OOPS

LAB MANUAL

Dr. Muhammad Siddique

Object-Oriented Programming Lab Manual (Using Python)

Dr. Muhammad Siddique

February 2025

Contents

1 Week 1: Introduction to Object-Oriented Programming in Python

Objective: Understand the basics of Object-Oriented Programming (OOP) and implement simple classes and objects in Python.

Tasks:

1. **Task 1:** Understand the four main principles of OOP: Encapsulation, Abstraction, Inheritance, and Polymorphism.
2. **Task 2:** Write a Python program to define a simple class and create an object of that class.
3. **Task 3:** Define a class with a constructor (`__init__` method) and instantiate objects.
4. **Task 4:** Implement instance variables and methods inside a class.
5. **Task 5:** Research and list five real-world applications of OOP.

Details of Lab Experiment:

Task 1: Understanding the principles of OOP

OOP consists of four main principles:

- **Encapsulation:** Binding data and methods together.
- **Abstraction:** Hiding implementation details from the user.
- **Inheritance:** Creating a new class from an existing class.
- **Polymorphism:** Using a unified interface to represent different functionalities.

Task 2: Define a simple class and create an object

In this task, you will define a basic class and create an object of that class.

```python
class Student:
    def greet(self):
        print("Hello, welcome to OOP with Python!")

# Creating an object of the class
s = Student()
s.greet()
```

Listing 1: Simple class and object

Explanation:

- The `Student` class contains a method `greet()`.
- An object `s` of class `Student` is created, and `greet()` is called using that object.

Task 3: Define a class with a constructor

This task demonstrates the use of the constructor method `__init__` to initialize an object.

```python
class Student:
    def __init__(self, name):
        self.name = name
        print(f"Student name is {self.name}")

# Creating an object of the class
s1 = Student("Ali")
```

Listing 2: Class with constructor

Explanation:

- The `__init__` method is a constructor that initializes the `name` attribute.
- When an object is created, the constructor is automatically invoked.

Task 4: Implement instance variables and methods

Define and access instance variables inside a class.

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

    def display_info(self):
        print(f"Car Brand: {self.brand}, Model: {self.model}
    ")

# Creating objects
car1 = Car("Toyota", "Corolla")
car1.display_info()
```

Listing 3: Instance variables and methods

Explanation:

- The class `Car` has two instance variables: `brand` and `model`.
- The method `display`$_info()printsthecardetails.$

Task 5: Research five real-world applications of OOP

OOP is widely used in software development. Here are five real-world applications:

- Graphical User Interfaces (GUIs) such as Tkinter and PyQt
- Game development using Pygame and Unity
- Web applications using Django and Flask
- Data Science and Machine Learning using Pandas and Scikit-learn
- Embedded Systems and Robotics

2 Week 2: Classes and Objects

Objective: Understand the concepts of classes and objects in Python. Learn how to create classes, instantiate objects, and use instance variables and methods.

Tasks:

1. **Task 1:** Define a simple class and create an object from it.
2. **Task 2:** Implement a class with instance variables and a method.
3. **Task 3:** Create a class with a constructor to initialize instance variables.
4. **Task 4:** Implement a class with multiple methods, including a getter and setter.
5. **Task 5:** Write a program to demonstrate multiple objects of the same class.

Details of Lab Experiment:

Task 1: Define a simple class and create an object

Define a class named `Car` and create an instance of it.

```python
class Car:
    pass

# Create an instance of Car class
my_car = Car()
print(type(my_car))
```

Listing 4: Simple Class Example

Task 2: Implement a class with instance variables and a method

Define a class with attributes and a method to display them.

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

    def display_info(self):
        print(f"Car: {self.brand} {self.model}")

# Create an object
car1 = Car("Toyota", "Corolla")
```

```python
car1.display_info()
```

Listing 5: Class with Instance Variables

Task 3: Create a class with a constructor to initialize instance variables

Define a class with a constructor to initialize instance variables.

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

    def show_details(self):
        print(f"Student: {self.name}, Age: {self.age}")

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

Listing 6: Class with Constructor

Task 4: Implement a class with multiple methods, including a getter and setter

Define a class with private attributes and use getter and setter methods.

```python
class BankAccount:
    def __init__(self, owner, balance):
        self.owner = owner
        self.__balance = balance  # Private variable

    def get_balance(self):
        return self.__balance

    def deposit(self, amount):
        if amount > 0:
            self.__balance += amount
            print(f"Deposited: {amount}")
        else:
            print("Invalid deposit amount")

# Create object
account = BankAccount("John Doe", 5000)
print("Balance:", account.get_balance())
account.deposit(1000)
print("Updated Balance:", account.get_balance())
```

Listing 7: Class with Getter and Setter

Task 5: Write a program to demonstrate multiple objects of the same class

Define a class and create multiple instances.

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

    def bark(self):
        print(f"{self.name} says Woof!")

# Create multiple objects
dog1 = Dog("Buddy", "Labrador")
dog2 = Dog("Max", "Beagle")

dog1.bark()
dog2.bark()
```

Listing 8: Multiple Objects Example

Expected Outcome:

Students will understand how to define and use classes, create instances, initialize instance variables using constructors, and implement methods in Python.

3 Week 3: Inheritance and Polymorphism

Objective: Understand the concepts of inheritance and polymorphism in Python. Learn how to create subclasses, override methods, and use polymorphism to enhance code reusability.

Tasks:

1. **Task 1:** Create a base class and derive a subclass from it.
2. **Task 2:** Implement method overriding in a subclass.
3. **Task 3:** Demonstrate multiple levels of inheritance.
4. **Task 4:** Use polymorphism with different classes.
5. **Task 5:** Implement an abstract base class.

Details of Lab Experiment:

Task 1: Create a base class and derive a subclass from it

Define a base class `Animal` and a subclass `Dog` that inherits from it.

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

    def speak(self):
        return "Some sound"

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

# Create an object of Dog class
dog = Dog("Buddy")
print(dog.name, "says", dog.speak())
```

Listing 9: Simple Inheritance Example

Task 2: Implement method overriding in a subclass

Override a method from the base class in the subclass.

```python
class Parent:
    def show(self):
        print("This is the parent class")

class Child(Parent):
    def show(self):
        print("This is the child class")

# Create an object of Child class
obj = Child()
obj.show()
```

Listing 10: Method Overriding Example

Task 3: Demonstrate multiple levels of inheritance

Create a chain of inheritance with three classes.

```python
class Grandparent:
    def display(self):
        print("This is the grandparent class")

class Parent(Grandparent):
    def display(self):
        print("This is the parent class")

class Child(Parent):
    def display(self):
        print("This is the child class")

# Create an object of Child class
c = Child()
c.display()
```

Listing 11: Multilevel Inheritance Example

Task 4: Use polymorphism with different classes

Demonstrate polymorphism by defining multiple classes with a common method.

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

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

# Function that takes any object
def animal_speak(animal):
    print(animal.speak())

# Create objects
cat = Cat()
dog = Dog()

```

```python
animal_speak(cat)
animal_speak(dog)
```

Listing 12: Polymorphism Example

Task 5: Implement an abstract base class

Use the `abc` module to create an abstract base class.

```python
from abc import ABC, abstractmethod

class Vehicle(ABC):
    @abstractmethod
    def start_engine(self):
        pass

class Car(Vehicle):
    def start_engine(self):
        print("Car engine started")

# Create an object of Car class
car = Car()
car.start_engine()
```

Listing 13: Abstract Base Class Example

Expected Outcome:

Students will learn about inheritance and polymorphism, understand how to override methods, implement multi-level inheritance, and use abstract base classes in Python.

4 Week 4: Encapsulation and Data Hiding

Objective: Understand the principles of encapsulation and data hiding in Python. Learn how to use access specifiers, getter and setter methods, and name mangling to control data access.

Tasks:

1. **Task 1:** Implement a class with private and public attributes.
2. **Task 2:** Use getter and setter methods to access private attributes.
3. **Task 3:** Demonstrate name mangling in Python.
4. **Task 4:** Implement encapsulation using a real-world example.
5. **Task 5:** Create a class with read-only properties.

Details of Lab Experiment:

Task 1: Implement a class with private and public attributes

Define a class `Person` with private and public attributes.

```python
class Person:
    def __init__(self, name, age):
        self.name = name       # Public attribute
        self.__age = age       # Private attribute

    def display(self):
        print(f"Name: {self.name}, Age: {self.__age}")

# Create an object
p = Person("John", 25)
p.display()
print(p.name)  # Accessible
# print(p.__age)  # Will raise an AttributeError
```

Listing 14: Private and Public Attributes

Task 2: Use getter and setter methods to access private attributes

Define methods to get and set private attributes safely.

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

    def get_age(self):
        return self.__age

    def set_age(self, age):
        if age > 0:
            self.__age = age
        else:
            print("Age must be positive!")

# Create an object
p = Person("Alice", 30)
print(p.get_age())
p.set_age(35)
print(p.get_age())
```

Listing 15: Using Getters and Setters

Task 3: Demonstrate name mangling in Python

Illustrate how name mangling works to access private attributes.

```python
class Test:
    def __init__(self):
        self.__hidden = "This is private"

    def reveal(self):
        return self.__hidden

# Create an object
t = Test()
# print(t.__hidden)  # Will raise an AttributeError
print(t._Test__hidden)  # Access using name mangling
```

Listing 16: Name Mangling Example

Task 4: Implement encapsulation using a real-world example

Create a `BankAccount` class that encapsulates data and restricts direct access.

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

    def deposit(self, amount):
        if amount > 0:
            self.__balance += amount
            print(f"Deposited: {amount}")
        else:
            print("Invalid amount")

    def withdraw(self, amount):
        if 0 < amount <= self.__balance:
            self.__balance -= amount
            print(f"Withdrawn: {amount}")
        else:
            print("Insufficient funds or invalid amount")

    def get_balance(self):
        return self.__balance

# Create an object
account = BankAccount("12345", 1000)
account.deposit(500)
account.withdraw(200)
print("Balance:", account.get_balance())
```

Listing 17: Encapsulation Example

Task 5: Create a class with read-only properties

Use the `@property` decorator to create read-only attributes.

```python
class Car:
    def __init__(self, model, year):
        self.__model = model
        self.__year = year

    @property
    def model(self):
        return self.__model

    @property
    def year(self):
        return self.__year

# Create an object
car = Car("Toyota", 2022)
print(car.model, car.year)
# car.model = "Honda"  # Will raise an AttributeError
```

Listing 18: Read-Only Property

Expected Outcome:

Students will learn about encapsulation, access control, and data hiding in Python. They will be able to use getter and setter methods, understand name mangling, and implement encapsulation in real-world applications.

5 Week 5: Operator Overloading and Magic Methods

Objective: Understand how to use operator overloading and magic methods in Python to customize the behavior of objects.

Tasks:

1. **Task 1:** Implement operator overloading for arithmetic operations.
2. **Task 2:** Overload comparison operators.
3. **Task 3:** Use magic methods for object representation.
4. **Task 4:** Implement custom behavior for indexing and length.
5. **Task 5:** Overload the call operator.

Details of Lab Experiment:

Task 1: Implement operator overloading for arithmetic operations

Demonstrate how to overload the '+' operator for a custom class.

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

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

p1 = Point(2, 3)
p2 = Point(4, 5)
p3 = p1 + p2
print(f"({p3.x}, {p3.y})")
```

Listing 19: Operator Overloading Example

Task 2: Overload comparison operators

Demonstrate how to overload the '¿' operator.

```python
class Box:
    def __init__(self, volume):
        self.volume = volume

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

box1 = Box(10)
box2 = Box(20)
print(box1 > box2)
```

Listing 20: Comparison Operator Overloading Example

Task 3: Use magic methods for object representation

Override the ‘$_{str‘and‘repr‘methods.}$

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

    def __str__(self):
        return f"Car Model: {self.model}, Year: {self.year}"

    def __repr__(self):
        return f"Car('{self.model}', {self.year})"

c = Car("Toyota", 2022)
print(str(c))
print(repr(c))
```

Listing 21: Magic Methods for Representation Example

Task 4: Implement custom behavior for indexing and length

Overload ‘$_{getitem‘and‘len‘methods.}$

```
class Team:
    def __init__(self, members):
        self.members = members

    def __getitem__(self, index):
        return self.members[index]

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

team = Team(["Alice", "Bob", "Charlie"])
print(team[1])  # Bob
print(len(team)) # 3
```

Listing 22: Custom Indexing and Length Example

Task 5: Overload the call operator

Implement the ‘$_{call_{‘methodtomakeanobjectcallable.}}$

```python
class Counter:
    def __init__(self, start=0):
        self.count = start

    def __call__(self, increment=1):
        self.count += increment
        return self.count

c = Counter()
print(c())    # 1
print(c(5))   # 6
```

Listing 23: Call Operator Overloading Example

Expected Outcome:

Students will learn how to customize the behavior of Python objects using operator overloading and magic methods, making classes more intuitive and powerful.

6 Week 6: Exception Handling in Python

Objective: Learn how to handle exceptions in Python, using try, except, else, and finally blocks to improve the robustness of Python programs.

Tasks:

1. **Task 1:** Understand the concept of exceptions and error handling in Python.
2. **Task 2:** Write a program that handles exceptions using the `try-except` block.
3. **Task 3:** Use `else` and `finally` blocks in exception handling.
4. **Task 4:** Demonstrate custom exception handling by creating a custom exception class.
5. **Task 5:** Handle multiple exceptions in a single program.

Details of Lab Experiment:

Task 1: Understanding exceptions and error handling in Python

Exceptions are errors that occur during the execution of a program. Python provides a mechanism to catch and handle these errors, preventing the program from crashing. Exception handling allows the programmer to respond to exceptions and take corrective actions.

Task 2: Handle exceptions using the `try-except` block

In this task, you will write a program that attempts an operation and handles any exceptions that may occur using the `try-except` block.

```python
try:
    num = int(input("Enter a number: "))
    print("The number is:", num)
except ValueError:
    print("Error: Invalid input. Please enter an integer.")
```

Listing 24: Handling Exceptions Using try-except

Explanation:

- The `try` block contains the code that may raise an exception.
- The `except` block catches the exception and executes if the exception occurs.

Task 3: Use `else` and `finally` blocks in exception handling

The `else` block is executed if no exceptions are raised, and the `finally` block is always executed, regardless of whether an exception occurs or not.

```python
try:
    num1 = int(input("Enter first number: "))
    num2 = int(input("Enter second number: "))
    result = num1 / num2
except ZeroDivisionError:
    print("Error: Division by zero is not allowed.")
else:
    print("The result is:", result)
finally:
    print("This block is always executed.")
```

Listing 25: Using else and finally blocks

Explanation:

- If no exception occurs, the `else` block runs.
- The `finally` block runs regardless of exceptions.

Task 4: Demonstrate custom exception handling by creating a custom exception class

Python allows you to create your own exception classes by inheriting from the base `Exception` class. In this task, you will create a custom exception class and raise it in your program.

```python
class NegativeValueError(Exception):
    pass

def check_value(value):
    if value < 0:
        raise NegativeValueError("Value cannot be negative")
    else:
        print("Value is:", value)

try:
    check_value(-5)
except NegativeValueError as e:
    print(f"Error: {e}")
```

Listing 26: Custom Exception Handling

Explanation:

- The `NegativeValueError` class is a custom exception that inherits from `Exception`.
- The `check_value` function raises this exception if the value is negative.

Task 5: Handle multiple exceptions in a single program

You can handle multiple exceptions using multiple `except` blocks or a single `except` block that handles different exceptions.

```python
try:
    num1 = int(input("Enter a number: "))
    num2 = int(input("Enter another number: "))
    result = num1 / num2
except ValueError:
    print("Error: Please enter valid integers.")
except ZeroDivisionError:
    print("Error: Division by zero is not allowed.")
except Exception as e:
    print(f"An unexpected error occurred: {e}")
else:
    print("The result is:", result)
finally:
    print("Execution completed.")
```

Listing 27: Handling Multiple Exceptions

Explanation:

- The `ValueError` block catches input errors.
- The `ZeroDivisionError` block catches division by zero errors.
- The `Exception` block catches any other unexpected exceptions.

Key Points to Remember:

- `try-except` blocks are used for catching and handling exceptions.
- `else` block executes only if no exceptions are raised in the `try` block.
- `finally` block always executes, regardless of exceptions.
- Custom exceptions can be created by inheriting from the base `Exception` class.
- Multiple exceptions can be handled by using multiple `except` blocks.

Additional Exercises:

1. Write a program to handle file operations. If the file does not exist, print an error message using exception handling.
2. Create a program that takes user input for an age and checks whether the user is eligible to vote. If the input is invalid, handle the exception properly.

3. Implement a calculator program that handles errors like division by zero and invalid operations using exception handling.

7 Week 7: Multiple and Multilevel Inheritance

Objective: Understand and implement multiple inheritance and multilevel inheritance in Python, along with the concept of Method Resolution Order (MRO).

Tasks:

1. **Task 1:** Implement multiple inheritance.
2. **Task 2:** Explore Method Resolution Order (MRO).
3. **Task 3:** Implement and demonstrate multilevel inheritance.
4. **Task 4:** Understand and resolve issues arising from multiple inheritance.

Details of Lab Experiment:

Task 1: Implementing Multiple Inheritance

In this task, you will create a class that inherits from two or more parent classes. This will demonstrate the ability to use features from multiple classes in one child class.

```python
class Animal:
    def sound(self):
        return "Animal sound"

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

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

# Multiple Inheritance
class Pet(Dog, Cat):
    def greet(self):
        return "Hello, I'm a pet!"

# Instantiate and test
pet = Pet()
print(pet.sound())  # Inherited from Animal
print(pet.speak())  # Inherited from Dog
print(pet.greet())  # Defined in Pet
```

Listing 28: Multiple Inheritance Example

Explanation:

- The `Pet` class inherits from both `Dog` and `Cat`.
- The `sound()` method is inherited from `Animal`, while `speak()` is inherited from `Dog`.
- `greet()` is a method unique to the `Pet` class.

Task 2: Method Resolution Order (MRO)

When using multiple inheritance, Python follows a specific order to resolve method calls. This order is called the Method Resolution Order (MRO).

```python
print(Pet.mro())
```

Listing 29: Method Resolution Order (MRO)

Explanation:

- The `mro()` method returns the order in which classes are inherited.
- The MRO ensures that Python knows the correct order of calling methods, particularly when multiple parent classes define the same method.

Task 3: Implementing Multilevel Inheritance

In multilevel inheritance, a class is derived from another class, which is also derived from a parent class. This demonstrates the hierarchy of inheritance.

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

class Dog(Animal):
    def speak(self):
        return "Dog barks"

class Puppy(Dog):
    def speak(self):
        return "Puppy barks cutely!"

# Instantiate and test
puppy = Puppy()
print(puppy.speak())  # Inherited from Puppy, but overridden
    in Dog
```

Listing 30: Multilevel Inheritance Example

Explanation:

- The `Puppy` class inherits from `Dog`, which in turn inherits from `Animal`.
- The `speak()` method is overridden at each level.

Task 4: Resolving Issues in Multiple Inheritance

Sometimes, multiple inheritance can lead to ambiguity, particularly when the same method is defined in multiple parent classes. This can be handled using the `super()` function or modifying the MRO.

Example of ambiguity resolution:

```python
class A:
    def greet(self):
        return "Hello from A"

class B(A):
    def greet(self):
        return "Hello from B"

class C(A):
    def greet(self):
        return "Hello from C"

class D(B, C):
    def greet(self):
        return super().greet()  # Uses super() to resolve
 ambiguity

# Instantiate and test
d = D()
print(d.greet())  # Resolves ambiguity using super()
```

Listing 31: Resolving Multiple Inheritance Ambiguity

Explanation:

- The `super()` function helps resolve ambiguity by explicitly calling the next class in the MRO.
- This is necessary when the same method is inherited from multiple parent classes.

Summary:

In this week's lab, you learned how to implement multiple and multilevel inheritance in Python. You also explored the Method Resolution Order (MRO) and resolved method ambiguity using `super()`.

Key Concepts:

- Multiple inheritance
- Method Resolution Order (MRO)
- Multilevel inheritance

- Using `super()` to resolve method ambiguity

8 Week 8: Polymorphism and Method Overloading

Objective: Understand the concepts of polymorphism, method overloading, operator overloading, and the dynamic nature of Python through duck typing.

Tasks:

1. **Task 1:** Implement function overloading using default arguments.
2. **Task 2:** Perform operator overloading for arithmetic operations.
3. **Task 3:** Demonstrate duck typing and dynamic typing in Python.

Details of Lab Experiment:

Task 1: Implementing Function Overloading Using Default Arguments

In Python, function overloading is achieved by using default arguments, allowing a function to accept a variable number of arguments.

```python
class Calculator:
    def add(self, a, b=0, c=0):
        return a + b + c

# Instantiate and test
calc = Calculator()
print(calc.add(10))           # Output: 10 (only one argument
    )
print(calc.add(10, 20))       # Output: 30 (two arguments)
print(calc.add(10, 20, 30))   # Output: 60 (three arguments)
```

Listing 32: Function Overloading using Default Arguments

Explanation:

- The `add()` method uses default values for `b` and `c`, allowing it to accept one, two, or three arguments.
- This simulates function overloading by varying the number of arguments passed.

Task 2: Operator Overloading

Operator overloading allows us to define how operators like `+`, `-`, `*`, etc., work with custom objects.

```python
class Point:
    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y

    # Overloading the + operator
    def __add__(self, other):
        return Point(self.x + other.x, self.y + other.y)

    # String representation for printing
    def __str__(self):
        return f"({self.x}, {self.y})"

# Instantiate and test
p1 = Point(1, 2)
p2 = Point(3, 4)
p3 = p1 + p2  # Operator overloading
print(p3)     # Output: (4, 6)
```

Listing 33: Operator Overloading Example

Explanation:

- The $_{a}dd_{()}methodoverloadsthe+operatorforthe\texttt{Point}class,allowingtheadditionoftwopoints.Thisdemonstrateshowcustomobjectsc$

Task 3: Duck Typing and Dynamic Typing

Duck typing in Python means that an object's behavior (methods and properties) determines its compatibility with another object, rather than its class type.

```python
class Dog:
    def speak(self):
        return "Bark!"

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

class AnimalCommunicator:
    def communicate(self, animal):
        return animal.speak()

# Instantiate and test
dog = Dog()
cat = Cat()
comm = AnimalCommunicator()

print(comm.communicate(dog))  # Output: Bark!
print(comm.communicate(cat))  # Output: Meow!
```

Listing 34: Duck Typing Example

Explanation:

- In this example, `Dog` and `Cat` do not share a common base class, but both implement a `speak()` method.
- The `AnimalCommunicator` class works with any object that implements `speak()`—this is the essence of duck typing.
- Python dynamically checks for the presence of the method, rather than the object's type, allowing for flexible and reusable code.

Summary:

In this lab, you learned about:

- Function overloading in Python using default arguments.
- Operator overloading by defining how operators work with custom objects.
- Duck typing and dynamic typing, demonstrating Python's flexibility in handling objects with similar behaviors, regardless of their actual type.

Key Concepts:

- Function overloading using default arguments
- Operator overloading
- Duck typing
- Dynamic typing

9 Week 9: Abstract Classes and Interfaces

Objective: Understand and implement abstract classes and interfaces using the `ABC` module in Python, and explore the need for abstraction in object-oriented programming.

Tasks:

1. **Task 1:** Implement an abstract class using the `ABC` module.
2. **Task 2:** Define and implement an interface in Python.
3. **Task 3:** Understand the necessity of abstraction in OOP through an example.

Details of Lab Experiment:

Task 1: Implementing an Abstract Class

Abstract classes provide a blueprint for derived classes. They cannot be instantiated directly and must contain at least one abstract method.

```python
from abc import ABC, abstractmethod

# Defining an abstract class
class Animal(ABC):
    @abstractmethod
    def make_sound(self):
        pass

# Concrete subclass
class Dog(Animal):
    def make_sound(self):
        return "Bark!"

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

# Instantiate and test
d = Dog()
c = Cat()
print(d.make_sound())  # Output: Bark!
print(c.make_sound())  # Output: Meow!
```

Listing 35: Abstract Class Example

Explanation:

- The `Animal` class is abstract and cannot be instantiated.

- Any subclass (like `Dog` and `Cat`) must implement the `make_sound()` method.
- This ensures all subclasses follow a consistent structure.

Task 2: Implementing an Interface

An interface is a contract that classes must follow. In Python, we use abstract classes to define interfaces.

```python
from abc import ABC, abstractmethod

# Defining an interface
class Vehicle(ABC):
    @abstractmethod
    def start_engine(self):
        pass

    @abstractmethod
    def stop_engine(self):
        pass

# Implementing the interface
class Car(Vehicle):
    def start_engine(self):
        return "Car engine started."

    def stop_engine(self):
        return "Car engine stopped."

class Bike(Vehicle):
    def start_engine(self):
        return "Bike engine started."

    def stop_engine(self):
        return "Bike engine stopped."

# Instantiate and test
car = Car()
bike = Bike()
print(car.start_engine())  # Output: Car engine started.
print(bike.start_engine())  # Output: Bike engine started.
```

Listing 36: Interface Implementation Example

Explanation:

- The `Vehicle` class defines an interface with two abstract methods.
- The `Car` and `Bike` classes implement these methods.
- Any class inheriting from `Vehicle` must implement `start_engine()` and `stop_engine()`.

Task 3: The Need for Abstraction in OOP

Abstraction hides implementation details and exposes only necessary features.

```python
from abc import ABC, abstractmethod

class Payment(ABC):
    @abstractmethod
    def process_payment(self, amount):
        pass

# Concrete classes
class CreditCardPayment(Payment):
    def process_payment(self, amount):
        return f"Processing credit card payment of ${amount}"

class PayPalPayment(Payment):
    def process_payment(self, amount):
        return f"Processing PayPal payment of ${amount}"

# Instantiate and test
payment_method = CreditCardPayment()
print(payment_method.process_payment(100))  # Output: Processing credit card payment of $100
```

Listing 37: Abstraction in OOP

Explanation:

- The `Payment` class defines a common interface for all payment methods.
- Specific payment methods (`CreditCardPayment`, `PayPalPayment`) implement the required method.
- This approach ensures modularity and scalability.

Summary:

In this week's lab, you explored:

- Abstract classes and how they enforce method implementation in derived classes.
- Interfaces in Python using the `ABC` module.
- The importance of abstraction in hiding implementation details and providing a clean interface.

10 Week 10: File Handling with OOP

Objective: Learn how to handle file operations in an object-oriented way, manage exceptions during file handling, and serialize objects using the `pickle` module in Python.

Tasks:

1. **Task 1:** Implement reading and writing files using classes and objects.
2. **Task 2:** Handle exceptions while working with file operations.
3. **Task 3:** Serialize and deserialize objects using the `pickle` module.

Details of Lab Experiment:

Task 1: Reading and Writing Files Using OOP

File handling is essential for storing and retrieving data. We use OOP principles to encapsulate file operations within a class.

```python
class FileManager:
    def __init__(self, filename, mode):
        """Initialize FileManager with filename and mode."""
        self.filename = filename
        self.mode = mode

    def write_file(self, content):
        """Write content to a file."""
        with open(self.filename, self.mode) as file:
            file.write(content)
        return "Data written successfully."

    def read_file(self):
        """Read content from a file."""
        with open(self.filename, 'r') as file:
            return file.read()

# Usage
file = FileManager("sample.txt", "w")
file.write_file("Hello, this is a test file.")

file = FileManager("sample.txt", "r")
print(file.read_file())  # Output: Hello, this is a test
    file.
```

Listing 38: File Handling with OOP

Explanation:

- The `FileManager` class encapsulates file operations.
- The `write_file()` method writes data to a file.
- The `read_file()` method reads data from a file.
- Using `with open()` ensures proper resource management.

Task 2: Handling Exceptions in File Operations

File handling can result in errors (e.g., file not found, permission issues). Exception handling ensures program stability.

```python
class SafeFileManager:
    def __init__(self, filename):
        self.filename = filename

    def safe_read(self):
        """Read a file safely with exception handling."""
        try:
            with open(self.filename, 'r') as file:
                return file.read()
        except FileNotFoundError:
            return "Error: File not found."
        except Exception as e:
            return f"An error occurred: {e}"

# Usage
file = SafeFileManager("non_existent.txt")
print(file.safe_read())  # Output: Error: File not found.
```

Listing 39: Exception Handling in File Operations

Explanation:

- The `safe_read()` method catches file-related errors.
- If the file is missing, it returns an appropriate error message.
- Generic exceptions are also handled to prevent program crashes.

Task 3: Serializing Objects Using `pickle`

Serialization allows storing objects in a file for later retrieval.

```python
import pickle

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

    def __repr__(self):
        return f"Student({self.name}, {self.age}, {self.grade})"

# Serialize object
student = Student("Alice", 20, "A")
with open("student.pkl", "wb") as file:
    pickle.dump(student, file)

# Deserialize object
with open("student.pkl", "rb") as file:
    loaded_student = pickle.load(file)

print(loaded_student)  # Output: Student(Alice, 20, A)
```

Listing 40: Object Serialization Using Pickle

Explanation:

- The `Student` class represents a student object.
- The `pickle.dump()` method stores the object in a binary file.
- The `pickle.load()` method retrieves and reconstructs the object.

Summary:

In this week's lab, you explored:

- Reading and writing files using classes and methods.
- Handling file-related exceptions to prevent program crashes.
- Serializing and deserializing objects using the `pickle` module.

11 Week 11: Exception Handling and Debugging

Objective: Learn how to handle errors using `try`, `except`, and `finally` blocks, raise custom exceptions, and use logging and debugging techniques to ensure smooth functioning of OOP programs.

Tasks:

1. **Task 1:** Handle errors using `try`, `except`, and `finally`.
2. **Task 2:** Raise custom exceptions to handle specific errors.
3. **Task 3:** Implement logging and debugging mechanisms in OOP programs.

Details of Lab Experiment:

Task 1: Handling Errors Using `try`, `except`, and `finally`

Error handling is essential in OOP for managing runtime exceptions. The `try`, `except`, and `finally` blocks allow you to catch and handle errors gracefully.

```python
class SafeMath:
    def divide(self, a, b):
        try:
            result = a / b
        except ZeroDivisionError:
            return "Error: Cannot divide by zero."
        except TypeError:
            return "Error: Invalid input type."
        finally:
            return "Execution complete."

# Usage
math = SafeMath()
print(math.divide(10, 2))  # Output: Execution complete.
print(math.divide(10, 0))  # Output: Error: Cannot divide by
    zero.
```

Listing 41: Handling Errors with try

Explanation:

- The `try` block attempts to execute code that may cause an exception.
- The `except` blocks handle specific errors such as division by zero and invalid input type.

- The `finally` block is executed regardless of whether an exception occurred, ensuring code execution completion.

Task 2: Raising Custom Exceptions

Sometimes, you need to raise custom exceptions to handle specific error conditions in your program.

```python
class InvalidAgeError(Exception):
    """Custom exception for invalid age."""
    pass

class Person:
    def __init__(self, name, age):
        self.name = name
        if age < 0:
            raise InvalidAgeError("Age cannot be negative.")
        self.age = age

# Usage
try:
    person = Person("Alice", -5)
except InvalidAgeError as e:
    print(f"Error: {e}")
```

Listing 42: Raising Custom Exceptions

Explanation:

- The `InvalidAgeError` class is a custom exception inheriting from the `Exception` class.
- If the age provided is negative, the exception is raised with an appropriate error message.
- The `try-except` block catches and handles the custom exception.

Task 3: Implementing Logging and Debugging

Logging and debugging are essential for tracking errors and understanding program behavior during execution. Python's `logging` module helps log error messages and other relevant information.

```python
import logging

# Configure the logging system
logging.basicConfig(level=logging.DEBUG, format='%(asctime)s
    - %(levelname)s - %(message)s')

class Calculator:
    def add(self, a, b):
        logging.debug(f"Adding {a} and {b}")
        return a + b

    def divide(self, a, b):
        try:
            logging.debug(f"Dividing {a} by {b}")
            return a / b
        except ZeroDivisionError:
            logging.error("Error: Division by zero.")
            return None

# Usage
calc = Calculator()
print(calc.add(5, 3))  # Output: 8
print(calc.divide(10, 0))  # Logs error: Error: Division by
    zero.
```

Listing 43: Using Logging and Debugging

Explanation:

- The `logging.debug()` method logs a debug message that helps trace program execution.
- The `logging.error()` method logs error messages if an exception occurs.
- The logging configuration sets the logging level to `DEBUG`, ensuring all messages from debug to critical are captured.

Summary:

In this week's lab, you learned:

- How to handle errors with `try`, `except`, and `finally` blocks.
- How to raise custom exceptions to handle specific errors in OOP.
- The basics of logging and debugging using Python's `logging` module to track program behavior and errors.

12 Week 12: Working with Modules and Packages

Objective: Understand how to create and use modules, import modules into OOP programs, and structure OOP projects with packages.

Tasks:

1. **Task 1:** Create and use Python modules.
2. **Task 2:** Import modules in an OOP program.
3. **Task 3:** Organize your project with packages for better structure and modularity.

Details of Lab Experiment:

Task 1: Creating and Using Python Modules

A module is simply a Python file containing classes, functions, and variables. You can create a module by writing Python code in a file with the `.py` extension.

```python
# Create a module called math_operations.py
def add(a, b):
    return a + b

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

# Now import the module in another script (main.py)
import math_operations

result = math_operations.add(5, 3)
print(f"Addition Result: {result}")  # Output: Addition
    Result: 8
```

Listing 44: Creating and Using a Module

Explanation:

- In the above example, the module `math_operations.py` contains two functions: `add()` and `subtract()`.
- In the main script `main.py`, we import the module using `import math_operations` and then call its functions.

Task 2: Importing Modules in OOP Programs

In OOP, modules can be used to import classes and functions into your programs. This helps separate concerns and keep the code modular.

```python
# Create a module called employee.py
class Employee:
    def __init__(self, name, salary):
        self.name = name
        self.salary = salary

    def display_info(self):
        return f"Employee: {self.name}, Salary: {self.salary
    }"

# Now import and use the Employee class in another script (
    main.py)
from employee import Employee

emp = Employee("John Doe", 50000)
print(emp.display_info())  # Output: Employee: John Doe,
    Salary: 50000
```

Listing 45: Importing Modules in OOP Programs

Explanation:

- The module `employee.py` defines the class `Employee`.
- In `main.py`, we import the `Employee` class using `from employee import Employee` and instantiate an object of the class.

Task 3: Structuring OOP Projects with Packages

A package is a collection of Python modules organized in directories. Packages help structure large projects into manageable components.

```python
# Project directory structure:
# myproject/
#           main.py
#           employees/
#               __init__.py
#               manager.py
#               developer.py

# The __init__.py file marks the directory as a package.

# In manager.py:
class Manager:
    def __init__(self, name):
        self.name = name

    def display_info(self):
        return f"Manager: {self.name}"

# In developer.py:
class Developer:
    def __init__(self, name):
        self.name = name

    def display_info(self):
        return f"Developer: {self.name}"

# In main.py:
from employees.manager import Manager
from employees.developer import Developer

manager = Manager("Alice")
developer = Developer("Bob")

print(manager.display_info())  # Output: Manager: Alice
print(developer.display_info())  # Output: Developer: Bob
```

Listing 46: Structuring Projects with Packages

Explanation:

- The directory structure represents an organized project, with modules `manager.py` and `developer.py` inside the `employees` package.
- The `__init__.py` file marks the `employees` directory as a Python package.
- In `main.py`, we import the classes `Manager` and `Developer` from their respective modules within the package and use them.

Summary:

In this week's lab, you learned:

- How to create and use Python modules to encapsulate functionality.
- How to import modules and classes into OOP programs to maintain modularity and separation of concerns.
- How to structure large projects using packages to organize related modules and improve project maintainability.

13 Week 13: Database Connectivity using OOP

Objective: Understand how to use the `sqlite3` module in Python, create a database-driven application, and perform CRUD operations (Create, Read, Update, Delete) using Object-Oriented Programming.

Tasks:

1. **Task 1:** Introduction to `sqlite3` module.
2. **Task 2:** Create a database-driven application.
3. **Task 3:** Perform CRUD operations using OOP principles.

Details of Lab Experiment:

Task 1: Introduction to `sqlite3` Module

The `sqlite3` module in Python provides a lightweight, disk-based database that doesn't require a separate server process. It allows you to store and retrieve data in a database file.

```python
import sqlite3

# Connect to a SQLite database (or create one if it doesn't exist)
connection = sqlite3.connect('students.db')

# Create a cursor object using the connection
cursor = connection.cursor()

# Create a table
cursor.execute('''CREATE TABLE IF NOT EXISTS students (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    name TEXT,
    age INTEGER,
    grade TEXT
)''')

# Commit changes and close connection
connection.commit()
connection.close()
```

Listing 47: Introduction to sqlite3

Explanation:

- We connect to the `students.db` SQLite database. If the file doesn't exist, it will be created automatically.

- The `cursor.execute()` method is used to execute SQL commands (in this case, creating a table).
- Always commit the transaction using `connection.commit()` and close the connection with `connection.close()`.

Task 2: Creating a Database-Driven Application

Now, let's create a simple class to represent the database operations. This class will encapsulate the logic for interacting with the SQLite database.

```python
class StudentDatabase:
    def __init__(self, db_name):
        self.db_name = db_name
        self.connection = sqlite3.connect(db_name)
        self.cursor = self.connection.cursor()

    def create_table(self):
        self.cursor.execute('''CREATE TABLE IF NOT EXISTS students (
            id INTEGER PRIMARY KEY AUTOINCREMENT,
            name TEXT,
            age INTEGER,
            grade TEXT
        )''')
        self.connection.commit()

    def add_student(self, name, age, grade):
        self.cursor.execute('''INSERT INTO students (name, age, grade) VALUES (?, ?, ?)''', (name, age, grade))
        self.connection.commit()

    def fetch_all_students(self):
        self.cursor.execute('SELECT * FROM students')
        return self.cursor.fetchall()

    def close(self):
        self.connection.close()

# Usage
db = StudentDatabase('students.db')
db.create_table()
db.add_student('John Doe', 20, 'A')
students = db.fetch_all_students()
print(students)
db.close()
```

Listing 48: Creating a Database-Driven Application

Explanation:

- The `StudentDatabase` class handles all interactions with the database, such as creating tables, adding students, and fetching records.

- Methods like `add_student()` and `fetch_all_students()` interact with the database to perform CRUD operations.
- The connection is closed using the `close()` method after the operations are complete.

Task 3: Performing CRUD Operations using OOP

Let's implement the CRUD operations in the `StudentDatabase` class.

```python
class StudentDatabase:
    def __init__(self, db_name):
        self.db_name = db_name
        self.connection = sqlite3.connect(db_name)
        self.cursor = self.connection.cursor()

    def create_table(self):
        self.cursor.execute('''CREATE TABLE IF NOT EXISTS students (
            id INTEGER PRIMARY KEY AUTOINCREMENT,
            name TEXT,
            age INTEGER,
            grade TEXT
        )''')
        self.connection.commit()

    def add_student(self, name, age, grade):
        self.cursor.execute('''INSERT INTO students (name, age, grade) VALUES (?, ?, ?)''', (name, age, grade))
        self.connection.commit()

    def fetch_all_students(self):
        self.cursor.execute('SELECT * FROM students')
        return self.cursor.fetchall()

    def update_student(self, student_id, name, age, grade):
        self.cursor.execute('''UPDATE students SET name = ?, age = ?, grade = ? WHERE id = ?''',
                            (name, age, grade, student_id))
        self.connection.commit()

    def delete_student(self, student_id):
        self.cursor.execute('''DELETE FROM students WHERE id = ?''', (student_id,))
        self.connection.commit()

    def close(self):
        self.connection.close()

# Usage
db = StudentDatabase('students.db')
db.create_table()
db.add_student('Alice', 22, 'B')
students = db.fetch_all_students()
print(students)

# Update a student's grade
db.update_student(1, 'Alice', 22, 'A')
students = db.fetch_all_students()
print(students)

# Delete a student
db.delete_student(1)
students = db.fetch_all_students()
print(students)

db.close()
```

Listing 49: Performing CRUD Operations

Explanation:

- `add_student()` performs the "Create" operation.
- `fetch_all_students()` performs the "Read" operation.
- `update_student()` performs the "Update" operation.
- `delete_student()` performs the "Delete" operation.
- After performing any operation, we commit the changes to the database using `connection.commit()`.

Summary:

In this week's lab, you learned:

- How to use the `sqlite3` module to interact with SQLite databases.
- How to create a database-driven application using Object-Oriented Programming.
- How to perform CRUD operations (Create, Read, Update, Delete) using OOP principles.

14 Week 14: GUI Programming with Tkinter (OOP Approach)

Objective: Learn how to create classes for GUI components using the `Tkinter` library, handle events, and build an interactive application using Object-Oriented Programming principles.

Tasks:

1. **Task 1:** Introduction to Tkinter and creating classes for GUI components.
2. **Task 2:** Handle events in GUI applications.
3. **Task 3:** Build an interactive GUI application.

Details of Lab Experiment:

Task 1: Introduction to Tkinter and Creating Classes for GUI Components

The `Tkinter` library is the standard Python interface to the Tk GUI toolkit. It is used to create graphical user interfaces (GUIs) in Python applications.

Let's create a class that initializes a simple Tkinter window and adds a label and a button.

```python
import tkinter as tk

class SimpleApp:
    def __init__(self, master):
        self.master = master
        self.master.title('Simple Tkinter App')
        self.master.geometry('300x200')

        # Create a label
        self.label = tk.Label(self.master, text="Hello, Tkinter!", font=('Arial', 14))
        self.label.pack(pady=20)

        # Create a button
        self.button = tk.Button(self.master, text="Click Me", command=self.on_button_click)
        self.button.pack(pady=10)

    def on_button_click(self):
        self.label.config(text="Button Clicked!")

```

```python
# Create the main window
root = tk.Tk()
app = SimpleApp(root)
root.mainloop()
```

Listing 50: Creating GUI Components with Tkinter

Explanation:

- The `SimpleApp` class takes the `master` window as a parameter, which is the Tkinter root window.
- We create a label and a button, and use the `pack()` method to display them in the window.
- The button has a command attached to it (`on_button_click`), which will change the label's text when clicked.
- Finally, the `root.mainloop()` starts the Tkinter event loop, making the application interactive.

Task 2: Handling Events in GUI Applications

Event handling in Tkinter is achieved by binding events to specific actions. In the previous example, we handled the button click event with the `command` option.

Now, let's add event binding to handle a mouse click event.

```python
import tkinter as tk

class EventHandlingApp:
    def __init__(self, master):
        self.master = master
        self.master.title('Event Handling Example')
        self.master.geometry('300x200')

        # Create a label
        self.label = tk.Label(self.master, text="Click
anywhere!", font=('Arial', 14))
        self.label.pack(pady=20)

        # Bind the mouse click event
        self.master.bind("<Button-1>", self.on_mouse_click)

    def on_mouse_click(self, event):
        self.label.config(text=f"Mouse clicked at ({event.x
}, {event.y})")

# Create the main window
root = tk.Tk()
app = EventHandlingApp(root)
```

```python
root.mainloop()
```

Listing 51: Event Handling with Mouse Click

Explanation:

- The `master.bind("<Button-1>", self.on_mouse_click)` line binds the mouse click event to the `on_mouse_click` method.
- The `event` object provides information about the mouse click, such as the coordinates `event.x` and `event.y`.
- When the user clicks anywhere in the window, the label text is updated to show the coordinates of the click.

Task 3: Building an Interactive GUI Application

Now, let's build a simple interactive calculator application. This application will have buttons for digits and operations, and will display the result in a label.

```python
import tkinter as tk

class CalculatorApp:
    def __init__(self, master):
        self.master = master
        self.master.title('Simple Calculator')
        self.master.geometry('300x400')

        self.result_var = tk.StringVar()

        # Entry widget for displaying the result
        self.result_entry = tk.Entry(self.master,
    textvariable=self.result_var, font=('Arial', 14), bd=10,
    relief='sunken', justify='right')
        self.result_entry.grid(row=0, column=0, columnspan
    =4)

        # Button grid layout
        buttons = [
            ('7', 1, 0), ('8', 1, 1), ('9', 1, 2), ('/', 1,
    3),
            ('4', 2, 0), ('5', 2, 1), ('6', 2, 2), ('*', 2,
    3),
            ('1', 3, 0), ('2', 3, 1), ('3', 3, 2), ('-', 3,
    3),
            ('0', 4, 0), ('.', 4, 1), ('=', 4, 2), ('+', 4,
    3)
        ]

        for (text, row, col) in buttons:
            self.create_button(text, row, col)

    def create_button(self, text, row, col):
        button = tk.Button(self.master, text=text, font=('Arial', 14), width=5, height=2, command=lambda: self.on_button_click(text))
        button.grid(row=row, column=col)

    def on_button_click(self, text):
        current_text = self.result_var.get()

        if text == '=':
            try:
                result = str(eval(current_text))  # Evaluate the expression
                self.result_var.set(result)
            except:
                self.result_var.set("Error")
        else:
            self.result_var.set(current_text + text)

# Create the main window
root = tk.Tk()
app = CalculatorApp(root)
root.mainloop()
```

Listing 52: Building an Interactive Calculator

Explanation:

- The `CalculatorApp` class creates a calculator window with an entry widget for displaying the result.
- The buttons are arranged in a grid using `grid(row, column)`.
- Each button's command calls the `on_button_click` method, passing the button's text as an argument.
- The `on_button_click` method updates the displayed expression and evaluates it when the '=' button is pressed.
- If an error occurs (e.g., invalid expression), "Error" is displayed.

Summary:

In this week's lab, you learned:

- How to use Tkinter to create GUI components in an OOP manner.
- How to handle events (such as button clicks and mouse clicks) in Tkinter applications.

- How to build an interactive calculator application using Object-Oriented Programming principles.

15 Week 15: Multithreading and Concurrency in OOP

Objective: Learn how to create and manage threads using the `threading` module, understand synchronization mechanisms such as Lock and Semaphore, and work with thread-safe data structures.

Tasks:

1. **Task 1:** Create and manage threads using the `threading` module.
2. **Task 2:** Learn synchronization mechanisms (Lock, Semaphore).
3. **Task 3:** Work with thread-safe data structures.

Details of Lab Experiment:

Task 1: Creating and Managing Threads

Threads allow you to run multiple operations concurrently in your program. In Python, the `threading` module provides a way to handle this.

Let's create two threads that print messages concurrently.

```python
import threading
import time

def print_numbers():
    for i in range(1, 6):
        print(f"Number: {i}")
        time.sleep(1)

def print_letters():
    for letter in ['A', 'B', 'C', 'D', 'E']:
        print(f"Letter: {letter}")
        time.sleep(1)

# Creating threads
thread1 = threading.Thread(target=print_numbers)
thread2 = threading.Thread(target=print_letters)

# Starting threads
thread1.start()
thread2.start()

# Waiting for threads to finish
thread1.join()
thread2.join()

```

```python
print("Both threads have finished.")
```

Listing 53: Creating Threads using threading module

Explanation:

- We define two functions `print_numbers` and `print_letters` that print numbers and letters, respectively, with a 1-second delay.
- We create two threads using the `threading.Thread` class and pass the target functions as arguments.
- The `start()` method starts the threads, and `join()` waits for the threads to complete.

Task 2: Synchronization Mechanisms (Lock, Semaphore)

In multithreading, you need to ensure that shared resources are accessed safely. This can be achieved using synchronization mechanisms like Lock and Semaphore.

Let's use a `Lock` to prevent two threads from accessing the same resource at the same time.

```python
import threading

lock = threading.Lock()

def critical_section():
    with lock:
        print(f"Thread {threading.current_thread().name} is
    in the critical section")

# Creating threads
threads = [threading.Thread(target=critical_section) for _
    in range(3)]

# Starting threads
for thread in threads:
    thread.start()

# Waiting for threads to finish
for thread in threads:
    thread.join()

print("All threads have finished.")
```

Listing 54: Using Lock for Synchronization

Explanation:

- The `with lock` statement ensures that only one thread can enter the critical section at a time.

- The `Lock` object is used to prevent multiple threads from executing the critical section simultaneously.

Task 3: Thread-safe Data Structures

Thread-safe data structures are designed to be safely accessed and modified by multiple threads concurrently.

```python
import threading
import queue

# Creating a thread-safe Queue
q = queue.Queue()

def producer():
    for i in range(5):
        q.put(i)
        print(f"Produced: {i}")

def consumer():
    while not q.empty():
        item = q.get()
        print(f"Consumed: {item}")

# Creating threads
producer_thread = threading.Thread(target=producer)
consumer_thread = threading.Thread(target=consumer)

# Starting threads
producer_thread.start()
consumer_thread.start()

# Waiting for threads to finish
producer_thread.join()
consumer_thread.join()

print("Both producer and consumer have finished.")
```

Listing 55: Using Queue as a Thread-safe Data Structure

Explanation:

- We use a `Queue` from the `queue` module, which is thread-safe and allows threads to safely produce and consume items.
- The `put()` method is used to add items to the queue, and `get()` is used to remove items.
- The producer thread adds items to the queue, while the consumer thread processes them.

Summary:

In this week's lab, you learned:

- How to create and manage threads using the `threading` module.
- The use of synchronization mechanisms like `Lock` to avoid race conditions.
- How to use thread-safe data structures such as `Queue` to safely share data between threads.

16 Week 16: Final Project and Best Practices in OOP

Objective: Apply OOP concepts to a real-world project, implement design patterns, and write clean, maintainable code following OOP best practices.

Tasks:

1. **Task 1:** Apply OOP concepts to a real-world project.
2. **Task 2:** Learn about design patterns such as Singleton and Factory.
3. **Task 3:** Write clean and maintainable OOP code.

Details of Lab Experiment:

Task 1: Applying OOP Concepts to a Real-World Project

For the final project, students are required to create an application using OOP principles. The application should implement various OOP concepts such as classes, objects, inheritance, polymorphism, and exception handling. Here is an example of a simple library management system that utilizes these concepts.

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

    def __str__(self):
        return f"{self.title} by {self.author} (ISBN: {self.
isbn})"

class Library:
    def __init__(self):
        self.books = []

    def add_book(self, book):
        self.books.append(book)

    def display_books(self):
        for book in self.books:
            print(book)

# Create Library object
library = Library()

# Create Book objects
```

```python
book1 = Book("The Great Gatsby", "F. Scott Fitzgerald", "1234567890")
book2 = Book("1984", "George Orwell", "0987654321")

# Add books to library
library.add_book(book1)
library.add_book(book2)

# Display books in library
library.display_books()
```

Listing 56: Simple Library Management System

Explanation:

- We define a `Book` class with attributes like title, author, and ISBN, and a `Library` class to manage a collection of books.
- The library can add books and display all available books.
- This example applies OOP principles such as class definition, object instantiation, and method calls.

Task 2: Design Patterns

Design patterns are reusable solutions to common problems in software design. Two commonly used design patterns are Singleton and Factory.

Singleton Pattern: Ensures that a class has only one instance and provides a global point of access to that instance.

```python
class Singleton:
    _instance = None

    def __new__(cls):
        if cls._instance is None:
            cls._instance = super().__new__(cls)
        return cls._instance

# Create Singleton objects
s1 = Singleton()
s2 = Singleton()

print(s1 is s2)  # Output: True
```

Listing 57: Singleton Pattern

Explanation:

- The `Singleton` class ensures that only one instance of the class is created. The $_{new_{methodchecksifaninstancealreadyexists.Whens1ands2arecreated,bothpointtothesameobject.}}$

Task 3: Writing Clean and Maintainable OOP Code

Following best practices in OOP leads to clean and maintainable code. Some key guidelines include:

- Use meaningful class, method, and variable names.
- Keep classes focused on a single responsibility.
- Use inheritance and composition to avoid code duplication.
- Write unit tests for your code to ensure reliability.
- Document your code with comments and docstrings.

Example: Here is an example of a cleanly written `Car` class with a method for starting the engine.

```python
class Car:
    def __init__(self, make, model, year):
        """
        Initializes a new car object.
        :param make: The make of the car.
        :param model: The model of the car.
        :param year: The year the car was made.
        """
        self.make = make
        self.model = model
        self.year = year

    def start_engine(self):
        """
        Starts the car engine.
        """
        print(f"The engine of the {self.year} {self.make} {
    self.model} is now running.")

# Create a Car object
car = Car("Toyota", "Corolla", 2020)

# Start the engine
car.start_engine()
```

Listing 58: Clean OOP Code Example

Explanation:

- The class is well-documented with docstrings explaining the purpose of each method.
- Meaningful names are used for methods and attributes, and the class is focused on a single responsibility—representing a car.

Summary:

In this week's lab, you learned:

- How to apply OOP concepts to build real-world applications.
- The Singleton and Factory design patterns.
- Best practices for writing clean, maintainable, and efficient OOP code.