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

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

1. Encapsulation o Encapsulation is the bundling of data (attributes) and methods (functions) that operate on the data into a single unit, called a class. o It restricts direct access to some of the object's components, which helps protect the integrity of the data. Access is typically controlled using access modifiers such as private, protected, and public.
2. Abstraction o Abstraction involves hiding the complex implementation details of a system and exposing only the essential features. o It allows a user to interact with the object through a simplified interface without needing to understand the underlying complexities.
3. Inheritance o Inheritance allows a class (child or derived class) to inherit properties and methods from another class (parent or base class). o It promotes code reuse and establishes a hierarchical relationship between classes.
4. Polymorphism o Polymorphism allows objects of different classes to be treated as objects of a common superclass. o It can be achieved through method overloading (compile-time polymorphism) and method overriding (runtime polymorphism), enabling flexibility and dynamic behavior.
5. Composition (or Association) o While not always explicitly listed, composition is a key principle in OOP where objects are built using references to other objects. o It describes a "has-a" relationship (e.g., a car has a steering wheel), enabling modular and scalable design.

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

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

    def display_info(self):
        """Display the car's information."""
        return f"{self.year} {self.make} {self.model}"

# Example usage
my_car = Car("Toyota", "Corolla", 2022)
print(my_car.display_info())  # Output: 2022 Toyota Corolla
```

```
2022 Toyota Corolla
```

3. Difference Between Instance Methods and Class Methods: Instance Methods:

Operate on instances of the class. Require self as the first parameter, which refers to the instance of the class. Can access and modify instance attributes. Class Methods:

Operate on the class itself rather than instances. Require cls as the first parameter, which refers to the class. Defined using the @classmethod decorator. Cannot access instance-specific data but can access and modify class-level data.

```
class Car:
    # Class attribute
    total_cars = 0

    def __init__(self, make, model, year):
        self.make = make
        self.model = model
        self.year = year
        Car.total_cars += 1

    # Instance method
    def display_info(self):
        """Displays the car's information."""
        return f"{self.year} {self.make} {self.model}"

    # Class method
    @classmethod
    def get_total_cars(cls):
        """Returns the total number of cars created."""
        return f"Total cars: {cls.total_cars}"

# Example usage
```

```
car1 = Car("Toyota", "Corolla", 2022)
car2 = Car("Honda", "Civic", 2023)

print(car1.display_info())          # Output: 2022 Toyota Corolla (Instance method)
print(Car.get_total_cars())         # Output: Total cars: 2 (Class method)
```

```
2022 Toyota Corolla
Total cars: 2
```

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

Example of Simulating Method Overloading:

```
class Calculator:
    def add(self, a, b=0, c=0):
        """Add up to three numbers."""
        return a + b + c

# Example usage
calc = Calculator()
print(calc.add(5))             # Output: 5 (1 argument)
print(calc.add(5, 10))         # Output: 15 (2 arguments)
print(calc.add(5, 10, 15))     # Output: 30 (3 arguments)
```

```
5
15
30
```

Using Variable-Length Arguments (*args):

```
class Calculator:
    def add(self, *args):
        """Add any number of values."""
        return sum(args)

# Example usage
calc = Calculator()
print(calc.add(5))             # Output: 5
print(calc.add(5, 10))         # Output: 15
print(calc.add(5, 10, 15, 20)) # Output: 50
```

```
5
15
50
```

Type Checking for Overloading-Like Behavior:

```
class Printer:
    def print_value(self, value):
        if isinstance(value, int):
            print(f"Integer: {value}")
        elif isinstance(value, str):
            print(f"String: '{value}'")
        else:
            print(f"Other type: {value}")

# Example usage
printer = Printer()
printer.print_value(42)           # Output: Integer: 42
printer.print_value("Hello")      # Output: String: 'Hello'
printer.print_value([1, 2, 3])    # Output: Other type: [1, 2, 3]
```

```
Integer: 42
String: 'Hello'
Other type: [1, 2, 3]
```

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

Access Modifiers in Python Python supports three types of access modifiers for controlling the visibility of class attributes and methods:

1. Public Access Modifier Description: Attributes and methods are accessible from anywhere. Notation: No special prefix (default). Usage: Public members are the default in Python.

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

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

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

```
Toyota
Toyota Corolla
```

2. Protected Access Modifier Description: Attributes and methods are accessible within the class and its subclasses, but not recommended for external use. Notation: Single underscore prefix _attribute or _method. Usage: Indicates that the member is intended for internal use.

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

    def _display_info(self):     # Protected method
        return f"{self._make} {self._model}"

class ElectricCar(Car):
    def get_info(self):
        return self._display_info()  # Accessible in subclass

e_car = ElectricCar("Tesla", "Model S")
print(e_car.get_info())              # Accessible
```

```
Tesla Model S
```

3. Private Access Modifier Description: Attributes and methods are accessible only within the class where they are defined. Notation: Double underscore prefix __attribute or __method. Usage: Provides encapsulation and prevents external access.

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

    def __display_info(self):    # Private method
        return f"{self.__make} {self.__model}"

    def get_info(self):
        return self.__display_info()  # Access private method internally

car = Car("Ford", "Mustang")
print(car.get_info())                # Accessible via public method
# print(car.__make)                  # Raises AttributeError
```

```
Ford Mustang
```

Name Mangling for Private Members: Private members are internally name-mangled to prevent direct access. For example, **make becomes _Car**make to prevent accidental overrides.

```
print(car._Car__make)  # Output: Ford (Accessing private member via name mangling)
```

```
Ford
```

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

Types of Inheritance in Python Inheritance allows a class (child) to inherit attributes and methods from another class (parent). Python supports five types of inheritance:

1. Single Inheritance Description: A child class inherits from one parent class.

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

class Child(Parent):
    pass

child = Child()
print(child.greet())  # Output: Hello from Parent
```

```
Hello from Parent
```

2. Multiple Inheritance Description: A child class inherits from multiple parent classes.

```
class Parent1:
    def greet1(self):
        return "Hello from Parent1"

class Parent2:
    def greet2(self):
        return "Hello from Parent2"

class Child(Parent1, Parent2):
    pass

child = Child()
print(child.greet1())
print(child.greet2())
```

```
Hello from Parent1
Hello from Parent2
```

3. Multilevel Inheritance Description: A class is derived from a child class, forming a chain of inheritance.

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

class Parent(Grandparent):
    pass

class Child(Parent):
    pass

child = Child()
print(child.greet())
```

```
Hello from Grandparent
```

4. Hierarchical Inheritance Description: Multiple child classes inherit from a single parent class.

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

```
class Child1(Parent):
    pass

class Child2(Parent):
    pass

child1 = Child1()
child2 = Child2()
print(child1.greet())
print(child2.greet())
```

```
Hello from Parent
Hello from Parent
```

5. Hybrid Inheritance Description: A combination of multiple types of inheritance.

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

class Child1(Parent):
    pass

class Child2(Parent):
    pass

class GrandChild(Child1, Child2):
    pass

grandchild = GrandChild()
print(grandchild.greet())
```

```
Hello from Parent
```

Multiple Inheritance in Detail:

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

class Bird:
    def fly(self):
        return "I can fly"

class Bat(Animal, Bird):
    pass

bat = Bat()
print(bat.speak())
print(bat.fly())
```

```
Animal sound
I can fly
```

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

Method Resolution Order (MRO) in Python The Method Resolution Order (MRO) is the sequence in which Python looks for methods and attributes in a class hierarchy. This is particularly important in the context of multiple inheritance to determine the order in which classes are searched when calling a method.

Python uses the C3 Linearization Algorithm (also known as the C3 superclass linearization) to compute the MRO. This ensures:

Left-to-right depth-first search in the inheritance tree. Consistency in method resolution across multiple inheritance paths. Avoidance of ambiguity in cases where a method appears in multiple parent classes.

```
class A:
    def show(self):
        return "A"

class B(A):
    def show(self):
        return "B"

class C(A):
    def show(self):
        return "C"

class D(B, C):
    pass

d = D()
print(d.show())
```

```
B
```

In this example, D inherits from B and C. When d.show() is called, Python looks for the show method following the MRO and finds it in B first.

How to Retrieve MRO Programmatically: Using **mro** Attribute:

```
print(D.__mro__)
```

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

2.Using mro() Method:

```
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.

Abstract Base Class in Python with Subclasses In Python, abstract base classes (ABCs) are used to define a blueprint for other classes. ABCs can have abstract methods that must be implemented by subclasses. The abc module provides the tools to create abstract classes.

Example: Abstract Base Class Shape with Subclasses Circle and Rectangle

```
from abc import ABC, abstractmethod
import math

# Abstract base class
class Shape(ABC):
    @abstractmethod
    def area(self):
        """Abstract method to calculate the area."""
        pass

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

    def area(self):
        """Calculate the area of the circle."""
        return math.pi * self.radius ** 2

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

    def area(self):
        """Calculate the area of the rectangle."""
        return self.width * self.height

# Example usage
circle = Circle(5)
rectangle = Rectangle(4, 6)

print(f"Circle area: {circle.area():.2f}")      # Output: Circle area: 78.54
print(f"Rectangle area: {rectangle.area()}")    # Output: Rectangle area: 24
```

```
Circle area: 78.54
Rectangle area: 24
```

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

Demonstrating Polymorphism with Shapes Polymorphism allows methods to operate on objects of different classes through a common interface. In this example, we'll use a function that works with different shape objects, each implementing the area() method.

```
from abc import ABC, abstractmethod
import math

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

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

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

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

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

# Triangle subclass
class Triangle(Shape):
    def __init__(self, base, height):
        self.base = base
        self.height = height

    def area(self):
        return 0.5 * self.base * self.height

# Polymorphic function to calculate and print areas
def print_area(shape):
    print(f"The area of the {shape.__class__.__name__} is: {shape.area():.2f}")

# Example usage
shapes = [
    Circle(5),
    Rectangle(4, 6),
    Triangle(3, 4)
]

for shape in shapes:
    print_area(shape)
```

```
The area of the Circle is: 78.54
The area of the Rectangle is: 24.00
The area of the Triangle is: 6.00
```

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

Encapsulation in a BankAccount Class Encapsulation involves restricting direct access to an object's attributes and providing controlled access through methods. Below is a BankAccount class demonstrating encapsulation with private attributes and public methods for interaction.

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

    # Method to deposit money
    def deposit(self, amount):
        if amount > 0:
            self.__balance += amount
            print(f"Deposited: ${amount:.2f}")
        else:
            print("Deposit amount must be positive.")

    # Method to withdraw money
    def withdraw(self, amount):
        if amount <= 0:
            print("Withdrawal amount must be positive.")
        elif amount > self.__balance:
            print("Insufficient funds.")
        else:
            self.__balance -= amount
            print(f"Withdrew: ${amount:.2f}")

    # Method to check balance
    def get_balance(self):
        return self.__balance

    # Method to get account number
    def get_account_number(self):
        return self.__account_number

# Example usage
account = BankAccount("1234567890", 500)

# Deposit
account.deposit(200)
print(f"Balance: ${account.get_balance():.2f}")  # Output: 700.00

# Withdrawal
account.withdraw(100)
print(f"Balance: ${account.get_balance():.2f}")  # Output: 600.00

# Attempting direct access to private attributes (will raise an AttributeError)
# print(account.__balance)  # Uncommenting this will raise an AttributeError
```

```
Deposited: $200.00
Balance: $700.00
Withdrew: $100.00
Balance: $600.00
```

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

Overriding **str** and **add** Magic Methods In Python, the **str** and **add** magic methods allow you to customize the string representation of an object and define how objects of a class can be added together, respectively.

str: This method is used to define the string representation of an object, which is what Python will use when you call str() on an instance of the class or use print() with the object. **add**: This method is used to define the behavior of the addition operator (+) for objects of the class.

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

    # Overriding the __str__ method
    def __str__(self):
        return f"Book: {self.title} by {self.author}"
    # Overriding the __add__ method
    def __add__(self, other):
        if isinstance(other, Book):
            return f"Combined Book: {self.title} & {other.title}"
        return "Cannot combine with non-Book object"

# Example usage
book1 = Book("The Great Gatsby", "F. Scott Fitzgerald")
book2 = Book("1984", "George Orwell")

# Using the __str__ method
print(book1)

# Using the __add__ method
combined_book = book1 + book2
print(combined_book)
```

```
Book: The Great Gatsby by F. Scott Fitzgerald
Combined Book: The Great Gatsby & 1984
```

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

Creating a Decorator to Measure Execution Time A decorator is a function that wraps another function to modify or enhance its behavior. In this case, we can create a decorator that measures and prints the execution time of a function.

We'll use Python's built-in time module to track the start and end times of the function's execution.

```
import time

# Decorator to measure execution time
def time_execution(func):
    def wrapper(*args, **kwargs):
        start_time = time.time()  # Record start time
        result = func(*args, **kwargs)  # Call the decorated function
        end_time = time.time()  # Record 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 wrapper

# Example function to decorate
@time_execution
def slow_function():
    time.sleep(2)  # Simulating a slow operation by sleeping for 2 seconds
    print("Function finished execution!")

# Example usage
slow_function()
```

```
Function finished execution!
Execution time of slow_function: 2.0031 seconds
```

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

The Diamond Problem in Multiple Inheritance The Diamond Problem (also called the Deadly Diamond of Death) occurs in object-oriented programming when a class inherits from two classes that have a common ancestor. This can lead to ambiguity in the method resolution order (MRO) because the child class inherits from two classes that, in turn, inherit from the same base class. This creates a "diamond-shaped" inheritance structure.

The Diamond Problem Diagram: css Copy code A /
B C \ / D Class D inherits from both Class B and Class C. Both Class B and Class C inherit from Class A. If Class D calls a method from Class A, it can be unclear whether it should call the method from B or C because both are derived from A. Issues Caused by the Diamond Problem: Ambiguity: When a method is called on a class that inherits from multiple classes, there could be confusion as to which parent class's method should be invoked. Redundancy: Without proper method resolution, the parent class's method might be inherited multiple times, leading to potential redundancy. How Python Resolves the Diamond Problem: Python resolves the Diamond Problem using a method called C3 Linearization or C3 superclass linearization. This algorithm ensures a consistent order in which classes are searched for methods.

The C3 Linearization gives a specific order in which methods should be inherited, taking into account both depth-first search and left-to-right traversal. Python uses the Method Resolution Order (MRO) to determine the order in which classes are searched when calling a method. How Python Handles MRO: In Python, the MRO (Method Resolution Order) is computed using C3 Linearization, which ensures that each class in the inheritance chain is only called once, and that the method search follows a consistent order.

You can view the MRO of a class by using the mro() method or the **mro** attribute.

```
class A:
    def show(self):
        print("A's show method")

class B(A):
    def show(self):
        print("B's show method")

class C(A):
    def show(self):
        print("C's show method")

class D(B, C):
    pass

# Example usage
d = D()
d.show()
```

```
B's show method
```

Viewing the MRO of a Class: You can inspect the method resolution order (MRO) of a class by calling the mro() method or by accessing the **mro** attribute.

```
print(D.mro())  # This will print the MRO of class D
```

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

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

Class Method to Track the Number of Instances In Python, you can use a class method to modify or access the state of the class itself (as opposed to instance methods, which operate on individual object instances). To track the number of instances created from a class, you can use a class attribute, and the class method can update this attribute every time a new instance is created.

```
class MyClass:
    # Class attribute to keep track of the number of instances
    instance_count = 0

    def __init__(self):
        # Increment the instance count every time a new instance is created
        MyClass.instance_count += 1

    # Class method to get the current number of instances
    @classmethod
    def get_instance_count(cls):
        return cls.instance_count

# Example usage
obj1 = MyClass()  # Creates the first instance
obj2 = MyClass()  # Creates the second instance
obj3 = MyClass()  # Creates the third instance

print(MyClass.get_instance_count())
```

```
3
```

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

Static Method to Check if a Year is a Leap Year A static method is a method that belongs to the class rather than an instance of the class. It does not take the instance (self) or class (cls) as its first argument, and it is used for functionality that doesn't depend on the state of the instance or class.

To implement a static method that checks if a given year is a leap year, we need to follow these rules:

A year is a leap year if it is divisible by 4. However, if the year is divisible by 100, it must also be divisible by 400 to be considered a leap year.

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

# Example usage
year = 2024
print(f"{year} is a leap year: {YearUtils.is_leap_year(year)}")

year = 2023
print(f"{year} is a leap year: {YearUtils.is_leap_year(year)}")
```

```
2024 is a leap year: True
2023 is a leap year: False
```