UNIT-3:

Lecture-25

Python Programming: Custom Modules

Creating a Basic Custom Module

Objective: To understand the structure of a Python module and learn how to create and import it to reuse code.

Theory: A module is a .py file that contains Python code, such as functions, classes, and variables. Modules are a fundamental way to organize code, promote reusability, and make programs more manageable. To use a module's contents, you must import it into your main script.

Sub-Topics:

- Module Creation: A module is simply a Python file. You can create a file (e.g., greetings.py) and add functions to it.
- Standard Import: The import module_name statement makes the entire module available. To call a function from the module, you must use the syntax module_name.function_name().

Program:

- File 1: greetings.py

```
# greetings.py

def say_hello(name):

"""Prints a simple greeting.""" print(f"Hello, {name}!")

def say_goodbye(name):

"""Prints a farewell message.""" print(f"Goodbye, {name}!")
```

- File 2: main_script.py

```
# main_script.py

# Import the entire 'greetings' module.

import greetings

# Use functions from the module by referencing the module name. greetings.say_hello("Alice")

greetings.say_goodbye("Bob")Output:
```

```
Hello, Alice! Goodbye, Bob!
```

Different Ways to Import from a Module

Objective: To explore various import methods and understand their use cases.

Theory: Python offers multiple ways to import items from a module, each with its own advantages.

• import module_name as alias: Imports the module with a shorter name, which is useful for long module names.

• from module_name import item1, item2, ...: Imports specific functions or variables directly into the current namespace. You can use them without the module name prefix.

• from module_name import * (Wildcard Import): Imports all items from the module directly into the current namespace. Caution: This is generally discouraged as it can lead to naming conflicts and make the code's origin difficult to trace.

Program:

• File 1: calculations.py

```
# calculations.py def add(a, b):

return a + b

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

• File 2: import_methods.py

```
# import_methods.py

# Method 1: Import with an alias

import calculations as calc

print(f"Using alias: 5 * 3 = {calc.multiply(5, 3)}")# Method 2: Import specific functions

from calculations import add

print(f"Using specific import: 10 + 7 = {add(10, 7)}")

# Method 3: Wildcard import

from calculations import *

print(f"Using wildcard: 4 * 4 = {multiply(4, 4)}")
```

Output:

```
Using alias: 5 * 3 = 15

Using specific import: 10 + 7 = 17 Using wildcard: 4 * 4 = 16
```

Creating a Custom Module with a Class

Objective: To demonstrate that a module can contain classes, which can be instantiated and used in other scripts.

Theory: Modules are not limited to functions. They can also contain classes, allowing you to organize your object-oriented code into separate, logical files. You can then import the class just like you would a function.

Program:

• File 1: vehicle.py

```
# vehicle.py

class Vehicle:

"""A class to represent a vehicle.""" def __init__(self, make, model):

self.make = make self.model = model

def display_info(self):

return f"Vehicle: {self.make} {self.model}"•     File 2: car_shop.py

# car_shop.py

# Import the Vehicle class from the 'vehicle' module. from vehicle import Vehicle

# Create an object of the imported class. my_car = Vehicle("Honda", "Civic")

# Use the object's method. print(my_car.display_info())
```

Output:

```
Vehicle: Honda Civic
```

Lecture-26

Object-Oriented Concepts in Python

Classes and Objects

Objective: To understand the fundamental concepts of classes and objects in Python by creating a simple Car class and instantiating its objects.

Theory: A class is a blueprint or a template for creating objects. It defines a set of attributes (data) and methods (functions) that the objects created from it will have. An object is a specific instance of a class. When a class is defined, no memory is allocated until an object is created.

Program:

```
# A class is a blueprint for objects.

class Car:

# The __init__ method is a constructor, called when a new object is created. # It initializes the object's attributes.

def __init__(self, make, model, year):

self.make = make   # Attribute 1

self.model = model # Attribute 2

self.year = year   # Attribute 3

print(f"A new {self.make} {self.model} has been created.")

# A method is a function defined within a class.

def display_info(self):

print(f"Make: {self.make}")

print(f"Model: {self.model}")

print(f"Year: {self.year}")

# An object is an instance of a class.

# Here, we create two objects (instances) of the Car class. car1 = Car("Toyota", "Camry", 2022)

car2 = Car("Honda", "Civic", 2023)# We can access the object's methods and attributes using dot notation. print("\n--- Car 1 Information ---")

car1.display_info()

print("\n--- Car 2 Information ---") car2.display_info()
```

Output:

A new Toyota Camry has been created. A new Honda Civic has been created.

--- Car 1 Information --- Make: Toyota

Model: Camry

Year: 2022

--- Car 2 Information --- Make: Honda

Model: Civic Year: 2023

Inheritance

Objective: To demonstrate the principle of inheritance by creating a Dog class that inherits from a parent Animal class, thus reusing its methods and attributes.

Theory: Inheritance is a mechanism where a new class (child class) inherits the attributes and methods of an existing class (parent class). This promotes code reuse and creates a logical hierarchy. The super() function is often used to call the parent class's constructor (__init__) and other methods.

Types of Inheritance:

1. Single Inheritance: A derived class inherits from only one base class.

2. Multiple Inheritance: A derived class inherits from multiple base classes.3. Multilevel Inheritance: A derived class inherits from a class that itself is a derived class, forming a chain (e.g., Class A -> Class B -> Class C).

4. Hierarchical Inheritance: Multiple derived classes inherit from a single base class.

5. Hybrid Inheritance: A combination of two or more types of inheritance, such as combining multilevel and multiple inheritance.

Polymorphism

Objective: To understand polymorphism by creating different classes that share the same method name but have different implementations.

Theory: Polymorphism (meaning "many forms") allows objects of different classes to be treated as objects of a common base class. This is often achieved through method overriding, where a child class provides its own unique implementation for a method already defined in its parent class.

Types of Polymorphism

Polymorphism in Python refers to ability of the same method or operation to behave differently based on object or context. It mainly includes compile-time and runtime polymorphism.

Encapsulation

Objective: To implement encapsulation by using "private" attributes to restrict direct access to an object's data.

Theory: Encapsulation is the principle of bundling data (attributes) and the methods that operate on that data into a single unit (the class). It also involves data hiding, where the internal state of an object is protected from direct external access. In Python, this is a convention using leading underscores. A single underscore (_) indicates a protected member, and a double underscore (__)

"mangles" the name to make it harder to access from outside.

Program:

```
class BankAccount:

def __init__(self, initial_balance):

# A private attribute, indicated by the double underscore. self.__balance = initial_balance

# A public method to deposit money. def deposit(self, amount):

if amount > 0:

self.__balance += amountprint(f"Deposited {amount}. New balance: {self.__balance}")

else:

print("Deposit amount must be positive.")

# A public method to get the balance. This is the controlled way to access the data. def
get_balance(self):

return self.__balance

# Create a bank account object.

my_account = BankAccount(1000)

# We can access public methods to interact with the object.

my_account.deposit(500)

print(f"Current balance from get_balance(): {my_account.get_balance()}")

# This direct access attempt will fail (or raise an error) because `__balance` is "private". # The
interpreter "mangles" the name to `_BankAccount__balance`.

print("Trying to access private attribute directly...") try:

print(my_account.__balance) except AttributeError as e:

print(f"Error: {e}")
```

Output:

```
Deposited 500. New balance: 1500

Current balance from get_balance(): 1500

Trying to access private attribute directly...

Error: 'BankAccount' object has no attribute '__balance'Abstraction
```

Objective: To understand abstraction by creating an abstract base class that defines a common interface for its derived classes.

Theory: Abstraction is the process of hiding complex implementation details and showing only the essential features of an object. In Python, this is achieved using abstract base classes (ABC) from the abc module. An abstract class defines methods that must be implemented by any concrete child class. It cannot be instantiated on its own.

Program:

```
from abc import ABC, abstractmethod

# Abstract Base Class

# A class that inherits from ABC is an abstract class.

class Shape(ABC):

# This is an abstract method. Child classes MUST provide an implementation for it. @abstractmethod

def area(self): pass

# Concrete Class

class Circle(Shape):

def __init__(self, radius):

self.radius = radius

# Provides the concrete implementation for the `area` method. def area(self):

return 3.14159 * self.radius * self.radius

# Concrete Class

class Rectangle(Shape):

def __init__(self, width, height):self.width = width

self.height = height

# Provides the concrete implementation for the `area` method. def area(self):

return self.width * self.height

# Create objects of the concrete classes

circle_obj = Circle(7)

rectangle_obj = Rectangle(5, 8)

print(f"Area of the circle: {circle_obj.area()}") print(f"Area of the rectangle: {rectangle_obj.area()}")

# Attempting to create an object of the abstract class will fail. try:

abstract_shape = Shape() except TypeError as e:

print(f"\nError: {e}")
```

Output:

```
Area of the circle: 153.93791

Area of the rectangle: 40
```

Error: Can't instantiate abstract class Shape with abstract method area

Lecture-27

Custom Classes

Creating a Basic Custom Class

Objective: To understand how to define a custom class, including attributes and methods, to create objects.

Theory: A class is a blueprint for creating objects. It defines the properties (called attributes) and behaviors (called methods) that all objects of that type will have. A custom class allows you to create your own data types that are tailored to your program's needs.

The __init__ method is a special function called a constructor. It's automatically executed when a new object of the class is created. Its purpose is to initialize the object's attributes.

The self parameter in a method definition refers to the instance of the object itself. It allows you to access and modify the object's attributes from within its methods.

Program:

```
# A class named 'Book' to represent a book object.

class Book:

# The constructor method, used to initialize the object's attributes. def __init__(self, title, author, pages):

self.title = title

self.author = author

self.pages = pages

print(f"A new book '{self.title}' has been created.")

# A method to display information about the book. def display_info(self):

"""Prints the title, author, and number of pages.""" print(f"Title: {self.title}")

print(f"Author: {self.author}") print(f"Pages: {self.pages}")

# Creating two objects (instances) of the Book class.book1 = Book("The Hitchhiker's Guide to the Galaxy", "Douglas Adams", 192) book2 = Book("Dune", "Frank Herbert", 412)

# Calling the 'display_info' method for each object. book1.display_info()

print("-" * 20) book2.display_info()
```

Output:

```
A new book 'The Hitchhiker's Guide to the Galaxy' has been created. A new book 'Dune' has been created.

Title: The Hitchhiker's Guide to the Galaxy Author: Douglas Adams

Pages: 192

--------------------

Title: Dune

Author: Frank Herbert Pages: 412
```

Class and Instance Attributes

Objective: To differentiate between class-level attributes and instance-level attributes and understand their usage.

Theory: Instance attributes are unique to each object and are defined inside the constructor

(__init__) using the self keyword.

Class attributes are shared by all objects of the class. They are defined directly within the class body but outside of any methods. They are useful for storing constants or data that is common to all instances.

Program:

```
class Car:

# A class attribute shared by all Car objects.number_of_wheels = 4

def __init__(self, make, color):

# Instance attributes, unique to each object.

self.make = make

self.color = color

def display_info(self):

print(f"This is a {self.color} {self.make} with {self.number_of_wheels} wheels.")

# Create two different car objects.

car1 = Car("Toyota", "blue")

car2 = Car("Ford", "red")

# Accessing instance attributes.

print(f"Car 1 make: {car1.make}")

print(f"Car 2 make: {car2.make}")

# Accessing the class attribute via the object or the class itself. print(f"All cars have {car1.number_of_wheels} wheels.") print(f"Class attribute directly: {Car.number_of_wheels}")

# Changing the class attribute affects all instances. Car.number_of_wheels = 3
```

```
print("\nNumber of wheels has been changed.")

car1.display_info() car2.display_info()Output:
```

```
Car 1 make: Toyota

Car 2 make: Ford

All cars have 4 wheels.

Class attribute directly: 4

Number of wheels has been changed. This is a blue Toyota with 3 wheels. This is a red Ford with 3
wheels.
```

Lecture-28

Attributes and Transformers

Instance Attributes

Objective: To understand how to create and use instance attributes that are unique to each object of a class.

Theory: Attributes are variables that belong to a class. Instance attributes are specific to an instance (object) of a class. They store data that defines the unique state of each object. You define them within the __init__ constructor method using the self keyword. Each time a new object is created, these attributes are initialized with the values passed during instantiation.

Program:

```
# A class representing a `Dog`.

class Dog:

# The __init__ method is the constructor.

def __init__(self, name, age):

# Instance attributes, unique to each dog object.

self.name = name

self.age = age

# A method to display the dog's information.

def get_info(self):

return f"Name: {self.name}, Age: {self.age}"

# Create two separate dog objects with different attributes. dog1 = Dog("Buddy", 3)

dog2 = Dog("Lucy", 5)

# Access and print the instance attributes for each object. print(f"Dog 1: {dog1.get_info()}")

print(f"Dog 2: {dog2.get_info()}")Output:
```

```
Dog 1: Name: Buddy, Age: 3 Dog 2: Name: Lucy, Age: 5
```

Class Attributes

Objective: To understand how to define and use class attributes that are shared by all objects of a class.

Theory: Class attributes are attributes that are common to all instances of a class. They are defined directly inside the class body but outside of any methods. They are useful for storing data that does not change from one object to another, such as constants or default values. You can access a class attribute using either the class name (ClassName.attribute) or an instance of the class

(object.attribute).

Program:

```
class Planet:

# A class attribute representing the constant gravitational acceleration. gravitational_constant = 9.8
# meters/second^2

def __init__(self, name, mass):

self.name = name

self.mass = mass

# A method to display planet information.

def get_info(self):

return f"Planet: {self.name}, Mass: {self.mass} kg"

# Create two planet objects. earth = Planet("Earth", 5.97e24) mars = Planet("Mars", 6.39e23)

# Accessing the class attribute via the class name.print(f"Gravitational constant for all planets:
{Planet.gravitational_constant}")

# Accessing the class attribute via an object.

print(f"Gravitational constant for Earth: {earth.gravitational_constant}") print(f"Gravitational
constant for Mars: {mars.gravitational_constant}")
```

Output:

```
Gravitational constant for all planets: 9.8

Gravitational constant for Earth: 9.8

Gravitational constant for Mars: 9.8
```

Property Decorators as Transformers

Objective: To learn how to use Python's @property decorator to create "smart" attributes that act as both attributes and methods.

Theory: A property is a special kind of attribute that is managed by a method. It allows you to add getter, setter, and deleter functionality to a class attribute without changing the way you access it. This is a powerful form of data encapsulation, as it gives you control over how an attribute's value is retrieved and modified. The @property decorator turns a method into a getter, and

@<property_name>.setter turns another method into a setter.

Program:

```
class Circle:

def __init__(self, radius):

self._radius = radius  # A "protected" attribute with a leading underscore

# The @property decorator turns this method into a getter for 'radius'. @property

def radius(self):

print("Getting value...") return self._radius# The @radius.setter decorator defines the setter method for 'radius'. @radius.setter

def radius(self, value):

print("Setting value...")

if value < 0:

raise ValueError("Radius cannot be negative.") self._radius = value

# Create a Circle object.

my_circle = Circle(5)

# Access the property (calls the getter).

print(f"Initial radius: {my_circle.radius}")

# Set the property (calls the setter).

my_circle.radius = 10

print(f"New radius: {my_circle.radius}")

# This will raise an error because of the setter's validation. try:

my_circle.radius = -1 except ValueError as e:

print(f"Error: {e}")
```

Output:

```
Getting value...

Initial radius: 5

Setting value...

Getting value...New radius: 10

Setting value...

Error: Radius cannot be negative.
```

@classmethod and @staticmethod as Transformers

Objective: To understand the difference between instance methods, class methods, and static methods and how they transform a method's behavior.

Theory:

- An instance method takes self as its first parameter and operates on the instance's attributes.
- A class method takes cls (the class itself) as its first parameter. It is created using the

@classmethod decorator and is used for methods that operate on class attributes or need to create a new instance of the class in a specific way.

- A static method takes neither self nor cls as its first parameter. It is created using the

@staticmethod decorator and is essentially a regular function that is logically part of the class but does not depend on the class's state or the instance's state.

Program:

```
class MathOperations:

@staticmethod

def add(x, y):

# A static method. It doesn't need 'self' or 'cls'.

return x + y

@classmethod

def create_from_tuple(cls, numbers):

# A class method. It takes the class 'cls' as a parameter. # It can create and return a new instance.
return cls(numbers[0], numbers[1])

def __init__(self, num1, num2): self.num1 = num1self.num2 = num2

def multiply(self):

# An instance method. It operates on the instance's data. return self.num1 * self.num2

# Calling the static method via the class.

sum_result = MathOperations.add(5, 10)

print(f"Static method result: {sum_result}")

# Calling the class method to create a new instance.

# This is a common pattern for factory methods.

op_instance = MathOperations.create_from_tuple((8, 3))

# Calling the instance method.

product_result = op_instance.multiply() print(f"Instance method result: {product_result}")
```

Output:

```
Static method result: 15 Instance method result: 24
```

Lecture-29

Inheritance and Polymorphism

Inheritance

Objective: To demonstrate the principle of inheritance by creating a Dog class that inherits from a parent Animal class, thus reusing its methods and attributes.

Theory: Inheritance is a mechanism where a new class (child class) inherits the attributes and methods of an existing class (parent class). This promotes code reuse and creates a logical hierarchy. The super() function is often used to call the parent class's constructor (__init__) and other methods.

Types of Inheritance in Python:

- Single Inheritance: A child class inherits from a single parent class.
- Multiple Inheritance: A child class inherits from multiple parent classes.
- Multilevel Inheritance: A class inherits from a child class, which itself inherits from another parent class, forming a chain of inheritance.
- Hierarchical Inheritance: Multiple child classes inherit from a single parent class.
- Hybrid Inheritance: A combination of two or more types of inheritance.

1. Single Inheritance

In single inheritance, a child class inherits from just one parent class.

Example: This example shows a child class Employee inheriting a property from the parent class Person.

```
class Person:

def __init__(self, name):

self.name = name

class Employee(Person):  # Employee inherits from Person def show_role(self):

print(self.name, "is an employee") emp = Employee("Sarah")

print("Name:", emp.name) emp.show_role()Output

Name: Sarah

Sarah is an employee
```

2. Multiple Inheritance

In multiple inheritance, a child class can inherit from more than one parent class.

Example: This example demonstrates Employee inheriting properties from two parent classes: Person and Job.

```
class Person:

def __init__(self, name):

self.name = name

class Job:

def __init__(self, salary):

self.salary = salary

class Employee(Person, Job):  # Inherits from both Person and Job def __init__(self, name, salary):

Person.__init__(self, name) Job.__init__(self, salary)

def details(self):

print(self.name, "earns", self.salary)

emp = Employee("Jennifer", 50000) emp.details()Output

Jennifer earns 50000
```

3. Multilevel Inheritance

In multilevel inheritance, a class is derived from another derived class (like a chain). Example: This example shows Manager inheriting from Employee, which in turn inherits from Person.

```
class Person:

def __init__(self, name):

self.name = name

class Employee(Person):

def show_role(self):

print(self.name, "is an employee")

class Manager(Employee):  # Manager inherits from Employee def department(self, dept):

print(self.name, "manages", dept, "department")

mgr = Manager("Joy") mgr.show_role()

mgr.department("HR")
```

Output

```
Joy is an employee

Joy manages HR department4. Hierarchical Inheritance
```

In hierarchical inheritance, multiple child classes inherit from the same parent class. Example: This example demonstrates two child classes (Employee and Intern) inheriting from a single parent class Person.

```
class Person:

def __init__(self, name):

self.name = name

class Employee(Person):

def role(self):

print(self.name, "works as an employee")

class Intern(Person):

def role(self):

print(self.name, "is an intern")

emp = Employee("David") emp.role()

intern = Intern("Eva") intern.role()
```

Output

```
David works as an employee Eva is an intern
```

5. Hybrid Inheritance

Hybrid inheritance is a combination of more than one type of inheritance.

Example: This example demonstrates TeamLead inheriting from both Employee (which inherits Person) and Project, combining multiple inheritance types.class Person:

```
def __init__(self, name):

self.name = name

class Employee(Person):

def role(self):

print(self.name, "is an employee")

class Project:

def __init__(self, project_name):

self.project_name = project_name

class TeamLead(Employee, Project):  # Hybrid Inheritance def __init__(self, name, project_name):

Employee.__init__(self, name)

Project.__init__(self, project_name)

def details(self):
```

print(self.name, "leads project:", self.project_name)

lead = TeamLead("Sophia", "AI Development") lead.role()

lead.details()

Output

Sophia is an employee

Sophia leads project: AI DevelopmentPolymorphism

Objective: To understand polymorphism by creating different classes that share the same method name but have different implementations.

Theory: Polymorphism (meaning "many forms") allows objects of different classes to be treated as objects of a common base class. This is often achieved through method overriding, where a child class provides its own unique implementation for a method already defined in its parent class.

Key aspects of polymorphism in Python:

Method Overriding (Polymorphism with Inheritance):

o Subclasses can provide their own specific implementation of methods already defined in their superclass.

o When a method is called on an object, the version of the method in the object's actual class (or its closest ancestor) is executed.

Program:

```
class Animal:

def speak(self):

print("Animal speaks")

class Dog(Animal):

def speak(self):

print("Woof, woof!")

class Cat(Animal):

def speak(self):

print("Meow, meow!")

def make_sound(animal):

animal.speak()

dog = Dog()

cat = Cat()

make_sound(dog) # Output: Woof, woof! make_sound(cat) # Output: Meow, meow!Output:
```

```
Woof, woof! Meow, meow!
```

Polymorphism with Functions and Objects (Duck Typing):

o Python's dynamic typing system allows functions to operate on objects of different types as long as those objects provide the necessary methods or attributes. This is often referred to as "duck typing" – "If it walks like a duck and quacks like a duck, then it must be a duck."

o There is no need for explicit inheritance or interfaces; simply having the required method is sufficient.

Program:

```
class Duck:

def fly(self):

print("Duck flying")

class Plane:

def fly(self):

print("Plane flying")

class Superman:

def fly(self):

print("Superman flying")

def make_it_fly(entity):

entity.fly()

duck = Duck()

plane = Plane()

superman = Superman()

make_it_fly(duck)     # Output: Duck flying make_it_fly(plane)    # Output: Plane flying
make_it_fly(superman) # Output: Superman flyingOutput:
```

```
Duck flying Plane flying Superman flying
```

Lecture-30

Using Properties to Control Attribute Access

Basic Property (@property)

Objective: To understand how to create a getter method for an attribute using the @property decorator, which allows a method to be accessed like an attribute.

Theory: In Python, the @property decorator is a powerful tool for creating "smart" attributes. It transforms a method into a read-only property, meaning you can access it like a variable without using parentheses. This is a common form of encapsulation, allowing you to hide internal logic from the user while still providing a simple interface. The internal, "private" attribute is typically prefixed with a single underscore (_).

Program:

```
class Student:

def __init__(self, name, age):

self._name = name  # A "protected" attribute

self._age = age    # A "protected" attribute

# The @property decorator turns this method into a getter for 'name'. @property

def name(self):

"""Getter for the student's name."""

return self._name

# The @property decorator turns this method into a getter for 'age'. @property

def age(self):

"""Getter for the student's age.""" return self._age

# Create a Student object.student1 = Student("Alice", 20)

# Access the properties like attributes, without using parentheses. print(f"Student Name: {student1.name}")

print(f"Student Age: {student1.age}")
```

Output:

```
Student Name: Alice Student Age: 20
```

Adding a Setter (@property.setter)

Objective: To learn how to add a setter method to a property to control how an attribute's value is modified.

Theory: While @property creates a read-only attribute, you can make it writable by defining a corresponding setter method. The setter is defined by using the @<property_name>.setter decorator. This allows you to add validation logic to ensure that an attribute is only set to a valid value, preventing errors and maintaining data integrity.

Program:

```
class Rectangle:

def __init__(self, width, height): self._width = width

self._height = height

@property

def width(self): return self._width

@width.setter def width(self, value):

if not isinstance(value, (int, float)) or value < 0:raise ValueError("Width must be a non-negative number.") self._width = value

@property

def height(self):

return self._height

@height.setter

def height(self, value):

if not isinstance(value, (int, float)) or value < 0:

raise ValueError("Height must be a non-negative number.") self._height = value

# Create a Rectangle object.

rect = Rectangle(10, 5)

# Try to set valid values.

rect.width = 15

print(f"New width: {rect.width}")

# Try to set an invalid value, which will raise an error. try:

rect.width = -5 except ValueError as e:

print(f"Error: {e}")

try:

rect.height = "invalid" except ValueError as e:
```

print(f"Error: {e}")Output:

```
New width: 15

Error: Width must be a non-negative number. Error: Height must be a non-negative number.
```

Computed Properties

Objective: To use @property to create a "computed" attribute whose value is derived from other attributes and is not stored directly.

Theory: Properties are not limited to wrapping existing attributes. You can also use them to create computed attributes. These are attributes that are calculated on the fly when they are accessed. For example, a total_price property might calculate its value based on a price and a quantity attribute. This is a clean way to provide a dynamic value without having to manually update a variable every time a change occurs.

Program:

```
class ShoppingCart:

def __init__(self):

self._items = {} # Dictionary to store item and quantity

def add_item(self, item, quantity, price):

self._items[item] = {"quantity": quantity, "price": price}

# A computed property that calculates the total cost. @property

def total_cost(self):

total = 0

for item in self._items.values():

total += item["quantity"] * item["price"] return total# Create a shopping cart.

cart = ShoppingCart()

cart.add_item("Apple", 5, 0.50)

cart.add_item("Banana", 3, 0.25)

# Access the computed property.

print(f"Total cost of cart: ${cart.total_cost}")

# Add another item and see the total cost automatically update. cart.add_item("Orange", 2, 0.75)

print(f"Total cost after adding oranges: ${cart.total_cost}")
```

Output:

```
Total cost of cart: $3.25

Total cost after adding oranges: $4.75
```

Lecture-31

File Handling - Reading and Writing Binary Data

Writing and Reading Binary Data

Objective: To understand how to open, write, and read data from a file in binary mode.

Theory: Binary files store data in the form of bytes, not human-readable text. Examples include images, audio files, and executables. When working with binary data, you must open the file in binary mode by adding a 'b' to the mode string (e.g., 'wb' for write-binary, 'rb' for read-binary).

- Writing Binary Data: You must convert the data (like text strings or numbers) into bytes before writing to the file. Python's built-in bytes() function or a string's encode() method can be used for this.
- Reading Binary Data: When you read from a binary file, the data comes back as a bytes object. You must then decode it back into a string using the decode() method if it contains text.

Program:

```
# Create a sample text string and an integer.

text_data = "This is a secret message."

integer_data = 12345

# --- Writing to a Binary File ---

# Open the file in 'write binary' mode ('wb').

# 'wb' creates a new file or overwrites an existing one. try:

with open("binary_file.bin", "wb") as file: print("Writing data to 'binary_file.bin'...") # Encode the string into bytes using UTF-8. file.write(text_data.encode('utf-8'))

# Binary data is often structured. For simplicity, we'll encode # the integer as a string and then as bytes.

file.write(str(integer_data).encode('utf-8'))print("Data written successfully.")

except IOError as e:

print(f"Error writing to file: {e}")

# --- Reading from the Binary File ---

# Open the file in 'read binary' mode ('rb').

try:
```

```
with open("binary_file.bin", "rb") as file:

print("Reading data from 'binary_file.bin'...")

# Read all bytes from the file.

read_data = file.read()

# Decode the bytes back to a string using UTF-8. decoded_data = read_data.decode('utf-8')

print("Data read successfully.")

print(f"Content read from file: {decoded_data}")

except IOError as e:

print(f"Error reading from file: {e}")
```

Output:

```
Writing data to 'binary_file.bin'...

Data written successfully.

Reading data from 'binary_file.bin'...

Data read successfully.

Content read from file: This is a secret message.12345Appending to a Binary File
```

Objective: To learn how to add new data to an existing binary file without overwriting its content. Theory: To add new data to the end of a binary file, you must open it in append-binary mode ('ab'). This mode moves the file pointer to the end of the file before writing, so any new data is simply added after the existing content.

Program:

```
# Assume 'binary_file.bin' from Experiment 1 exists. new_data = " -- A new message appended."

# --- Appending to a Binary File ---

# Open the file in 'append binary' mode ('ab').

try:

with open("binary_file.bin", "ab") as file:

print("Appending new data to 'binary_file.bin'...") # Encode and write the new data.

file.write(new_data.encode('utf-8')) print("Data appended successfully.")

except IOError as e:

print(f"Error appending to file: {e}")

# --- Reading the updated Binary File ---

# Open in 'read binary' mode to see the combined content. try:

with open("binary_file.bin", "rb") as file: read_data = file.read()
```

```
decoded_data = read_data.decode('utf-8') print("\nReading the entire file again:")print(f"Updated
content: {decoded_data}")

except IOError as e:

print(f"Error reading from file: {e}")
```

Output:

```
Appending new data to 'binary_file.bin'...

Data appended successfully.

Reading the entire file again:

Updated content: This is a secret message.12345 -- A new message appended.
```

Lecture-32

Writing and Parsing Text Files

Writing and Appending to a Text File

Objective: To learn how to create and write content to a text file in Python, and how to add new data to it without overwriting existing content.

Theory: File handling is a fundamental part of programming. When working with text files, you use specific modes to tell Python how to interact with the file.

- 'w' (Write Mode): Opens a file for writing. If the file already exists, its content is completely overwritten. If it doesn't exist, a new file is created.
- 'a' (Append Mode): Opens a file for writing. If the file exists, new content is added to the end of the file. If it doesn't exist, a new file is created. The open() function is used to get a file object, and the with statement is the recommended way to handle files, as it ensures the file is automatically closed, even if errors occur.

Program:

```
# --- Writing to a new file ---

# Open the file in 'write' mode ('w').

# 'with open(...)' ensures the file is closed automatically. try:

with open("notes.txt", "w") as file:

print("Writing to 'notes.txt'...")

file.write("Python Programming Notes\n") file.write("1. Functions\n")

file.write("2. Classes\n")

print("Content written successfully.") except IOError as e:

print(f"Error writing to file: {e}")

# --- Appending to the same file ---

# Open the file in 'append' mode ('a'). try:

with open("notes.txt", "a") as file:print("\nAppending to 'notes.txt'...") file.write("3. File Handling\n")

print("Content appended successfully.") except IOError as e:

print(f"Error appending to file: {e}")
```

Output:

Writing to 'notes.txt'... Content written successfully.

Appending to 'notes.txt'... Content appended successfully.

Reading and Parsing a Text File

Objective: To learn how to read data from a text file and parse its content for use in a program. Theory: To read a file, you open it in read mode ('r'). You can read the entire content at once or process it line by line. Processing line by line is generally more memory-efficient for very large files.

- read(): Reads the entire file content into a single string.
- readline(): Reads a single line from the file.
- readlines(): Reads all lines from the file into a list of strings.
- Looping over the file object: A common and efficient way to read a file line by line.

Parsing involves processing the read data to extract meaningful information, often by splitting strings or converting data types.

Program:

```
# The 'notes.txt' file from Experiment 1 now contains all the notes. # We will read this file and parse its contents.

# --- Reading and parsing line by line ---

try:

print("Reading and parsing 'notes.txt' line by line:")with open("notes.txt", "r") as file:

# Looping over the file object is an efficient way to read line by line. for line_number, line in enumerate(file, 1):

# The 'line' variable includes a newline character, so we use .strip(). clean_line = line.strip()

# Check if the line is not empty before parsing. if clean_line:

if clean_line.startswith("Python"): print(f"Header: {clean_line}")

elif clean_line.startswith("3."):

# Parsing the line to extract the topic. parts = clean_line.split(". ") print(f"Topic {parts[0]}: {parts[1]}")

else:

print(f"Normal line: {clean_line}")

except FileNotFoundError:

print("The file 'notes.txt' was not found.") except IOError as e:

print(f"Error reading the file: {e}")
```

Output:

Reading and parsing 'notes.txt' line by line: Header: Python Programming Notes Normal line: 1. Functions

Normal line: 2. Classes Topic 3: File Handling