● OOPS ASSIGNMENT

☐ THEORY QUESTION

QUESTION1: What is Object-Oriented Programming (OOP)?

-> Object-Oriented Programming (OOP) is a way of writing code by creating "objects" that represent real-world things. These objects have properties (data) and behaviors (functions). OOP helps to keep code organized, reusable, and easier to manage. It uses key ideas like classes, objects, inheritance, polymorphism, encapsulation, and abstraction.

QUESTION2: What is a class in OOP?

-> In Object-Oriented Programming (OOP), a class is like a blueprint or a design that I create to describe how something should be structured. It's not the actual thing, but more like a guide for making things. Inside a class, I can define variables, which store data or information about the object, and I can also write functions, which are called methods, that describe what the object can do. Once I have a class, I can create multiple objects from it, and each object will follow the structure and behavior I set in the class. This makes my code more organized, reusable, and easier to manage when working on bigger projects.

QUESTION3: What is an object in OOP?

-> An object in OOP is like a real example of a class. If a class is the design or blueprint, then the object is the actual thing I create from that design. It holds real values in its variables and can do actions using the methods from the class. I can make many objects from the same class, and each one can have its own data, but they all follow the same structure.

QUESTION4:What is the difference between abstraction and encapsulation?

-> Abstraction is about showing only the important stuff and hiding all the extra details, so I can focus on what something does instead of how it works behind the scenes. Encapsulation is about wrapping up the data and code together in one place, like a class, and keeping some things private so no one else can mess with them directly. Basically, abstraction hides complexity, and encapsulation protects the data

---

QUESTION5: What are dunder methods in Python?

-> In Python, dunder methods (short for "double underscore methods") are special methods that have double underscores at both the beginning and the end of their names. These methods are also known as magic methods or special methods. Dunder methods are an essential feature in Python, enabling the customization of how objects of a class interact with built-in Python functions and operators. They provide a mechanism for defining the behavior of objects during various operations, such as comparisons, arithmetic operations, string representation, and more.

Common Dunder Methods:

init(self): This is the constructor method, which is called when an object is created from a class. It initializes the object's attributes and prepares the object for use. For example, **init** is used to define the initial state of an object when it is instantiated.

str(self): This method defines the string representation of an object. It is automatically called when the object is passed to the print() function or when

Python needs to convert the object to a string. The output of **str** should be a readable and user-friendly string representation of the object.

repr(self): Similar to **str**, the **repr** method returns a string that represents the object. However, the string returned by **repr** is meant to be a more detailed and unambiguous representation of the object, and ideally, it should allow the object to be recreated using eval().

add(self, other): This method allows the implementation of the addition operator (+) for objects of a class. When two objects of the class are added together using the + operator, Python will automatically call the **add** method to determine the result.

len(self): This method is used to define the behavior of the len() function for objects of a class. By implementing **len**, an object can return the length of its content, such as the number of elements in a list or characters in a string.

eq(self, other): This method is used to compare two objects for equality using the == operator. If the objects are equal, it returns True; otherwise, it returns False.

QUESTION6: Explain the concept of inheritance in OOP?

-> Inheritance is one of the fundamental concepts of Object-Oriented Programming (OOP). It is a mechanism by which one class can acquire the properties and behaviors (data members and member functions) of another class. The class that inherits is called the derived class or child class, and the class from which it inherits is called the base class or parent class.

The primary purpose of inheritance is to promote code reusability. Instead of writing the same code again in multiple classes, a programmer can write a common class and inherit its features wherever needed. Inheritance also helps in maintaining a clear hierarchical relationship between classes.

There are several types of inheritance in OOP:

1. Single Inheritance – A child class inherits from one parent class.
2. Multiple Inheritance – A child class inherits from more than one parent class.
3. Multilevel Inheritance – A class inherits from a child class which in turn inherits from another parent class.
4. Hierarchical Inheritance – Multiple child classes inherit from a single parent class.
5. Hybrid Inheritance – A combination of more than one type of inheritance.

QUESTION7: What is polymorphism in OOP?

-> Polymorphism is a key concept in Object-Oriented Programming (OOP) that refers to the ability of different objects to respond to the same method in different ways. It allows one interface to be used for a general class of actions, with specific behavior determined by the exact nature of the situation.

In simple terms, polymorphism means "many forms". It enables the same function or method to behave differently on different classes or objects. This enhances code flexibility and reusability.

- There are two main types of polymorphism in Python:

1. Compile-time Polymorphism (also called static polymorphism): Achieved through method overloading (not natively supported in Python) and operator overloading.
2. Run-time Polymorphism (also called dynamic polymorphism): Achieved through method overriding, where a subclass provides a specific implementation of a method already defined in its superclass.
   Example

[ ]

```
class Animal:

  def speak(self):

    print("The animal makes a sound")



class Dog(Animal):
```

```python
    def speak(self):

        print("The dog barks")



class Cat(Animal):

    def speak(self):

        print("The cat meows")



# Polymorphism in action

def animal_sound(animal):

    animal.speak()



# Different objects with the same interface

a1 = Dog()

a2 = Cat()



animal_sound(a1)

animal_sound(a2)
```

```
The dog barks
The cat meows
```

QUESTION8: How is encapsulation achieved in Python?

-> Encapsulation is one of the four fundamental concepts of Object-Oriented Programming (OOP), along with inheritance, polymorphism, and abstraction.

Encapsulation refers to the process of wrapping or bundling data (variables) and methods (functions) that operate on the data into a single unit known as a class. The main goal of encapsulation is to restrict direct access to some of the object's components, which can help prevent accidental modification of data and improve security. In Python, encapsulation is achieved using:

1. Classes: Python allows grouping data and methods inside a class to form a single unit.
2. Access Modifiers: Python uses naming conventions to indicate the level of access control for class attributes and methods.

- Public members: Accessible from anywhere. No underscore prefix is used. Example: self.name
- Protected members: Should not be accessed outside the class or its subclasses. Indicated by a single underscore _. Example: self._salary
- Private members: Not accessible directly outside the class. Indicated by a double underscore __. Python applies name mangling to make these attributes harder to access. Example: self._bonus
  Example:

```
[ ]
```

```
class Employee:

  def __init__(self, name, salary):

    self.name = name

    self._salary = salary

    self.__bonus = 5000



  def display_info(self):

    print("Name:", self.name)

    print("Salary:", self._salary)

    print("Bonus:", self.__bonus)
```

```
emp = Employee("John", 60000)

emp.display_info()



print(emp.name)

print(emp._salary)



print(emp._Employee__bonus)
```

```
Name: John
Salary: 60000
Bonus: 5000
John
60000
5000
```

QUESTION9: What is a constructor in Python?

-> In Object-Oriented Programming (OOP), a constructor is a special method that is automatically called when an object is created from a class. Its main purpose is to initialize the attributes of the object. Python, being an object-oriented language, supports constructors through a special method called *init*().

A constructor in Python is defined using the **init**() method. It is a built-in function in Python classes and is automatically invoked when a new object is instantiated. The constructor is used to assign values to the object's properties when it is created.

Example

[ ]

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

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

# Creating an object of Student class
s1 = Student("Alice", 20)
s1.show_details()
```

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

QUESTION10: What are class and static methods in Python?

-> In Python, methods defined within a class can be classified into three types: instance methods, class methods, and static methods. While instance methods are the most commonly used, class methods and static methods serve specific purposes that distinguish them from regular instance methods.

Class Methods:

A class method is a method that is bound to the class and not the instance of the class. It takes cls as its first parameter, which refers to the class itself, rather than self which refers to an instance. This allows class methods to modify or access class-level variables, but not instance-specific data.

Class methods are defined using the @classmethod decorator, and they are typically used to:

- Modify class-level data.
- Serve as alternative constructors.

Example

```
[ ]
```

```python
class Car:

    wheels = 4



    @classmethod

    def set_wheels(cls, count):

        cls.wheels = count



Car.set_wheels(6)

print(Car.wheels)
```

```
6
```

Static Methods:

A static method is a method that does not depend on class or instance-specific data. Unlike instance methods and class methods, a static method does not take self or cls as its first parameter. Static methods are defined using the @staticmethod decorator and behave like regular functions, but they belong to the class due to their logical association with the class.

Static methods do not access or modify any class or instance-level attributes. They are used for utility functions that perform operations related to the class but do not require access to the class or instance.

Example

```
[ ]
```

```
class Math:

    @staticmethod

    def add(x, y):

        return x + y



print(Math.add(5, 3))
```

```
8
```

QUESTION11: What is method overloading in Python?

-> Method overloading is a concept in object-oriented programming that allows a class to have more than one method with the same name but different parameters. It improves the readability of the program by allowing multiple ways to call the same method depending on the number or type of arguments passed. Method Overloading in Python:

Python does not support traditional method overloading like other languages. If multiple methods with the same name are defined in a class, only the last method is retained — earlier definitions are overwritten.

However, Python can simulate method overloading by using:

Default parameter values

Variable-length arguments using *args and* *kwargs

This allows a single method to handle different numbers or types of arguments.

Example: Using Default Parameters

```
[ ]
```

```
class Calculator:

    def add(self, a=0, b=0, c=0):

        return a + b + c



calc = Calculator()

print(calc.add(2))

print(calc.add(2, 3))

print(calc.add(2, 3, 4))
```

```
2
5
9
```

In the example above, the add() method works with one, two, or three arguments by assigning default values to the parameters.

Example: Using Variable Arguments

```
[ ]
```

```
class Calculator:

    def add(self, *args):
```

```
    return sum(args)



calc = Calculator()

print(calc.add(1, 2))

print(calc.add(1, 2, 3, 4))
```

```
3
10
```

Here, *args allows the method to accept any number of arguments, enabling flexible use similar to overloading.

QUESTION12: What is method overriding in OOP?

-> Method overriding is an important concept in Object-Oriented Programming (OOP) that allows a subclass (child class) to provide a specific implementation of a method that is already defined in its superclass (parent class). It is a form of runtime polymorphism, which means the method that gets executed is determined at runtime based on the object's type.

Method overriding occurs when a subclass defines a method with the same name, return type, and parameters as a method in its parent class. The overridden method in the subclass replaces the version defined in the parent class. This allows the subclass to provide behavior specific to its type, even when using the same method name. Example

---

```
[ ]
```

```
class Animal:
    def speak(self):
        print("The animal makes a sound.")


class Dog(Animal):
    def speak(self):
        print("The dog barks.")


# Create objects
a = Animal()
d = Dog()


a.speak()
d.speak()
```

```
The animal makes a sound.
The dog barks.
```

QUESTION13: What is a property decorator in Python?

-> The @property decorator is used to define a getter method, which allows a class method to be accessed like an attribute. It can be combined with @.setter and @.deleter to create corresponding setter and deleter methods.

The main purpose of the @property decorator is to provide controlled access to private attributes, allowing validation, computation, or logging when getting or setting values.

QUESTION14: Why is polymorphism important in OOP?

-> Polymorphism is one of the four fundamental principles of Object-Oriented Programming (OOP), along with inheritance, encapsulation, and abstraction. The term polymorphism comes from the Greek words poly (meaning “many”) and morph (meaning “forms”). In the context of OOP, polymorphism allows objects of different classes to be treated as objects of a common superclass. It enables a single function, method, or operator to work in different ways depending on the

Importance of Polymorphism in OOP:

1. Code Reusability and Flexibility:
   Polymorphism allows the same method or interface to behave differently based on the object calling it. This reduces code duplication and increases reusability, as the same code can handle different data types or classes without modification.
2. Improved Maintainability:
   Since polymorphism allows the use of a common interface for different data types, it becomes easier to maintain and modify the code. New classes can be added with minimal changes to the existing code.
3. Supports Extensibility:
   Polymorphism makes it easy to add new functionality to an existing system by introducing new subclasses. These new subclasses can override existing methods, providing specific behavior while still being used through a common interface.
4. Enables Dynamic Method Binding (Runtime Polymorphism):
   In runtime polymorphism, the method that is executed is determined at runtime based on the object. This allows more dynamic and flexible programs, where decisions are made during execution rather than at compile-time.
5. Enhances Code Readability and Organization:
   By allowing a unified interface for multiple classes, polymorphism helps in writing cleaner and more understandable code. It also helps in organizing the program structure effectively.
   Example

```
[ ]
```

```
class Animal:
    def speak(self):
        print("The animal makes a sound.")


class Dog(Animal):
    def speak(self):
        print("The dog barks.")


class Cat(Animal):
    def speak(self):
        print("The cat meows.")


# Polymorphism in action
def make_sound(animal):
    animal.speak()


make_sound(Dog())
make_sound(Cat())
```

```
The dog barks.
The cat meows.
```

QUESTION15: What is an abstract class in Python? -> An abstract class in Python is a class that contains one or more abstract methods. An abstract method is a method that is declared but contains no implementation. Abstract classes are used to define a common interface for a group of related classes and to ensure that certain methods are implemented in all subclasses.

To create an abstract class in Python, we use the following:

- Import the ABC class and abstract method decorator from the abc module.
- Inherit from ABC to make a class abstract.
- Use the @abstractmethod decorator to define abstract methods.

Example

[ ]

```
from abc import ABC, abstractmethod



class Animal(ABC):

    @abstractmethod

    def make_sound(self):

        pass



class Dog(Animal):

    def make_sound(self):

        return "Bark"



class Cat(Animal):

    def make_sound(self):

        return "Meow"



d = Dog()
```

```
print(d.make_sound())
```

```
Bark
```

QUESTION16: What are the advantages of OOP?

->Object-Oriented Programming (OOP) is a programming paradigm that organizes software design around objects, which are instances of classes. It focuses on using objects to represent real-world entities and promotes concepts like inheritance, encapsulation, abstraction, and polymorphism. OOP is widely used because it offers several advantages over traditional procedural programming.

Advantages of Object-Oriented Programming:

1. Modularity:

OOP allows developers to break down a program into smaller, manageable, and modular pieces using classes and objects. Each class can be developed, tested, and debugged independently, which improves code organization.

2.Reusability:

With the help of inheritance, classes can be reused in new programs. This avoids code duplication and reduces development time, as existing code can be extended rather than rewritten.

3. Encapsulation:

OOP supports the concept of encapsulation, which means that data (attributes) and the methods (functions) that operate on that data are bundled together inside a class. It hides the internal state of objects and protects data from unintended modifications.

4. Abstraction:

OOP promotes abstraction, where only essential details are shown to the user, and complex implementation details are hidden. This simplifies programming by reducing complexity and improving usability.

5. Polymorphism:

Polymorphism allows the same method or function name to behave differently based on the object that calls it. This makes it easier to write flexible and extensible code that can work with objects of different classes.

6. Easy Maintenance and Upgrades:

Due to the modular and reusable nature of OOP, programs are easier to update, debug, and maintain. New features can be added with minimal changes to the existing codebase.

7, Real-World Modeling:

OOP provides a way to model real-world entities as software objects. This helps in understanding and designing the system more intuitively and effectivel

---

QUESTION17: What is the difference between a class variable and an instance variable? -> Difference Between Class Variable and Instance Variable:

A class variable is a variable that is shared by all instances (objects) of a class. It is defined inside the class but outside any methods, and it holds the same value for every object unless explicitly changed. In contrast, an instance variable is specific to each object and is defined inside the constructor method (usually **init**) using self. Instance variables hold different values for different objects. For example, if you create multiple Student objects, each can have its own name and grade (instance variables), but all can share the same school_name (class variable).

QUESTION18: What is multiple inheritance in Python?

-> Multiple inheritance is a feature in Python where a class can inherit attributes and methods from more than one parent class. This allows the child class to access the functionality of multiple base classes. Python supports multiple inheritance directly, which helps in building complex systems by combining functionalities from different classes. However, it also requires careful design to avoid issues like the diamond problem, where the method resolution order (MRO) must be handled properly.

Example:

[ ]

```python
class Parent1:

    def display1(self):

        print("This is from Parent1")



class Parent2:

    def display2(self):

        print("This is from Parent2")



class Child(Parent1, Parent2):

    pass



c = Child()

c.display1()

c.display2()
```

```
This is from Parent1
This is from Parent2
```

QUESTION19: Explain the purpose of “**str**’ and ‘**repr**’ ‘ methods in Python.

-> In Python, the **str** and **repr** methods are special (or "magic") methods used to define how an object is represented as a string.

str: This method is used to return a user-friendly string representation of the object. It is called by the print() function or str() when you want to display the object in a readable format.

repr: This method is used to return an official string representation of the object that is mainly intended for developers. It should ideally return a string that can be used to recreate the object using eval().

Example:

[ ]

```
class Book:

  def __init__(self, title, author):

    self.title = title

    self.author = author



  def __str__(self):

    return f"'{self.title}' by {self.author}"



  def __repr__(self):

    return f"Book('{self.title}', '{self.author}')"
```

```
b = Book("1984", "George Orwell")

print(str(b))

print(repr(b))
```

```
'1984' by George Orwell
Book('1984', 'George Orwell')
```

QUESTION20: What is the significance of the ‘super()’ function in Python?

->The super() function in Python is used to call a method from the parent (or superclass) in a class that inherits from it. It is especially useful in inheritance when you want to extend or modify the behavior of the parent class without completely overriding it.

Using super() helps avoid repeating code and ensures that the parent class is initialized properly, especially in cases of multiple inheritance.

Example:

[ ]

```
class Parent:

    def __init__(self):

        print("Parent class initialized")



class Child(Parent):

    def __init__(self):

        super().__init__()

        print("Child class initialized")
```

```
c = Child()
```

```
Parent class initialized
Child class initialized
```

QUESTION21: What is the significance of the **del** method in Python?

-> The **del** method in Python is a special (magic) method called a destructor. It is automatically invoked when an object is about to be destroyed, typically when it is no longer in use or when the program ends. The main purpose of **del** is to perform cleanup operations, such as closing files, releasing memory, or disconnecting from a database before the object is deleted from memory.

Example

[ ]

```
class FileHandler:

    def __init__(self, filename):

        self.file = open(filename, 'w')

        print("File opened.")



    def __del__(self):

        self.file.close()

        print("File closed.")



fh = FileHandler("example.txt")
```

```
del fh
```

```
File opened.
File closed.
```

QUESTION22: What is the difference between @staticmethod and @classmethod in Python?

-> In Python, @staticmethod and @classmethod are decorators used to define special types of methods inside a class. A @staticmethod is a method that does not take any special first argument like self or cls. It behaves like a regular function but belongs to the class's namespace. It cannot access or modify class or instance variables. In contrast, a @classmethod takes cls as its first parameter, which refers to the class itself. This allows it to access and modify class-level data. Use @staticmethod for utility functions, and use @classmethod when you need to interact with the class rather than a specific instance.

QUESTION23: How does polymorphism work in Python with inheritance?

->Polymorphism in Python allows objects of different classes to be treated as if they are objects of the same class through a common interface. When combined with inheritance, polymorphism enables a child class to provide a specific implementation of a method that is already defined in its parent class. This is called method overriding.

The same method name can perform different behaviors depending on the object calling it. This allows for flexible and reusable code, especially when working with collections of objects from different subclasses.

Example:

```
[ ]
```

```python
class Animal:

    def sound(self):

        return "Some sound"



class Dog(Animal):

    def sound(self):

        return "Bark"



class Cat(Animal):

    def sound(self):

        return "Meow"



# Polymorphism in action

animals = [Dog(), Cat()]



for animal in animals:

    print(animal.sound())
```

```
Bark
Meow
```

QUESTION24: What is method chaining in Python OOP?

-> Method chaining in Python is a technique where multiple methods are called on the same object in a single line of code. Each method call returns the object itself, allowing you to chain subsequent method calls together. This can make code more concise and readable, especially when performing multiple operations on the same object.

How it Works: For method chaining to work, each method must return the object (self) after performing its operation.

Example:

[ ]

```python
class Calculator:
    def __init__(self):
        self.result = 0

    def add(self, value):
        self.result += value
        return self

    def subtract(self, value):
        self.result -= value
        return self

    def multiply(self, value):
        self.result *= value
        return self
    def display(self):
        print("Result:", self.result)
        return self


# Method chaining
calc = Calculator()
calc.add(5).subtract(2).multiply(3).display()
```

```
Result: 9

<__main__.Calculator at 0x7f2a28cc5dd0>
```

QUESTION25: What is the purpose of the **call** method in Python?

-> The **call** method in Python is a special method that allows an instance of a class to be called like a function. In other words, it enables an object to be invoked as if it were a function. This method is used to customize the behavior of an object when it is called directly.

By defining **call**, you can make objects behave like callable functions, which is useful in scenarios where you want an object to perform specific actions or calculations when invoked, without having to explicitly call a method.

Example:

```
[ ]
```

```
class Adder:

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

    def __call__(self, number):
        return self.value + number

# Creating an object
add_five = Adder(5)

# Calling the object as a function
result = add_five(10)  # This calls __call__ method
print(result)
```

```
15
```

☐ PRACTICAL QUESTION

```
""" QUESTION1: Create a parent class Animal with a method speak() that prints a generic message. Create a child class Dog
that overrides the speak() method to print "Bark!"."""

 # Parent class
class Animal:
    def speak(self):
        print("Animal speaks")


# Child class

class Dog(Animal):

    def speak(self):

        print("Bark!")


# Creating an object of Dog class

dog = Dog()

dog.speak()
```

```
Bark!
```

```
""" QUESTION2: Write a program to create an abstract class Shape with a method area().
Derive classes Circle and Rectangle

from it and implement the area() method in both."""


from abc import ABC, abstractmethod

import math


# Abstract class Shape

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


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


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


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


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


# Creating objects of Circle and Rectangle
circle = Circle(5)
```

```python
rectangle = Rectangle(4, 6)


# Printing areas

print("Area of Circle:", circle.area())

print("Area of Rectangle:", rectangle.area())
```

```
Area of Circle: 78.53981633974483
Area of Rectangle: 24
```

```python
"""QUESTION3:  Implement a multi-level inheritance scenario where a class Vehicle has an attribute type. Derive a class Car

and further derive a class ElectricCar that adds a battery attribute."""


# Base class

class Vehicle:

    def __init__(self, vehicle_type):

        self.type = vehicle_type


# Derived class from Vehicle

class Car(Vehicle):

    def __init__(self, vehicle_type, brand):

        super().__init__(vehicle_type)

        self.brand = brand
```

```python
# Derived class from Car (multi-level inheritance)

class ElectricCar(Car):

  def __init__(self, vehicle_type, brand, battery):

      super().__init__(vehicle_type, brand)

      self.battery = battery



  def display_info(self):

      print(f"Type: {self.type}")

      print(f"Brand: {self.brand}")

      print(f"Battery: {self.battery} kWh")



# Creating an object of ElectricCar

tesla = ElectricCar("Car", "Tesla", 75)

tesla.display_info()
```

```
Type: Car
Brand: Tesla
Battery: 75 kWh
```

```python
"""QUESTION4:Demonstrate polymorphism by creating a base class Bird with a method fly().
Create two derived classes

Sparrow and Penguin that override the fly() method."""
```

```python
class Bird:

    def fly(self):

        print("Birds can fly")


class Sparrow(Bird):

    def fly(self):

        print("Sparrow can fly high")


class Penguin(Bird):

    def fly(self):

        print("Penguins cannot fly")


def show_flight(bird):

    bird.fly()


sparrow = Sparrow()

penguin = Penguin()


show_flight(sparrow)

show_flight(penguin)
```

```
Sparrow can fly high
Penguins cannot fly
```

```
"""QUESTION5:  Write a program to demonstrate encapsulation by creating a class
BankAccount with private attributes

balance and methods to deposit, withdraw, and check balance."""



class BankAccount:

    def __init__(self, initial_balance=0):

        self.__balance = initial_balance  # Private attribute



    def deposit(self, amount):

        if amount > 0:

            self.__balance += amount

            print(f"Deposited: ${amount}")

        else:

            print("Deposit amount must be positive.")



    def withdraw(self, amount):

        if 0 < amount <= self.__balance:

            self.__balance -= amount

            print(f"Withdrew: ${amount}")
        else:

            print("Insufficient balance or invalid amount.")



    def check_balance(self):

        print(f"Current Balance: ${self.__balance}")



# Using the class

account = BankAccount(100)

account.deposit(50)

account.withdraw(30)

account.check_balance()
```

```
Deposited: $50
Withdrew: $30
Current Balance: $120
```

```
"""QUESTION6:  Demonstrate runtime polymorphism using a method play() in a base class Instrument. Derive classes Guitar

and Piano that implement their own version of play()."""



class Instrument:

    def play(self):

        print("Playing an instrument")
```

```python
class Guitar(Instrument):

    def play(self):

        print("Strumming the guitar")



class Piano(Instrument):

    def play(self):

        print("Playing the piano")



def perform(instrument):

    instrument.play()



guitar = Guitar()

piano = Piano()



perform(guitar)

perform(piano)
```

```
Strumming the guitar
Playing the piano
```

```
"""QUESTION7:  Create a class MathOperations with a class method add_numbers() to add two
numbers and a static

method subtract_numbers() to subtract two numbers."""


class MathOperations:

   @classmethod

   def add_numbers(cls, a, b):

       return a + b


   @staticmethod

   def subtract_numbers(a, b):

       return a - b


# Using class method

sum_result = MathOperations.add_numbers(10, 5)

print("Sum:", sum_result)


# Using static method

difference = MathOperations.subtract_numbers(10, 5)

print("Difference:", difference)
```

```
Sum: 15
Difference: 5
```

```
"""QUESTION8:  Implement a class Person with a class method to count
the total number of persons created."""

class Person:

    count = 0

    def __init__(self, name):

        self.name = name

        Person.count += 1

    @classmethod

    def total_persons(cls):

        print(f"Total Persons Created: {cls.count}")

# Creating Person objects

p1 = Person("Alice")

p2 = Person("Bob")

p3 = Person("Charlie")

Person.total_persons()
```

```
Total Persons Created: 3
```

```
"""QUESTION9:  Write a class Fraction with attributes numerator and
denominator. Override the str method to display the

fraction as "numerator/denominator"."""

class Fraction:

    def __init__(self, numerator, denominator):

        self.numerator = numerator

        self.denominator = denominator

    def __str__(self):
        return f"{self.numerator}/{self.denominator}"

# Example usage

fraction = Fraction(3, 4)

print(fraction)
```

```
3/4
```

```
"""QUESTION10:  Demonstrate operator overloading by creating a class Vector and overriding the add method to add two

vectors."""



class Vector:

    def __init__(self, x, y):

        self.x = x

        self.y = y



    def __add__(self, other):

        # Adding corresponding components of two vectors

        return Vector(self.x + other.x, self.y + other.y)



    def __str__(self):

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



# Example usage

vector1 = Vector(1, 2)
```

```
vector2 = Vector(3, 4)


# Adding two vectors

result = vector1 + vector2

print(result)
```

```
(4, 6)
```

```
""" QUESTION11:  Create a class Person with attributes name and age. Add a method greet()
that prints "Hello, my name is

{name} and I am {age} years old."""


class Person:

    def __init__(self, name, age):

        self.name = name

        self.age = age


    def greet(self):

        print(f"Hello, my name is {self.name} and I am {self.age} years old.")


# Example usage

person = Person("Alice", 30)

person.greet()
```

```
Hello, my name is Alice and I am 30 years old.
```

```
"""QUESTION12:  Implement a class Student with attributes name and grades. Create a method average_grade() to compute

the average of the grades."""



class Student:

    def __init__(self, name, grades): # Changed init to __init__

        self.name = name

        self.grades = grades



    def average_grade(self):

        if not self.grades:

            return 0

        return sum(self.grades) / len(self.grades)




student1 = Student("Alice", [85, 90, 78])

student2 = Student("Bob", [70, 88, 92, 76])



print(f"{student1.name}'s average grade: {student1.average_grade():.2f}")

print(f"{student2.name}'s average grade: {student2.average_grade():.2f}")
```

```
Alice's average grade: 84.33
```

```
Bob's average grade: 81.50
```

```
""" QUESTION13: Create a class Rectangle with methods set_dimensions() to set the
dimensions and area() to calculate the area."""


class Rectangle:

    def __init__(self):

        self.length = 0

        self.width = 0


    def set_dimensions(self, length, width):

        self.length = length

        self.width = width


    def area(self):

        return self.length * self.width


rectangle = Rectangle()

rectangle.set_dimensions(5, 3)

print(f"Area of rectangle: {rectangle.area()}")
```

```
Area of rectangle: 15
```

```python
"""QUESTION14:  Create a class Employee with a method calculate_salary() that computes the salary based on hours worked

and hourly rate. Create a derived class Manager that adds a bonus to the salary."""


class Employee:

    def __init__(self, name, hours_worked, hourly_rate):

        self.name = name

        self.hours_worked = hours_worked

        self.hourly_rate = hourly_rate


    def calculate_salary(self):

        return self.hours_worked * self.hourly_rate


class Manager(Employee):

    def __init__(self, name, hours_worked, hourly_rate, bonus):

        super().__init__(name, hours_worked, hourly_rate)

        self.bonus = bonus


    def calculate_salary(self):

        return super().calculate_salary() + self.bonus


# Example usage

employee = Employee("John", 40, 20)

manager = Manager("Alice", 40, 25, 500)
```

```
print(f"Employee salary: {employee.calculate_salary()}")

print(f"Manager salary: {manager.calculate_salary()}")
```

```
Employee salary: 800
Manager salary: 1500
```

```
"""QUESTION15: . Create a class Product with attributes name, price, and quantity. Implement a method total_price() that

calculates the total price of the product."""



class Product:

    def __init__(self, name, price, quantity):

        self.name = name

        self.price = price

        self.quantity = quantity



    def total_price(self):

        return self.price * self.quantity



# Example usage

product = Product("Laptop", 1000, 3)
```

```
print(f"Total price of {product.name}: ${product.total_price()}")
```

```
Total price of Laptop: $3000
```

```
"""QUESTION16:  Create a class Animal with an abstract method sound(). Create two derived classes Cow and Sheep that

implement the sound() method."""




from abc import ABC, abstractmethod



class Animal(ABC):

    @abstractmethod

    def sound(self):

        pass



class Cow(Animal):

    def sound(self):

        return "Moo"



class Sheep(Animal):
    def sound(self):

        return "Baa"



# Example usage

cow = Cow()

sheep = Sheep()



print(f"Cow sound: {cow.sound()}")

print(f"Sheep sound: {sheep.sound()}")
```

```
Cow sound: Moo
Sheep sound: Baa
```

```
"""QUESTION17: Create a class Book with attributes title, author, and year_published. Add a method get_book_info() that

returns a formatted string with the book's details."""



class Book:

    def __init__(self, title, author, year_published):

        self.title = title

        self.author = author

        self.year_published = year_published



    def get_book_info(self):
        return f"Title: {self.title}\nAuthor: {self.author}\nYear Published: {self.year_published}"



# Example usage

book = Book("The Great Gatsby", "F. Scott Fitzgerald", 1925)

print(book.get_book_info())
```

```
Title: The Great Gatsby
Author: F. Scott Fitzgerald
Year Published: 1925
```

```
"""QUESTION18:  Create a class House with attributes address and price. Create a derived
class Mansion that adds an

attribute number_of_rooms."""



class House:

    def __init__(self, address, price):

        self.address = address

        self.price = price



class Mansion(House):

    def __init__(self, address, price, number_of_rooms):

        super().__init__(address, price)

        self.number_of_rooms = number_of_rooms



# Example usage

house = House("123 Main St", 250000)
```

```
mansion = Mansion("456 Luxury Blvd", 5000000, 12)



print(f"House: Address - {house.address}, Price - ${house.price}")

print(f"Mansion: Address - {mansion.address}, Price - ${mansion.price}, Rooms - {mansion.number_of_rooms}")
```

```
House: Address - 123 Main St, Price - $250000
Mansion: Address - 456 Luxury Blvd, Price - $5000000, Rooms - 12
```