri...

edureka.co

Object Oriented Programming Python | Python OOP Concepts | Edureka

Core Concepts: Class (blueprint), Object (instance), Method, Inheritance, Polymorphism, Abstraction, Encapsulation. OOP vs Procedu...

Scribd

Python OOP Concepts Overview | PDF | Object Oriented Programming | Class (Computer Programming)

Core Pillars: Class (logical entity/collection of objects); Object (instance with state/behavior); Method (function associated wit...

YouTube·freeCodeCamp.org

25m

Python Object Oriented Programming (OOP)