OOPS-Assignment Questions

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

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

1. **Encapsulation:**
    - Bundling data (attributes) and methods (functions) that operate on that data within a single unit called a class.
    - This helps in hiding implementation details and protecting data integrity.
2. **Inheritance:**
    - Creating new classes (child classes) based on existing classes (parent classes).
    - Child classes inherit attributes and methods from their parent classes, [1] promoting code reusability and creating hierarchical relationships.
3. **Polymorphism:**
    - The ability of objects to take on many forms.
    - It allows objects of different classes to be treated as if they were objects of a common superclass.
    - This enables flexible and dynamic code.
4. **Abstraction:**
    - Focusing on the essential features of an object while hiding unnecessary details.
    - It simplifies complex systems by representing them in a more abstract way.

5. **Objects and Classes:**
    - **Objects:** Instances of a class. They have their own state (attributes) and behavior (methods).
    - **Classes:** Blueprints or templates for creating objects. They define the structure and behavior of objects.

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

Ans :- python

```
class Car:
    def __init__(self, make, model, year):
        """Initialize the attributes of the car."""
        self.make = make
        self.model = model
        self.year = year

    def display_info(self):
        """Display the car's information."""
        info = f"Car Make: {self.make}\nCar Model: {self.model}\nYear: {self.year}"
        print(info)
```

 # Example usage:

if __name__ == "__main__":

my_car = Car("Toyota", "Camry", 2020)

my_car.display_info()

Explanation

Class Definition

- **__init__ Method**: This is the constructor that initializes the attributes make, model, and year when a new instance of Car is created.
- **display_info Method**: This method formats and prints the car's information in a readable manner.

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

Ans :- Instance methods and class methods in Python serve different purposes and have distinct characteristics. Here's a detailed explanation of each, along with examples.

Instance Methods

Definition: Instance methods are functions defined within a class that operate on instances of that class. They take self as their first parameter, which refers to the specific instance of the class.

Characteristics:

- Access instance variables and methods.
- Can modify object state.
- Must be called on an instance of the class.

Example:

python

class Car:

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

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

# Creating an instance and calling the instance method
my_car = Car("Toyota", "Corolla")
my_car.display_info()  # Output: Car Make: Toyota, Model: Corolla
```

Class Methods

Definition: Class methods are functions defined within a class that operate on the class itself rather than instances. They take cls as their first parameter, which refers to the class.

Characteristics:

- Can access and modify class variables.
- Decorated with @classmethod.
- Can be called on the class itself or on instances.

Example:

python

```
class Car:
    number_of_wheels = 4  # Class variable

    @classmethod
    def display_wheels(cls):
        print(f"A car typically has {cls.number_of_wheels} wheels.")
```

 # Calling the class method

Car.display_wheels() *# Output: A car typically has 4 wheels.*

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

Ans:- Python does not support traditional method overloading as seen in languages like Java or C++. However, it allows you to achieve similar functionality through various techniques, such as using default parameters, variable-length arguments, or third-party libraries.

Method Overloading in Python

Default Parameters and Variable-Length Arguments

In Python, you can simulate method overloading by defining a single method that can accept different numbers of arguments. This can be done using default parameter values or by utilizing the *args syntax for variable-length arguments.

Example Using Default Parameters

Here's an example demonstrating method overloading using default parameters:

python

```
class Calculator:
    def add(self, a=None, b=None, c=None):
        if a is not None and b is not None and c is not None:
            return a + b + c  # Three arguments
        elif a is not None and b is not None:
            return a + b      # Two arguments
        elif a is not None:
```

```
        return a        # One argument
    return 0            # No arguments

# Example usage
calc = Calculator()
print(calc.add(10, 20, 30))  # Output: 60
print(calc.add(10, 20))      # Output: 30
print(calc.add(10))          # Output: 10
print(calc.add())            # Output: 0
```

Example Using Variable-Length Arguments

You can also use the *args syntax to handle an arbitrary number of arguments:

python

```
class Calculator:
    def add(self, *args):
        return sum(args)  # Sums all provided arguments

# Example usage
calc = Calculator()
print(calc.add(10, 20, 30))  # Output: 60
print(calc.add(10, 20))      # Output: 30
print(calc.add(10))          # Output: 10
print(calc.add())            # Output: 0
```

Third-Party Libraries for Method Overloading

For more complex scenarios, you can use the multipledispatch library to implement method overloading based on argument types. Here's an example:

python

```
from multipledispatch import dispatch


class Calculator:
    @dispatch(int, int)
    def add(self, a, b):
        return a + b

    @dispatch(int, int, int)
    def add(self, a, b, c):
        return a + b + c

# Example usage
calc = Calculator()
print(calc.add(10, 20))       # Output: 30
print(calc.add(10, 20, 30))   # Output: 60
```

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

Ans :- Python implements three types of access modifiers to control the visibility of class members (attributes and methods). These modifiers are public, protected, and private, and they are denoted using underscores.

Types of Access Modifiers

1. Public Access Modifier

- **Definition: Members declared as public can be accessed from anywhere in the program, both inside and outside the class.**
- **Denotation: By default, all class members are public; no special notation is needed.**

Example:

python

```
class Employee:
    def __init__(self, name, salary):
        self.name = name  # public attribute
        self.salary = salary  # public attribute

emp = Employee("Alice", 50000)
print(emp.name)  # Accessible from outside the class
```

2. Protected Access Modifier

- **Definition: Members declared as protected can only be accessed within the class and by subclasses (derived classes). This modifier is a way to indicate that these members should not be accessed directly from outside the class.**
- **Denotation: A single underscore _ is prefixed to the member name.**

Example:

python

```
class Employee:
    def __init__(self, name, salary):
        self._name = name  # protected attribute
        self._salary = salary  # protected attribute
```

```
class Manager(Employee):
    def display(self):
        print(self._name)  # Accessible in subclass

mgr = Manager("Bob", 70000)
mgr.display()  # Output: Bob
```

3. Private Access Modifier

- **Definition: Members declared as private can only be accessed within the class itself. They are not accessible from outside the class or by subclasses.**
- **Denotation: A double underscore __ is prefixed to the member name. Python uses a technique called name mangling to prevent access to these members.**

Example:

python

```
class Employee:
    def __init__(self, name, salary):
        self.__name = name  # private attribute
        self.__salary = salary  # private attribute

    def display(self):
        print(self.__name)  # Accessible within the class

emp = Employee("Charlie", 60000)
emp.display()  # Output: Charlie
```

 # print(emp.__name) # Raises AttributeError: 'Employee' object has no attribute '__name'

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

Ans :- In Python, inheritance is a fundamental concept in object-oriented programming that allows a class (child class) to inherit attributes and methods from another class (parent class). There are five main types of inheritance in Python:

Types of Inheritance

1. Single Inheritance

In single inheritance, a child class inherits from one parent class. This type promotes code reusability and allows for the extension of existing functionalities.Example:

python

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


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


# Usage
child_instance = Child()
child_instance.display()  # Output: This is the parent class.
```

2. Multiple Inheritance

Multiple inheritance occurs when a child class inherits from more than one parent class. This allows the child class to access methods and attributes from multiple classes.Example:

python

```
class Mother:
    def mother_name(self):
        return "Mother's Name"


class Father:
    def father_name(self):
        return "Father's Name"


class Child(Mother, Father):
    def display_parents(self):
        print(self.mother_name())
        print(self.father_name())


# Usage
child_instance = Child()
child_instance.display_parents()
# Output:
# Mother's Name
# Father's Name
```

3. Multilevel Inheritance

In multilevel inheritance, a class derives from another derived class, forming a chain of inheritance. This can be visualized as a hierarchy

where a grandparent class passes attributes to a parent class, which in turn passes them to a child class.Example:

python

```
class Grandparent:
    def display_grandparent(self):
        print("This is the grandparent.")


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


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


# Usage
child_instance = Child()
child_instance.display_grandparent()  # Output: This is the grandparent.
```

4. Hierarchical Inheritance

Hierarchical inheritance occurs when multiple child classes inherit from a single parent class. This structure allows different classes to share common functionality defined in the parent.Example:

python

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

```
class Child1(Parent):
    def show_child1(self):
        print("This is child 1.")

class Child2(Parent):
    def show_child2(self):
        print("This is child 2.")

# Usage
child1_instance = Child1()
child1_instance.display()  # Output: This is the parent class.
child2_instance = Child2()
child2_instance.display()  # Output: This is the parent class.
```

5. Hybrid Inheritance

Hybrid inheritance combines two or more types of inheritance. It can involve multiple and hierarchical inheritance, allowing for complex relationships among classes.Example:

python

```
class Base:
    def base_method(self):
        print("This is the base class.")

class Derived1(Base):
    def derived1_method(self):
        print("This is derived class 1.")
```

```
class Derived2(Base):
    def derived2_method(self):
        print("This is derived class 2.")

class MultiDerived(Derived1, Derived2):
    def multi_derived_method(self):
        print("This is a multi-derived class.")

# Usage
multi_derived_instance = MultiDerived()
multi_derived_instance.base_method()  # Output: This is the base class.
```

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

Ans :- Method Resolution Order (MRO) in Python is a crucial concept that defines the order in which base classes are searched when executing a method. This is particularly significant in the context of multiple inheritance, where a class can inherit from multiple parent classes. The MRO determines how Python resolves method calls and ensures that the correct method is executed.

Understanding Method Resolution Order (MRO)

1. **Definition: MRO is the sequence of classes that Python checks when searching for a method. It follows a specific algorithm to ensure a consistent and predictable order, which is especially important in complex inheritance hierarchies.**
2. **C3 Linearization: Python uses the C3 linearization algorithm to compute the MRO. This algorithm combines depth-first search**

with certain rules to avoid ambiguity and maintain a consistent order. It ensures that:

- **The order of base classes is preserved.**
- **A class cannot appear before its base classes in the MRO.**

3. **Example:**

```
python

class A:
    def process(self):
        print('A process()')


class B(A):
    def process(self):
        print('B process()')


class C(A):
    def process(self):
        print('C process()')


class D(B, C):
    pass


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

In this example, when obj.process() is called, it first looks in D, then B, and finds the method in B. The MRO for class D shows the order in which Python will search for methods.

Retrieving MRO Programmatically

You can retrieve the MRO of a class using two methods:

1. **Using the mro() method:**

python

print(D.mro())

2. **Using the __mro__ attribute:**

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

Ans :- To create an abstract base class in Python, you can use the abc module, which provides infrastructure for defining Abstract Base Classes (ABCs). Below is an example of how to define an abstract base class Shape with an abstract method area(), and then implement two subclasses, Circle and Rectangle, that provide their own implementations of the area() method.

Implementation

Step 1: Define the Abstract Base Class

python

```
from abc import ABC, abstractmethod
import math

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

```
        pass
```

Step 2: Implement Subclasses

Circle Class

python

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

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

Rectangle Class

python

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

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

Step 3: Example Usage

Now that we have defined our classes, let's see how to use them:

python

```
# Create instances of Circle and Rectangle
circle = Circle(radius=5)
rectangle = Rectangle(width=4, height=6)
```

```
# Calculate and print areas
print(f"Area of Circle: {circle.area():.2f}")        # Output: Area of Circle: 78.54
print(f"Area of Rectangle: {rectangle.area():.2f}")  # Output: Area of Rectangle: 24.00
```

Complete Code

Here is the complete code for your reference:

```
python
from abc import ABC, abstractmethod
import math

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

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

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

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(radius=5)
rectangle = Rectangle(width=4, height=6)

print(f"Area of Circle: {circle.area():.2f}")        # Output: Area of Circle: 78.54
print(f"Area of Rectangle: {rectangle.area():.2f}")  # Output: Area of Rectangle: 24.00
```

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

Ans :- To demonstrate polymorphism in Python, we can create a function that accepts different shape objects and calculates their areas. This function will work with any object that adheres to the Shape interface, showcasing how polymorphism allows for flexibility in code.

Step 1: Define the Abstract Base Class and Subclasses

We'll start by defining the abstract base class Shape with an abstract method area(), and then implement two subclasses: Circle and Rectangle.

python

```
from abc import ABC, abstractmethod
import math
```

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

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

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

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

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

Step 2: Create a Function to Calculate Areas

Next, we will define a function that takes a list of shape objects and prints their areas.

python

```
def print_areas(shapes):
```

for shape in shapes:

print(f"The area of the {shape.__class__.__name__} is: {shape.area():.2f}")

Step 3: Example Usage

Now let's create instances of Circle and Rectangle, and use the print_areas function to demonstrate polymorphism.

python

 # Create instances of Circle and Rectangle

shapes = [

Circle(radius=5),

Rectangle(width=4, height=6)

]

 # Print areas of the shapes

print_areas(shapes)

Complete Code

Here's the complete code for your reference:

python

from abc import ABC, abstractmethod

import math

class Shape(ABC):

@abstractmethod

def area(self):

pass

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

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

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

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

def print_areas(shapes):
    for shape in shapes:
        print(f"The area of the {shape.__class__.__name__} is:
{shape.area():.2f}")

# Example usage
shapes = [
    Circle(radius=5),
    Rectangle(width=4, height=6)
]
```

print_areas(shapes)

Output

When you run this code, you will see output similar to:

text

The area of the Circle is: 78.54

The area of the Rectangle is: 24.00

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

Ans :- Encapsulation is a fundamental principle of object-oriented programming that restricts direct access to some of an object's attributes and methods. In Python, this is typically achieved using private attributes, which can be indicated by prefixing the attribute names with double underscores (__).Below is an implementation of a BankAccount class that encapsulates
the balance and account_number attributes. It includes methods for depositing money, withdrawing money, and inquiring about the balance.

Implementation of the BankAccount Class

python

```
class BankAccount:
    def __init__(self, account_number, initial_balance=0.0):
        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:.2f}. New balance:
${self.__balance:.2f}")
        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:.2f}. New balance:
${self.__balance:.2f}")
            else:
                print("Insufficient funds for this withdrawal.")
        else:
            print("Withdrawal amount must be positive.")

    def get_balance(self):
        """Return the current balance."""
        return self.__balance

    def get_account_number(self):
        """Return the account number."""
```

```
        return self.__account_number


# Example Usage
if __name__ == "__main__":
    # Create a bank account with an initial balance
    account = BankAccount(account_number="123456789",
initial_balance=1000.00)

    # Deposit money
    account.deposit(250.00)

    # Withdraw money
    account.withdraw(150.00)

    # Check balance
    print(f"Current balance: ${account.get_balance():.2f}")

    # Attempt to withdraw more than available balance
    account.withdraw(1200.00)

    # Check account number (demonstrating encapsulation)
    print(f"Account Number: {account.get_account_number()}")
```

Explanation of the Code

1. **Private Attributes:**

- **The __account_number and __balance attributes are private, meaning they cannot be accessed directly from outside the class.**

2. **Methods:**
    - **deposit(amount): This method allows the user to deposit a specified amount into the bank account. It checks that the deposit amount is positive before updating the balance.**
    - **withdraw(amount): This method allows the user to withdraw a specified amount from the bank account. It checks that the withdrawal amount is positive and does not exceed the current balance.**
    - **get_balance(): This method returns the current balance of the account.**
    - **get_account_number(): This method returns the account number.**
3. **Example Usage:**
    - **An instance of BankAccount is created with an initial balance.**
    - **The methods are called to demonstrate depositing, withdrawing, and checking the balance.**

Output

When you run this code, you will see output similar to:

text

Deposited: $250.00. New balance: $1250.00

Withdrew: $150.00. New balance: $1100.00

Current balance: $1100.00

Insufficient funds for this withdrawal.

Account Number: 123456789

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

Ans :- To implement a class that overrides the __str__ and __add__ magic methods, we can create a simple class that represents a mathematical vector. This class will allow us to represent vectors as strings and perform vector addition using the + operator.

Implementation of the Vector Class

python

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

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

    def __add__(self, other):
        """Add two vectors together."""
        if isinstance(other, Vector):
            return Vector(self.x + other.x, self.y + other.y)
        return NotImplemented

# Example Usage
if __name__ == "__main__":
```

```
v1 = Vector(2, 3)
v2 = Vector(4, 5)

# Print the string representation of the vectors
print(v1)  # Output: Vector(2, 3)
print(v2)  # Output: Vector(4, 5)

# Add two vectors
v3 = v1 + v2
print(v3)  # Output: Vector(6, 8)
```

Explanation of the Code

1. **Class Definition: The Vector class has two attributes, x and y, representing the components of the vector.**
2. **__str__ Method:**
   - **This method is overridden to provide a custom string representation of the vector object.**
   - **When you call print(v1), it will use this method to display the vector in a readable format (e.g., Vector(2, 3)).**
3. **__add__ Method:**
   - **This method is overridden to define how two Vector objects should be added together using the + operator.**
   - **It checks if the other object is also an instance of Vector. If so, it creates and returns a new Vector object with the summed components.**
   - **If the other object is not a Vector, it returns NotImplemented, which allows Python to handle unsupported operations gracefully.**

What These Methods Allow You to Do

- **Custom String Representation: The overridden __str__ method allows you to define how instances of your class are represented as strings. This is particularly useful for debugging and logging purposes.**
- **Operator Overloading: The overridden __add__ method enables you to use the + operator with instances of your class. This makes your class more intuitive and easier to use, as it allows for natural mathematical expressions.**

Example Output

When you run this code, you will see output similar to:

text

Vector(2, 3)

Vector(4, 5)

Vector(6, 8)

This demonstrates that we can easily add two vectors and get a new vector as a result while also having a clear string representation for each vector.

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

Ans :- To create a decorator that measures and prints the execution time of a function in Python, you can use the time module to capture the start and end times around the function call. Below is a complete implementation of such a decorator.

Implementation of the Execution Time Decorator

python

import time

```
from functools import wraps

def measure_execution_time(func):
    """Decorator to measure the execution time of a function."""

    @wraps(func)  # Preserve the original function's metadata
    def wrapper(*args, **kwargs):
        start_time = time.perf_counter()  # Start time
        result = func(*args, **kwargs)    # Call the original function
        end_time = time.perf_counter()    # End time
        execution_time = end_time - start_time  # Calculate execution time
        print(f"Function '{func.__name__}' took {execution_time:.4f}
seconds to execute")
        return result  # Return the result of the original function

    return wrapper

# Example usage of the decorator

@measure_execution_time
def calculate_sum(n):
    """Function to calculate the sum of numbers from 1 to n."""
    return sum(range(1, n + 1))

if __name__ == "__main__":
```

```
result = calculate_sum(1000000)  # Call the decorated function
print("Result:", result)
```

Explanation of the Code

1. **Decorator Definition:**
   - **The measure_execution_time function is defined as a decorator that takes another function func as an argument.**
   - **The @wraps(func) decorator from functools is used to preserve the metadata (like name and docstring) of the original function.**
2. **Wrapper Function:**
   - **Inside measure_execution_time, a nested function wrapper is defined. This function will execute the original function and measure its execution time.**
   - **The start_time is recorded before calling the original function, and end_time is recorded immediately after.**
   - **The execution time is calculated by subtracting start_time from end_time.**
3. **Output:**
   - **The execution time is printed in a formatted string that includes the name of the function being measured.**
   - **Finally, the result of the original function is returned.**
4. **Example Function:**
   - **The example function calculate_sum calculates the sum of numbers from 1 to n*n*. It is decorated with @measure_execution_time, so when it is called, its execution time will be measured and printed.**

Example Output

When you run this code, you might see output similar to:

text

Function 'calculate_sum' took 0.1234 seconds to execute

Result: 500000500000

This output indicates how long it took for the calculate_sum function to execute, along with its resulting value.

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

Ans :- The Diamond Problem is a well-known issue in object-oriented programming, particularly in languages that support multiple inheritance, such as Python. It arises when a class inherits from two classes that both inherit from a common superclass, creating an ambiguous situation regarding which method or attribute should be inherited.

Understanding the Diamond Problem

Structure of the Diamond Problem

Consider the following class hierarchy:

text

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

In this structure:

- **Class A is the common superclass.**
- **Classes B and C both inherit from A.**
- **Class D inherits from both B and C.**

The Ambiguity

The ambiguity occurs when:

- **Both B and C override a method from A.**
- **When an instance of D calls this method, it is unclear whether the method from B or the method from C should be executed. This can lead to unexpected behavior or errors.**

For example, if both B and C provide their own implementation of a method display(), calling display() on an instance of D raises the question: "Which display() method should be executed?"

How Python Resolves the Diamond Problem

Python resolves the Diamond Problem using a mechanism called Method Resolution Order (MRO). The MRO defines the order in which base classes are searched when executing a method. Python uses the C3 linearization algorithm to compute this order, ensuring that:

1. **The order of inheritance is preserved.**
2. **A class appears before its parents in the MRO.**

Using MRO in Python

You can view the MRO of a class using the __mro__ attribute or by calling the mro() method. Here's how it works with an example:

python

```
class A:
    def display(self):
        print("Display from A")


class B(A):
    def display(self):
        print("Display from B")
```

```
class C(A):
    def display(self):
        print("Display from C")

class D(B, C):
    pass

# Create an instance of D
d = D()
d.display()  # Output: Display from B

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

Explanation of Output

- **When d.display() is called, Python looks for the display method in the order defined by MRO: first in D, then in B, then in C, and finally in A. Since B is found first in this order, its display method is executed.**
- **The output of print(D.__mro__) shows the order in which Python will search for methods: D → B → C → A → object.**

Conclusion

The Diamond Problem illustrates a fundamental challenge in multiple inheritance scenarios. However, Python's use of Method Resolution Order (MRO) effectively resolves these ambiguities by providing a clear

and consistent approach to method lookup. This ensures that even with complex inheritance structures, Python can determine which methods to call without confusion.

Share

Rewrite

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

Ans :- To create a class method that keeps track of the number of instances created from a class, you can use a class variable to store the count. Each time a new instance is created, you can increment this count in the __init__ method. Additionally, you can define a class method to retrieve the current count of instances.Here's how you can implement this:

Implementation of the Class with Instance Count

python

```
class InstanceCounter:
    instance_count = 0  # Class variable to keep track of instances

    def __init__(self):
        InstanceCounter.instance_count += 1  # Increment count on instance creation

    @classmethod
    def get_instance_count(cls):
        """Class method to get the current instance count."""
        return cls.instance_count

# Example Usage
```

```
if __name__ == "__main__":
    obj1 = InstanceCounter()
    obj2 = InstanceCounter()
    obj3 = InstanceCounter()

    print(f"Number of instances created: {InstanceCounter.get_instance_count()}")  # Output: 3
```

Explanation of the Code

1. **Class Variable:**
    - **instance_count: This is a class variable that is shared among all instances of the class. It keeps track of how many instances have been created.**
2. **Constructor (__init__ method):**
    - **Each time an instance of InstanceCounter is created, the constructor increments the instance_count by one.**
3. **Class Method (get_instance_count):**
    - **This method is defined using the @classmethod decorator and takes cls as its first parameter (which refers to the class itself).**
    - **It returns the current value of instance_count.**
4. **Example Usage:**
    - **Three instances of InstanceCounter are created.**
    - **The total number of instances created is printed using the class method get_instance_count().**

Output

When you run this code, you will see output similar to:

text

Number of instances created: 3

Summary

This implementation effectively demonstrates how to keep track of the number of instances created from a class using a class variable and a class method. This approach allows you to easily query the total number of instances at any point in your program.

Share

Rewrite

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

Ans :- To implement a static method in a class that checks if a given year is a leap year, you can define the method using the @staticmethod decorator. A leap year is defined by the following rules:

1. **A year is a leap year if it is divisible by 4.**
2. **However, if the year is divisible by 100, it is not a leap year, unless:**
3. **The year is also divisible by 400, in which case it is a leap year.**

Implementation of the Leap Year Checker Class

Here's how you can implement this in Python:

python

```
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
if __name__ == "__main__":
    years_to_check = [2000, 2001, 2004, 1900, 2020, 2100]


    for year in years_to_check:
        if YearUtils.is_leap_year(year):
            print(f"{year} is a leap year.")
        else:
            print(f"{year} is not a leap year.")
```

Explanation of the Code

1. **Class Definition:**
    - **The YearUtils class contains a static method to check for leap years.**
2. **Static Method (is_leap_year):**
    - **This method takes an integer year as an argument.**
    - **It implements the rules for determining whether the specified year is a leap year.**
    - **The method returns True if the year is a leap year and False otherwise.**
3. **Example Usage:**
    - **A list of years to check is defined.**
    - **The program iterates over each year and calls the static method is_leap_year to determine if it is a leap year, printing the result accordingly.**

Output

When you run this code, you will see output similar to:

text

2000 is a leap year.

2001 is not a leap year.

2004 is a leap year.

1900 is not a leap year.

2020 is a leap year.

2100 is not a leap year.