OOPS Assignment

1. What are the five key concepts of Object-Oriented Programming (OOP)?

A. The five key concepts of Object-Oriented Programming (OOP) are:

1. **Encapsulation**: This refers to bundling the data (attributes) and the methods (functions) that operate on the data into a single unit or class, and restricting access to some of the object's components to protect the integrity of the data. It hides the internal state and allows controlled access through public methods.
2. **Abstraction**: Abstraction involves simplifying complex systems by modeling classes based on essential characteristics, without including unnecessary details. It allows focusing on what an object does rather than how it does it.
3. **Inheritance**: This allows a new class (child or subclass) to inherit properties and behaviour's (methods) from an existing class (parent or superclass). Inheritance promotes code reuse and establishes a relationship between classes.
4. **Polymorphism**: Polymorphism allows objects of different classes to be treated as objects of a common superclass. It means "many shapes" and allows methods to do different things based on the object or data type calling them, enabling flexibility in method use.
5. **Composition**: Composition is a concept where a class is composed of one or more objects from other classes. It models a "has-a" relationship, meaning that one object is a part or component of another, promoting modular and maintainable designs

2. Write a Python class for a `Car` with attributes for `make`, `model`, and `year`. Include a method to display the car's information.

A.
```
class Car:

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

    self.make = make

    self.model = model

    self.year = year
    def display_info(self):
        print(f"Car Information: {self.year} {self.make} {self.model}")

# Example usage
my_car = Car("Toyota", "Corolla", 2020)
my_car.display_info()
```

3. Explain the difference between instance methods and class methods. Provide an example of each.

A. **1. Instance Methods:**

- **Definition**: Instance methods are defined within a class and operate on instances of the class (objects). They can access and modify instance attributes (data specific to each object).
- **Access**: They automatically receive the instance (`self`) as their first argument.
- **Usage**: Used for behaviors that are related to individual instances of the class.

Examples:

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

    def display_info(self):  # Instance method
        print(f"Car Information: {self.year} {self.make} {self.model}")

# Creating an instance
my_car = Car("Toyota", "Corolla", 2020)
my_car.display_info()  # Calls the instance method
```

Class Methods:

- **Definition**: Class methods are defined within a class and operate on the class itself rather than instances of the class. They cannot modify individual instance attributes but can modify class-level data (data shared by all instances of the class).
- **Access**: They automatically receive the class (`cls`) as their first argument.
- **Usage**: Used for behaviors that are related to the class as a whole, such as creating factory methods or modifying class-wide state.
- **Decorator**: Class methods are defined using the `@classmethod` decorator.

Example:

```
      class Car:
  # Class attribute (shared by all instances)
  car_count = 0
  def __init__(self, make, model, year):
    self.make = make
    self.model = model
    self.year = year
    Car.car_count += 1  # Update the class attribute when a new car is created


  @classmethod
  def total_cars(cls):  # Class method
    print(f"Total number of cars: {cls.car_count}")
```

```
# Creating instances (objects) of the Car class

car1 = Car("Toyota", "Corolla", 2020)

car2 = Car("Honda", "Civic", 2021)

# Calling the class method

Car.total_cars()  # Outputs: Total number of cars: 2
```

4. How does Python implement method overloading? Give an example.

A. Python does not support method overloading in the same way as languages like Java or C++. In those languages, you can define multiple methods with the same name but different parameter types or numbers of parameters. However, in Python, method overloading is not directly supported because a method with the same name will simply override any previous definitions.

Instead, Python allows a similar behavior through techniques such as:

1. **Using default parameters**: You can define default values for parameters to allow the same method to handle different numbers of arguments.
2. **Using `*args` and `**kwargs`**: These special arguments can capture a variable number of positional or keyword arguments.
3. **Manually checking types**: Inside the method, you can use type checks (with `isinstance()` or similar) to simulate overloading.

Example:

```
class MathOperations:

  def add(self, a, b=0):  # Second parameter has a default value

    return a + b

# Usage

math_op = MathOperations()

print(math_op.add(5))      # Outputs: 5 (b defaults to 0)
print(math_op.add(5, 10))   # Outputs: 15 (b is passed as 10)
```

5. What are the three types of access modifiers in Python? How are they denoted?

A. In Python, access modifiers control the visibility and accessibility of class attributes and methods. While Python does not enforce strict access control like some other languages (e.g., `public`, `private`, `protected` in C++ or Java), it provides a way to indicate the intended level of access using naming conventions. The three types of access modifiers in Python

1. Public:

- **Denoted by**: No special notation (default).
- **Description**: Public members (attributes or methods) are accessible from anywhere, both inside and outside the class. All members of a class are public by default unless otherwise specified

Example:

```
class Car:
    def __init__(self, make, model):
        self.make = make  # Public attribute
        self.model = model  # Public attribute

    def display_info(self):  # Public method
        print(f"{self.make} {self.model}")

car = Car("Toyota", "Corolla")
print(car.make)  # Accessible outside the class
car.display_info()  # Accessible outside the class
```

2. Protected:

- **Denoted by**: A single underscore prefix (_).
- **Description**: Protected members are intended to be accessible only within the class and its subclasses. While Python doesn't prevent external access to these

members, the single underscore signals to programmers that they should not be accessed directly outside the class or subclass.

Example:

```
class Car:
  def __init__(self, make, model):
    self._make = make  # Protected attribute
    self._model = model  # Protected attribute

  def _display_info(self):  # Protected method
    print(f"{self._make} {_self.model}")

class ElectricCar(Car):
  def display_make(self):
    print(self._make)  # Accessible within subclass

car = Car("Toyota", "Corolla")
print(car._make)  # Technically accessible, but discouraged
```

3. Private:

- **Denoted by**: A double underscore prefix (__).
- **Description**: Private members are intended to be accessible only within the class that defines them. Python performs *name mangling*, which makes it harder (but not impossible) to access these members from outside the class. Private members are not accessible in subclasses.

Example:

```
class Car:
  def __init__(self, make, model):
    self.__make = make  # Private attribute
    self.__model = model  # Private attribute

  def __display_info(self):  # Private method
```

```
        print(f"{self.__make} {self.__model}")

    def get_info(self):  # Public method that can access private members
        self.__display_info()

car = Car("Toyota", "Corolla")
# print(car.__make)  # This would raise an AttributeError
car.get_info()  # Works because it's accessing the private method internally

#To access private members outside the class, name mangling is used, like this:
print(car._Car__make)  # Not recommended, but possible through name mangling
```

6. Describe the five types of inheritance in Python. Provide a simple example of multiple inheritance.

A. In Python, inheritance allows a class (derived or child class) to inherit attributes and methods from another class (base or parent class). Python supports five types of inheritance:

1. Single Inheritance:

- **Description**: A derived class inherits from one base class. This is the simplest form of inheritance.

Example:

```
python
Copy code
class Animal:
    def speak(self):
        print("Animal is speaking")

class Dog(Animal):  # Single inheritance
```

```
    def bark(self):
        print("Dog is barking")

dog = Dog()
dog.speak()  # Inherits from Animal
dog.bark()
```

2. Multiple Inheritance:

- **Description**: A derived class inherits from more than one base class. The derived class has access to attributes and methods from all parent classes.

Example:

```
python
Copy code
class Animal:
    def speak(self):
        print("Animal is speaking")

class Vehicle:
    def drive(self):
        print("Vehicle is driving")

class AmphibiousVehicle(Animal, Vehicle):  # Multiple inheritance
    def amphibious_mode(self):
        print("Amphibious vehicle in action")

av = AmphibiousVehicle()
av.speak()  # Inherited from Animal
av.drive()  # Inherited from Vehicle
av.amphibious_mode()  # AmphibiousVehicle's own method
```

3. Multilevel Inheritance:

- **Description**: A class is derived from another derived class, forming a chain of inheritance. Each level inherits from the class directly above it.

Example:

```
python
Copy code
class Animal:
    def speak(self):
        print("Animal is speaking")

class Dog(Animal):  # Dog inherits from Animal
    def bark(self):
        print("Dog is barking")

class Bulldog(Dog):  # Bulldog inherits from Dog
    def breed(self):
        print("I am a Bulldog")

bulldog = Bulldog()
bulldog.speak()  # Inherits from Animal
bulldog.bark()   # Inherits from Dog
bulldog.breed()  # Bulldog's own method
```

4. Hierarchical Inheritance:

- **Description**: Multiple derived classes inherit from the same base class. They share the same parent class but can have their own specific methods and attributes.

Example:

```
python
Copy code
class Animal:
    def speak(self):
        print("Animal is speaking")

class Dog(Animal):  # Dog inherits from Animal
    def bark(self):
        print("Dog is barking")

class Cat(Animal):  # Cat inherits from Animal
    def meow(self):
        print("Cat is meowing")

dog = Dog()
cat = Cat()
dog.speak()  # Inherited from Animal
cat.speak()  # Inherited from Animal
```

5. Hybrid Inheritance:

- **Description**: Hybrid inheritance is a combination of two or more types of inheritance, typically a combination of multiple and hierarchical inheritance. This can create a complex inheritance structure.

Example:

```
python
Copy code
class Animal:
    def speak(self):
        print("Animal is speaking")
```

```
class Mammal(Animal):
    def has_fur(self):
        print("Mammal has fur")

class Bird(Animal):
    def has_feathers(self):
        print("Bird has feathers")

class Bat(Mammal, Bird):  # Multiple inheritance
    def fly(self):
        print("Bat is flying")

bat = Bat()
bat.speak()  # Inherited from Animal
bat.has_fur()  # Inherited from Mammal
bat.has_feathers()  # Inherited from Bird
bat.fly()  # Bat's own method
```

Example of Multiple Inheritance:

```
python
Copy code
class Father:
    def height(self):
        print("Height from Father")

class Mother:
    def eyes(self):
        print("Eye color from Mother")

class Child(Father, Mother):  # Inherits from both Father and Mother
    def talent(self):
        print("Talent of the Child")

child = Child()
child.height()  # Inherited from Father
child.eyes()  # Inherited from Mother
child.talent()  # Child's own method
```

7. What is the Method Resolution Order (MRO) in Python? How can you retrieve it programmatically?

A. **Method Resolution Order (MRO) in Python:**

The **Method Resolution Order (MRO)** defines the order in which Python looks for methods and attributes in a class hierarchy when there is inheritance involved, particularly multiple inheritance. MRO determines the sequence in which classes are searched for a called method or attribute. Python uses the **C3 linearization algorithm** (also known as **C3 superclass linearization**) to calculate the MRO.

In essence, the MRO ensures that:

- A method in the child class is prioritized over methods in the parent classes.
- In the case of multiple inheritance, Python follows a specific order to look up methods from parent classes, avoiding ambiguity.

MRO Rules:

1. The child class is searched first.
2. Parent classes are searched in a left-to-right depth-first manner based on the inheritance hierarchy.
3. The same class is never searched twice in the hierarchy.

How to Retrieve MRO Programmatically:

You can retrieve the MRO in Python using two methods:

1. **Using the `__mro__` attribute**: This special attribute stores the MRO of a class.

Example:

```
python
Copy code
class A:
    pass

class B(A):
    pass

class C(A):
    pass

class D(B, C):
    pass

print(D.__mro__) # Output: (<class '__main__.D'>, <class
'__main__.B'>, <class '__main__.C'>, <class '__main__.A'>, <class
'object'>)
```

2. **Using the `mro()` method**: The `mro()` method is a built-in method that returns a list of classes in the order they are checked for method resolution

   Example:

   ```
   class A:
     pass

   class B(A):
     pass

   class C(A):
     pass
   ```

```
    class D(B, C):
        pass

    print(D.mro()) #Output: [<class '__main__.D'>, <class '__main__.B'>, <class
    '__main__.C'>, <class '__main__.A'>, <class 'object'>]
```

8. Create an abstract base class `Shape` with an abstract method `area()`. Then create two subclasses `Circle` and `Rectangle` that implement the `area()` method.

A.

```
from abc import ABC, abstractmethod

import math


# Abstract base class

class Shape(ABC):

    @abstractmethod

    def area(self):

        pass  # Abstract method, must be implemented in subclasses


# Subclass Circle implementing the area method

class Circle(Shape):

    def __init__(self, radius):

        self.radius = radius


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


# Subclass Rectangle implementing the area method
class Rectangle(Shape):
    def __init__(self, width, height):
        self.width = width
        self.height = height


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


# Example usage
circle = Circle(5)  # Circle with radius 5
rectangle = Rectangle(4, 6)  # Rectangle with width 4 and height 6


print(f"Circle area: {circle.area()}")  # Outputs: Circle area: 78.5398...
print(f"Rectangle area: {rectangle.area()}")  # Outputs: Rectangle area: 24
```

9. Demonstrate polymorphism by creating a function that can work with different shape objects to calculate and print their areas.

A. Polymorphism in Python allows us to define methods in different classes with the same name, and the method that gets executed depends on the object (class) that calls it. This means we can create a function that works with different types of objects (in this case, `Circle` and `Rectangle`) and call the `area()` method on them without needing to know the specific type of shape.

Example:

```
from abc import ABC, abstractmethod

import math


# Abstract base class

class Shape(ABC):

    @abstractmethod

    def area(self):

        pass  # Abstract method, must be implemented in subclasses


# Subclass Circle implementing the area method

class Circle(Shape):

    def __init__(self, radius):

        self.radius = radius


    def area(self):

        return math.pi * self.radius ** 2


# Subclass Rectangle implementing the area method

class Rectangle(Shape):

    def __init__(self, width, height):

        self.width = width

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

# Polymorphic function to calculate and print area
def print_area(shape):
    print(f"The area of the {type(shape).__name__} is: {shape.area()}")

# Example usage with different shape objects
circle = Circle(5)  # Circle with radius 5
rectangle = Rectangle(4, 6)  # Rectangle with width 4 and height 6

# Calling the polymorphic function
print_area(circle)  # Outputs: The area of the Circle is: 78.5398...
print_area(rectangle)  # Outputs: The area of the Rectangle is: 24
```

10. Implement encapsulation in a `BankAccount` class with private attributes for `balance` and `account_number`. Include methods for deposit, withdrawal, and balance inquiry.

A.

```
class BankAccount:
    def __init__(self, account_number, initial_balance=0):
        self.__account_number = account_number  # Private attribute
        self.__balance = initial_balance  # Private attribute
```

```
def deposit(self, amount):
    """Deposit money into the account."""
    if amount > 0:
        self.__balance += amount
        print(f"Deposited: ${amount}. New balance: ${self.__balance}")
    else:
        print("Deposit amount must be positive.")

def withdraw(self, amount):
    """Withdraw money from the account."""
    if amount > 0:
        if amount <= self.__balance:
            self.__balance -= amount
            print(f"Withdrew: ${amount}. New balance: ${self.__balance}")
        else:
            print("Insufficient funds for withdrawal.")
    else:
        print("Withdrawal amount must be positive.")

def get_balance(self):
    """Get the current balance of the account."""
    return self.__balance
    def get_account_number(self):
        """Get the account number."""
        return self.__account_number

# Example usage
account = BankAccount("123456789", 1000)  # Create a BankAccount with initial
balance of $1000

# Deposit money
account.deposit(500)  # Outputs: Deposited: $500. New balance: $1500

# Withdraw money
account.withdraw(200)  # Outputs: Withdrew: $200. New balance: $1300

# Check balance
print(f"Current balance: ${account.get_balance()}")  # Outputs: Current balance: $1300

# Check account number
print(f"Account number: {account.get_account_number()}")  # Outputs: Account
number: 123456789
```

11. Write a class that overrides the `__str__` and `__add__` magic methods. What will these methods allow you to do?

A.

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

    def __str__(self):
        """Return a string representation of the Point."""
        return f"Point({self.x}, {self.y})"

    def __add__(self, other):
        """Return a new Point that is the sum of this Point and another Point."""
        if isinstance(other, Point):
            return Point(self.x + other.x, self.y + other.y)
        return NotImplemented  # If 'other' is not a Point, return NotImplemented

# Example usage
point1 = Point(2, 3)
point2 = Point(4, 5)

# Using the __str__ method
print(point1)  # Outputs: Point(2, 3)
```

```
# Using the __add__ method
point3 = point1 + point2  # Adds point1 and point2
print(point3)  # Outputs: Point(6, 8)
```

12. Create a decorator that measures and prints the execution time of a function.

A.

```
import time
from functools import wraps


def timing_decorator(func):
    """Decorator that measures the execution time of a function."""
    @wraps(func)  # Preserves the metadata of the original function
    def wrapper(*args, **kwargs):
        start_time = time.time()  # Record the start time
        result = func(*args, **kwargs)  # Call the original function
        end_time = time.time()  # Record the end time
        execution_time = end_time - start_time  # Calculate the execution time
        print(f"Execution time of {func.__name__}: {execution_time:.4f} seconds")
        return result  # Return the result of the original function
    return wrapper


# Example usage of the decorator
@timing_decorator
```

```
def example_function(n):
    """Example function that simulates a time-consuming operation."""
    total = 0
    for i in range(n):
        total += i ** 2  # Sum of squares from 0 to n-1
    return total


# Call the example function
result = example_function(1000000)  # Passing a large number to see the timing
print(f"Result: {result}")
```

Example:

```
   A
  / \
 B   C
  \ /
   D
```

In this example:

- Class A is the base class.
- Classes B and C both inherit from A.
- Class D inherits from both B and C.

If both B and C override a method from A, and then D calls that method, it can be unclear which version of the method should be executed—B's or C's. This creates a "diamond" shape in the inheritance diagram, leading to potential ambiguity.

How Python Resolves the Diamond Problem

Python uses the **C3 linearization algorithm** (also known as **C3 superclass linearization**) to resolve the diamond problem and determine the method resolution order (MRO). This algorithm ensures a consistent order in which classes are searched when looking for a method or attribute.

Key Rules of C3 Linearization:

1. The method resolution order of a class is determined by the order of the classes in its inheritance declaration.
2. The order is based on the following criteria:
    a. A class is searched before its parents.
    b. The parents are searched from left to right, but only if they have not already been included in the search.
    c. If a class is inherited from multiple classes, the MRO ensures that the method from the closest class in the hierarchy is chosen.

Example:

```
class A:

  def method(self):

    print("Method from class A")



class B(A):

  def method(self):

    print("Method from class B")



class C(A):

  def method(self):
        print("Method from class C")


class D(B, C):
    pass


# Create an instance of D and call the method
d = D()
d.method()  # Outputs: Method from class B


# Checking the method resolution order (MRO)
print(D.__mro__)  # Outputs the MRO: (<class '__main__.D'>, <class '__main__.B'>,
<class '__main__.C'>, <class '__main__.A'>, <class 'object'>)
```

13. Explain the concept of the Diamond Problem in multiple inheritance. How does Python resolve it?

A. The **Diamond Problem** is a common issue in object-oriented programming, particularly in languages that support multiple inheritance. It arises when a class inherits from two classes that both inherit from a common base class. This situation can create ambiguity in method resolution, particularly when the method or attribute is defined in the base class.

14. Write a class method that keeps track of the number of instances created from a class.

A. class InstanceCounter:

```
    # Class variable to keep track of the number of instances
    instance_count = 0
```

```
  def __init__(self):
    # Increment the instance count whenever a new instance is created
    InstanceCounter.instance_count += 1

  @classmethod
  def get_instance_count(cls):
    """Return the current instance count."""
    return cls.instance_count

# Example usage
obj1 = InstanceCounter()  # First instance
obj2 = InstanceCounter()  # Second instance
obj3 = InstanceCounter()  # Third instance

# Get the number of instances created
print(f"Number of instances created: {InstanceCounter.get_instance_count()}")  # Outputs: 3
```

15. Implement a static method in a class that checks if a given year is a leap year.

```
class YearUtils:
  @staticmethod
  def is_leap_year(year):
    """Check if the given year is a leap year."""
    if (year % 4 == 0 and year % 100 != 0) or (year % 400 == 0):
      return True
```

```
        return False


# Example usage

year = 2024

if YearUtils.is_leap_year(year):

    print(f"{year} is a leap year.")

else:

    print(f"{year} is not a leap year.")

year = 1900

if YearUtils.is_leap_year(year):

    print(f"{year} is a leap year.")

else:

year = 2000

if YearUtils.is_leap_year(year):

    print(f"{year} is a leap year.")

else:

    print(f"{year} is not a leap year.")
```