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

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

Classes: A class is a blueprint for creating objects. It defines attributes (data) and methods (functions) that the objects created from the class will have.

Objects: An object is an instance of a class. It is a self-contained entity that contains both data (attributes) and methods (behavior) that operate on the data.

Encapsulation: Encapsulation refers to bundling data (attributes) and methods (functions) that operate on the data into a single unit or class. It also involves restricting access to certain components of an object to hide its internal state and protect the integrity of the data. This is often done using access specifiers like private, protected, and public.

Inheritance: Inheritance allows one class (child class or subclass) to inherit properties and methods from another class (parent class or superclass). This promotes code reusability and establishes a natural hierarchy between classes.

Polymorphism: Polymorphism allows objects of different classes to be treated as objects of a common superclass. It also allows methods to be defined in a base class and overridden in derived classes to provide specific implementations, enabling one interface to represent different underlying forms (e.g., method overloading and method overriding).

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

the car's information.

Here is a Python class for a Car with attributes for make, model, and year, along with a method to display the car's information:

```
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}")
my_car = Car("Toyota", "Camry", 2022)
my_car.display_info()

Car Information: 2022 Toyota Camry
```

In this example:

The Car class has a constructor (**init**) to initialize the attributes. The display_info method prints out the car's information in a readable format. The example usage creates a Car object and calls the display_info method to display its details.

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

1.Instance Methods:

Bound to an instance of the class.

They can access and modify instance variables (attributes) of the object.

The first parameter is always self, which refers to the instance of the class.

2.Class Methods:

Bound to the class rather than an instance.

They cannot modify instance variables, but they can modify class variables (shared by all instances of the class).

The first parameter is cls, which refers to the class itself. Class methods are defined using the @classmethod decorator.

```
# Example of Each
class Car:
    wheels = 4
    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}")

    @classmethod
    def update_wheels(cls, new_wheel_count):
        cls.wheels = new_wheel_count
        print(f"Updated wheel count for all cars: {cls.wheels}")

my_car = Car("Toyota", "Camry", 2022)
my_car.display_info()
Car.update_wheels(6)
```

```
Car Information: 2022 Toyota Camry
Updated wheel count for all cars: 6
```

Key Points:

Instance Method (display_info): Accesses individual instance attributes like self.make, self.model, etc.

Class Method (update_wheels): Accesses or modifies the class-level attribute wheels, shared across all instances, without needing an instance to be created.

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

In Python, method overloading (the ability to define multiple methods with the same name but different signatures) is not directly supported like in some other languages (e.g., Java or C++). However, Python can simulate method overloading using default arguments or **args and** kwargs to accept a variable number of arguments.

Example Using Default Arguments

One way to achieve a similar effect is by using default arguments in methods:

```python
class Calculator:
    def add(self, a, b=0, c=0):
        return a + b + c
calc = Calculator()
print(calc.add(5))
print(calc.add(5, 10))
print(calc.add(5, 10, 20))
```

```
5
15
35
```

Here, the add method behaves like it is "overloaded" for 1, 2, or 3 arguments due to the use of default values for b and c

Example Using *args for Flexible Argument Lists

Python can also use *args to accept a variable number of arguments:

```
class Calculator:

    def add(self, *args):
        return sum(args)


calc = Calculator()

print(calc.add(5))
print(calc.add(5, 10))
print(calc.add(5, 10, 20))
```

```
5
15
35
```

Here, the method add can handle any number of arguments by summing them up, simulating overloading.

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

In Python, access modifiers control the visibility and accessibility of class members (attributes and methods). There are three types of access modifiers, although Python does not enforce strict access control like some other languages (e.g., Java, C++). Instead, it uses naming conventions to suggest access control.

1. Public Members Description: Public members are accessible from anywhere, both inside and outside the class. Denoted by: No special notation is used; by default, all members in Python are public.

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

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

my_car = Car("Toyota", "Camry")
print(my_car.make)
my_car.display_info()
```

```
Toyota
Car: Toyota Camry
```

2. Protected Members

Description: Protected members are intended to be accessible within the class and its subclasses. By convention, these members are not meant to be accessed directly outside the class hierarchy. However, they can still be accessed from outside the class since Python does not strictly enforce this.

Denoted by: A single underscore _ before the member name.

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

    def _display_info(self):
        print(f"Car: {self._make} {_self.model}")

class ElectricCar(Car):
    def display_battery_info(self):
        print(f"Battery info for {self._make}")

my_car = ElectricCar("Tesla", "Model S")
print(my_car._make)
```

```
Tesla
```

3. Private Members

Description: Private members are intended to be inaccessible from outside the class. These members are accessible only within the class that defines them. Python achieves this using name mangling, where the interpreter changes the name of the private member internally to prevent accidental access.

Denoted by: Two underscores __ before the member name.

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

    def __display_info(self):
        print(f"Car: {self.__make} {self.__model}")

    def public_method(self):
        self.__display_info()


my_car = Car("Honda", "Accord")
my_car.public_method()
```

```
Car: Honda Accord
```

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

In Python, there are five main types of inheritance, which describe the different ways a class can inherit properties (attributes and methods) from one or more parent classes.

1. Single Inheritance

A class inherits from one parent class.

```python
# Example:
class Parent:
    def parent_method(self):
        print("Parent method")

class Child(Parent):
    def child_method(self):
        print("Child method")

child = Child()
child.parent_method()
```

```
Parent method
```

2. Multiple Inheritance

A class inherits from more than one parent class. Example (explained in detail below).

```python
# Example :
class Engine:
    def start_engine(self):
        print("Engine started")

class Transmission:
    def change_gear(self):
        print("Gear changed")

class Car(Engine, Transmission):
    def drive(self):
        print("Car is driving")

my_car = Car()
my_car.start_engine()
my_car.change_gear()
my_car.drive()
```

```
Engine started
Gear changed
Car is driving
```

3. Multilevel Inheritance

A class inherits from a parent class, which in turn inherits from another class. This forms a chain of inheritance.

```python
# Example:
class Grandparent:
    def grandparent_method(self):
        print("Grandparent method")

class Parent(Grandparent):
    def parent_method(self):
        print("Parent method")

class Child(Parent):
    def child_method(self):
        print("Child method")

child = Child()
child.grandparent_method()
```

```
Grandparent method
```

4. Hierarchical Inheritance

Multiple classes inherit from the same parent class.

```python
# Example:
class Parent:
    def parent_method(self):
        print("Parent method")

class Child1(Parent):
    def child1_method(self):
        print("Child1 method")

class Child2(Parent):
    def child2_method(self):
        print("Child2 method")

child1 = Child1()
child1.parent_method()
```

```
Parent method
```

5. Hybrid Inheritance

A combination of two or more types of inheritance, forming a more complex structure. It typically involves multiple and hierarchical inheritance.

```python
# Example:
class Parent:
    def parent_method(self):
        print("Parent method")

class Child1(Parent):
    def child1_method(self):
        print("Child1 method")

class Child2(Parent):
    def child2_method(self):
        print("Child2 method")

class GrandChild(Child1, Child2):
    def grandchild_method(self):
        print("GrandChild method")
```

```
grandchild = GrandChild()
grandchild.parent_method()

Parent method
```

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

The Method Resolution Order (MRO) in Python is the order in which Python looks for a method or attribute in a hierarchy of classes during inheritance, especially in the case of multiple inheritance. When a class inherits from multiple classes, the MRO defines the sequence in which the parent classes are searched to resolve a method or attribute call.

Python uses the C3 linearization algorithm (also known as C3 superclass linearization) to create the MRO. This ensures a consistent and predictable order in resolving methods, even with complex inheritance structures.

MRO and Multiple Inheritance

In multiple inheritance, if two or more parent classes have the same method or attribute name, Python will follow the MRO to determine which one to call first. The MRO ensures that:

The child class is checked first.

Parent classes are searched in a depth-first, left-to-right manner (in the order in which they are inherited).

The order respects the inheritance chain to avoid inconsistencies or ambiguity.

How to Retrieve the MRO Programmatically

You can retrieve the MRO of a class using:

The **mro** attribute.

The built-in mro() method.

```
# Example 1: Using __mro__ Attribute
class A:
    pass

class B(A):
    pass

class C(A):
    pass

class D(B, C):
    pass

print(D.__mro__)
```

```
(<class '__main__.D'>, <class '__main__.B'>, <class '__main__.C'>,
<class '__main__.A'>, <class 'object'>)
```

Example 2: Using mro() Method

```
class A:
    pass

class B(A):
    pass

class C(A):
    pass

class D(B, C):
    pass

print(D.mro())
```

```
[<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.

##Here is how you can create an abstract base class Shape with an abstract method area(), and two subclasses Circle and Rectangle that implement the area() method:

Step-by-Step Explanation:

Abstract Base Class (Shape):

Python provides the abc module to define abstract base classes. The abstract class contains at least one abstract method, which is declared but not implemented.

Subclasses (Circle and Rectangle):

Subclasses inherit from the abstract base class and must provide an implementation of the abstract method (area()).

```python
# Code Example
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


circle = Circle(5)
rectangle = Rectangle(4, 6)

print(f"Circle area: {circle.area()}")
print(f"Rectangle area: {rectangle.area()}")
```

```
Circle area: 78.53981633974483
Rectangle area: 24
```

Explanation:

Shape (Abstract Base Class):

Inherits from ABC (Abstract Base Class) from the abc module. The method area() is marked as an abstract method using the @abstractmethod decorator.

Any subclass of Shape must implement the area() method. Circle (Subclass):

Implements the area() method, which calculates the area of a circle using the formula:

π

r

2

πr

2

, where r is the radius. Rectangle (Subclass):

Implements the area() method, which calculates the area of a rectangle using the formula: width × height.

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

Here’s a simple demonstration of polymorphism using a function that calculates the area of different shape objects, such as a rectangle and a circle.

We’ll use Python, where polymorphism is demonstrated by creating a base class Shape with a method area(). Different shapes like Rectangle and Circle will inherit from Shape and provide their own implementation of the area() method.

```
import math

class Shape:
    def area(self):
        pass


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

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

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

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

def print_area(shape):
    print(f"The area is: {shape.area()}")

rectangle = Rectangle(10, 5)
circle = Circle(7)

print_area(rectangle)
print_area(circle)
```

```
The area is: 50
The area is: 153.93804002589985
```

Explanation:

The Shape class is the base class with a placeholder area() method. The Rectangle and Circle classes inherit from Shape and provide their own implementations of area().

The print_area function accepts any object that is derived from Shape and calls its area() method. This demonstrates polymorphism, as the same function can handle different types of shapes.

You can add more shape classes (like Triangle or Square) without changing the print_area function, showing how polymorphism enables extensibility.

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

Here’s an implementation of encapsulation in a BankAccount class, where balance and account_number are private attributes. We'll provide public methods for depositing, withdrawing, and inquiring about the balance, while keeping the internal state secure.

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

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

    def get_balance(self):
        return f"Current balance: ${self.__balance}"
    def get_account_number(self):
        return self.__account_number

account = BankAccount("1234567890", 1000)
print(account.get_balance())
account.deposit(500)
account.withdraw(200)
account.withdraw(1500)
```

```
Current balance: $1000
Deposited: $500. New balance: $1500.
Withdrew: $200. New balance: $1300.
Insufficient balance for the withdrawal.
```

#Explanation:

Private attributes: __account_number and **balance are prefixed with** , making them private and inaccessible directly from outside the class.

Public methods:

deposit(): Adds money to the account if the deposit amount is positive.

withdraw(): Subtracts money from the account if the withdrawal amount is positive and doesn't exceed the balance.

get_balance(): Returns the current balance.

get_account_number(): Returns the account number.

This design ensures that the balance and account_number can't be changed directly from outside the class, maintaining the integrity of the data (encapsulation).

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

##Overriding the **str** and **add** magic methods in a class provides specific functionality:

str: Controls how the object is displayed when converted to a string, like when you call print() on the object.

add: Defines how objects of the class are added together using the + operator.

```
# Here's an example to demonstrate both:
class Book:
    def __init__(self, title, pages):
        self.title = title
        self.pages = pages

    def __str__(self):
        return f"'{self.title}' with {self.pages} pages"

    def __add__(self, other):
```

```python
        if isinstance(other, Book):
            total_pages = self.pages + other.pages
            return f"Combined total pages: {total_pages}"
        return "Cannot add non-Book object to a Book."

book1 = Book("The Great Gatsby", 180)
book2 = Book("1984", 328)

print(book1)
print(book2)

print(book1 + book2)
```

```
'The Great Gatsby' with 180 pages
'1984' with 328 pages
Combined total pages: 508
```

Explanation:

str:

This method returns a string when an object is passed to print(), or whenever the object is implicitly converted to a string. In this case, it provides a custom string that includes the book's title and the number of pages. For print(book1), it returns "'The Great Gatsby' with 180 pages".

add:

This method allows you to define how two objects of the class should be added together using the + operator. In this case, when two Book objects are added, the total number of pages is calculated and displayed.

For book1 + book2, it returns "Combined total pages: 508".

What these methods allow:

str: Allows an intuitive and human-readable string representation of an object.

add: Allows custom behavior when using the + operator between instances of a class.

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

Here's how you can create a Python decorator that measures and prints the execution time of a function:

```python
import time

def measure_time(func):
    def wrapper(*args, **kwargs):
        start_time = time.time()
        result = func(*args, **kwargs)
        end_time = time.time()
        execution_time = end_time - start_time
        print(f"Execution time: {execution_time:.6f} seconds")
        return result
    return wrapper

@measure_time
def example_function(n):
    sum_result = 0
    for i in range(n):
        sum_result += i
    return sum_result

result = example_function(1000000)
print(f"Result: {result}")
```

```
Execution time: 0.075181 seconds
Result: 499999500000
```

Explanation:

measure_time decorator:

It wraps the original function inside another function (wrapper) that measures the execution time.

The time.time() function is used to record the time before and after the function runs.

The difference between the start and end times is calculated and printed.

Decorator usage:

The @measure_time decorator is placed above the example_function to automatically apply the execution time measurement.

When example_function(1000000) is called, it sums numbers from 0 to 999,999 and prints the time taken to perform the calculation.

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

##The Diamond Problem in multiple inheritance occurs when a class inherits from two or more classes that have a common ancestor, leading to ambiguity about which inherited method or attribute to use. The name comes from the inheritance structure, which resembles a diamond shape.

Diamond Problem Example:

Suppose we have the following class structure:

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

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

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

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

Here:

Class A is the base class.

Classes B and C inherit from A.

Class D inherits from both B and C.

When we create an instance of D and call the greet method, there's ambiguity about whether D should use greet from class B or class C, because both of them inherit from A.

```
d = D()
d.greet()

Hello from B
```

The Diamond Problem:

In this scenario, both B and C inherit from A, and D inherits from both B and C. If you call greet on an instance of D, the interpreter might get confused about which path to follow:

Should it call B's implementation of greet or C's?

This is the Diamond Problem, where multiple inheritance creates ambiguity in the method resolution order (MRO).

Python's Resolution: Method Resolution Order (MRO)

Python resolves the Diamond Problem using C3 linearization (also called MRO) to determine the order in which base classes are searched when calling a method.

To see the MRO in Python, you can use the mro() method or **mro** attribute:

```
print(D.mro())

[<class '__main__.D'>, <class '__main__.B'>, <class '__main__.C'>,
<class '__main__.A'>, <class 'object'>]
```

In this case, Python follows the depth-first, left-to-right rule:

First, it looks at D.

Then it checks B (the first parent class of D).

After that, it checks C (the second parent class of D).

Finally, it looks at A (the common ancestor of B and C).

Key Points:

MRO (C3 linearization): Python resolves the ambiguity by determining a specific method resolution order, ensuring there’s no confusion about which method to use.

How MRO works:

It follows a left-to-right and depth-first rule, checking parent classes one at a time and skipping any class that has already been checked.

Avoiding confusion: You can use super() in classes to ensure that methods are called in the proper MRO order, further simplifying multiple inheritance.

```python
# For example:
class A:
    def greet(self):
        print("Hello from A")

class B(A):
    def greet(self):
        super().greet()
        print("Hello from B")

class C(A):
    def greet(self):
        super().greet()
        print("Hello from C")

class D(B, C):
    def greet(self):
        super().greet()

d = D()
d.greet()
```

```
Hello from A
Hello from C
Hello from B
```

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

##To keep track of the number of instances created from a class, you can use a class variable and a class method. The class variable will store the count, and the class method will return the number of instances created.

Here's an example:

```python
class InstanceCounter:
    instance_count = 0
    def __init__(self):

        InstanceCounter.instance_count += 1
    @classmethod
    def get_instance_count(cls):
        return cls.instance_count

obj1 = InstanceCounter()
obj2 = InstanceCounter()
obj3 = InstanceCounter()

print(InstanceCounter.get_instance_count())

3
```

Explanation:

Class variable (instance_count):

This variable belongs to the class itself, not to any instance, so all instances share it.

It is initialized to 0 and gets incremented by 1 every time a new instance of InstanceCounter is created (in the **init** method).

Class method (get_instance_count):

The @classmethod decorator allows this method to access the class (using the cls parameter) and return the current value of instance_count. You can call this method on the class to know how many instances have been created.

Example usage:

When obj1, obj2, and obj3 are created, the instance_count is incremented each time.

The get_instance_count method returns 3, as three instances have been created. This demonstrates how you can track the number of instances of a class using a class method and class variable.

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

Here’s how you can implement a static method in a class that checks if a given year is a leap year:

```
class DateUtils:
    @staticmethod
    def is_leap_year(year):
        if (year % 4 == 0 and year % 100 != 0) or (year % 400 == 0):
            return True
        return False


print(DateUtils.is_leap_year(2020))
print(DateUtils.is_leap_year(2021))
print(DateUtils.is_leap_year(1900))
print(DateUtils.is_leap_year(2000))
```

```
True
False
False
True
```

Explanation:

Static method (is_leap_year):

Defined using the @staticmethod decorator. Static methods do not require access to the instance (self) or the class (cls), as they operate independently of the object or class state.

The static method is_leap_year accepts a year as a parameter and returns True if the year is a leap year and False otherwise.

Leap year logic:

A year is a leap year if: It is divisible by 4 and not divisible by 100, or It is divisible by 400.

Example usage:

The static method can be called directly on the class (DateUtils.is_leap_year(year)), without the need to create an instance of the class. This implementation allows you to check if a year is a leap year in an easy and reusable way through the static method.