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: This concept involves bundling data and methods that manipulate that data into a single unit, called a class or object. This helps to hide the implementation details and protect the data from external interference.
   2. Abstraction: Abstraction involves showing only the necessary information to the outside world while hiding the internal details. This helps to reduce complexity and improve modularity.
   3. Inheritance: Inheritance allows one class to inherit the properties and behavior of another class. This promotes code reuse and facilitates the creation of a hierarchy of related classes.
   4. Polymorphism: Polymorphism is the ability of an object to take on multiple forms, depending on the context. This can be achieved through method overloading or method overriding.
   5. Composition: Composition involves combining multiple objects or classes to form a new object or class. This helps to create complex systems from simpler components and promotes modularity and reuse.
   6. Write a Python class for a Car with attributes for make, model, and year. Include a method to display the car's information.

ANS.Here's a Python class for a Car:

class Car: """ A class representing a Car with attributes for make, model, and year. """

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

"""

Initializes a Car object with make, model, and year.

Args:

make (str): The car's make.

model (str): The car's model.

year (int): The car's year.

"""

self.make = make

self.model = model

self.year = year

def display\_info(self):

"""

Displays the car's information.

"""

print(f"Make: {self.make}") print(f"Model: {self.model}") print(f"Year: {self.year}")

Example usage my\_car = Car("Toyota", "Camry", 2020) my\_car.display\_info() Output:

Make: Toyota Model: Camry Year: 2020

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

In Python, instance methods and class methods are two types of methods that can be defined in a class.

Instance Methods Instance methods are methods that belong to an instance of a class. They are used to perform operations that are specific to a particular instance. Instance methods have access to the instance's attributes and can modify them.

Example of an instance method:

class Dog: def init(self, name, age): self.name = name self.age = age

def bark(self):

print(f"{self.name} says Woof!")

my\_dog = Dog("Fido", 3) my\_dog.bark() # Output: Fido says Woof!

Class Methods Class methods are methods that belong to a class rather than an instance of the class. They are used to perform operations that are related to the class itself, rather than to a specific instance. Class methods have access to the class's attributes and can modify them.

Example of a class method: class Dog: num\_dogs = 0

def init (self, name, age):

self.name = name

self.age = age

Dog.num\_dogs += 1

@classmethod

def get\_num\_dogs(cls):

return cls.num\_dogs

my\_dog1 = Dog("Fido", 3) my\_dog2 = Dog("Rex", 2) print(Dog.get\_num\_dogs()) # Output: 2

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

ANS.Python does not support method overloading in the classical sense, unlike languages such as Java or C++. In Python, method overloading is achieved using optional arguments or default argument values.

Here's an example:

class Calculator: def calculate(self, \*args): if len(args) == 1: return args[0] \*\* 2 elif len(args) == 2: return args[0] + args[1] else: raise ValueError("Invalid number of arguments")

calculator = Calculator() print(calculator.calculate(5)) # Output: 25 print(calculator.calculate(5, 3)) # Output: 8

In this example, the calculate method can take either one or two arguments. The method's behavior changes depending on the number of arguments passed.

Alternatively, you can use the \*\*kwargs syntax to accept a variable number of keyword arguments:

class Calculator: def calculate(self, kwargs): if 'square' in kwargs: return kwargs['square'] 2 elif 'add' in kwargs: return kwargs['add'][0] + kwargs['add'][1] else: raise ValueError("Invalid keyword arguments")

calculator = Calculator() print(calculator.calculate(square=5)) # Output: 25 print(calculator.calculate(add=(5, 3))) # Output: 8

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

ANS.Python does not have strict access modifiers like some other languages (e.g., Java, C++). However, Python does have some conventions to indicate the intended level of access:

1. Public members: No underscore prefix. These members are intended to be accessed directly and are part of the public API.
2. Protected members (convention): Single underscore prefix (e.g., \_variable). These members are intended to be accessed within the class and its subclasses, but not directly from outside the class.
3. Private members (convention): Double underscore prefix (e.g., variable). These members are intended to be accessed only within the class itself and are name- mangled to avoid accidental access from outside the class.

Note that Python's access control is not enforced at runtime, but rather serves as a convention and a hint to other developers.

Here's an example:

class MyClass: def init(self): self.public\_variable = 10 # Public member self.\_protected\_variable

= 20 # Protected member (convention) self. private\_variable = 30 # Private member (convention)

def access\_variables(self):

print(self.public\_variable)

print(self.\_protected\_variable)

print(self. private\_variable)

obj = MyClass() obj.access\_variables() # Accessing variables within the class

print(obj.public\_variable) # Accessing public variable directly print(obj.\_protected\_variable) # Accessing protected variable directly (not recommended)

Accessing private variable directly is not recommended and will raise an AttributeError print(obj. private\_variable)

However, you can still access the private variable using name mangling print(obj.\_MyClass private\_variable)

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

ANS.Here are the five types of inheritance in Python:

1. Single Inheritance: A child class inherits from a single parent class.
2. Multiple Inheritance: A child class inherits from multiple parent classes.
3. Multilevel Inheritance: A child class inherits from a parent class, which in turn inherits from another parent class.
4. Hierarchical Inheritance: Multiple child classes inherit from a single parent class.
5. Hybrid Inheritance: A combination of multiple inheritance types.

Here's an example of multiple inheritance:

class Animal: def eat(self): print("Eating...") class Mammal: def walk(self): print("Walking...")

class Dog(Animal, Mammal): def bark(self): print("Barking...")

my\_dog = Dog() my\_dog.eat() # Output: Eating... my\_dog.walk() # Output: Walking... my\_dog.bark() # Output: Barking...

In this example, the Dog class inherits from both the Animal and Mammal classes, demonstrating multiple inheritance.

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

ANS. The Method Resolution Order (MRO) in Python is the order in which Python searches for methods in a class hierarchy when using multiple inheritance. It's a standard for resolving the order of methods in a class hierarchy.

Python uses a technique called C3 Linearization to resolve the MRO. This algorithm ensures that the MRO is consistent and predictable.

To retrieve the MRO programmatically, you can use the mro() method, which returns a list of classes in the order they would be searched for methods.

Here's an example:

class Animal: pass

class Mammal(Animal): pass class Dog(Mammal): pass print(Dog.mro())

Output:

(<class 'main.Dog'>, <class 'main.Mammal'>, <class 'main.Animal'>, <class 'object'>)

This output shows the MRO for the Dog class, which includes the Dog class itself, followed by its parent classes Mammal, Animal, and finally the object class, which is the base class for

all Python classes.

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

ANS.Here's an example implementation in Python:

from abc import ABC, abstractmethod import math

Abstract base class Shape class Shape(ABC): @abstractmethod def area(self): pass Subclass Circle class Circle(Shape): def init(self, radius): self.radius = radius

def area(self):

return math.pi \* (self.radius \*\* 2)

Subclass Rectangle 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) print(f"Circle area: {circle.area():.2f}") rectangle = Rectangle(4, 6) print(f"Rectangle area: {rectangle.area()}") Output:

Circle area: 78.54 Rectangle area: 24 In this example:

1. We define an abstract base class Shape with an abstract method area().
2. We create two subclasses Circle and Rectangle that inherit from Shape.
3. Each subclass implements the area() method according to its specific geometry.
4. We demonstrate the usage of these classes by creating instances and calculating their areas.
5. Demonstrate polymorphism by creating a function that can work with different shape objects to calculate and print their areas.

ANS.Here's an example demonstration of polymorphism in Python: from abc import ABC, abstractmethod import math

Abstract base class Shape class Shape(ABC): @abstractmethod def area(self): pass Subclass Circle class Circle(Shape): def init(self, radius): self.radius = radius

def area(self):

return math.pi \* (self.radius \*\* 2)

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

def area(self):

return self.width \* self.height

Subclass Triangle 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 shape areas def print\_shape\_areas(shapes): for shape in shapes: print(f"{type(shape). name } area: {shape.area():.2f}")

Example usage circle = Circle(5) rectangle = Rectangle(4, 6) triangle = Triangle(3, 7) shapes = [circle, rectangle, triangle] print\_shape\_areas(shapes)

Output:

Circle area: 78.54 Rectangle area: 24.00 Triangle area: 10.50 In this example:

1. We define an abstract base class Shape with an abstract method area().
2. We create three subclasses Circle, Rectangle, and Triangle that inherit from Shape.
3. Each subclass implements the area() method according to its specific geometry.
4. We define a polymorphic function print\_shape\_areas() that takes a list of shape objects as input.
5. Within the function, we iterate over the shape objects and call their area() methods to calculate and print their areas.
6. We demonstrate the usage of this polymorphic function by creating a list of shape objects and passing it to the function.
7. Implement encapsulation in a BankAccount class with private attributes for balance and account\_number. Include methods for deposit, withdrawal, and balance inquiry

ANS.Here's an example implementation of encapsulation in a BankAccount class in Python:

class BankAccount: def init(self, account\_number, initial\_balance): self. account\_number = account\_number self. balance = initial\_balance

def deposit(self, amount):

if amount > 0:

self. balance += amount

print(f"Deposited ${amount:.2f}. New balance: $

{self. balance:.2f}")

else:

print("Invalid deposit amount.")

def withdraw(self, amount):

if 0 < amount <= self. balance:

self. balance -= amount

print(f"Withdrew ${amount:.2f}. New balance: $

{self. balance:.2f}")

elif amount <= 0:

print("Invalid withdrawal amount.")

else:

print("Insufficient funds.")

def get\_balance(self):

return self. balance

def get\_account\_number(self): return self. account\_number

Example usage account = BankAccount("1234567890", 1000.0) print(f"Account Number:

{account.get\_account\_number()}") print(f"Initial Balance: ${account.get\_balance():.2f}") account.deposit(500.0) account.withdraw(200.0) account.withdraw(1500.0) # Insufficient funds Output:

Account Number: 1234567890 Initial Balance: $1000.00 Deposited $500.00. New balance:

$1500.00 Withdrew $200.00. New balance: $1300.00 Insufficient funds. In this example:

1. We define a BankAccount class with private attributes account\_number and

balance.

1. We provide public methods deposit(), withdraw(), get\_balance(), and get\_account\_number() to interact with the private attributes.
2. The deposit() and withdraw() methods modify the balance attribute while performing validation checks.
3. The get\_balance() and get\_account\_number() methods provide read-only access to the private attributes.
4. We demonstrate the usage of the BankAccount class by creating an instance, depositing and withdrawing funds, and retrieving the account balance and number.
5. Write a class that overrides the **str** and **add** magic methods. What will these methods allow you to do?

ANS.Here's an example implementation of a Vector class that overrides the str and add magic methods:

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

def str (self):

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

def add (self, other):

if isinstance(other, Vector):

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

else:

raise TypeError("Unsupported operand type for +")

Example usage v1 = Vector(2, 3) v2 = Vector(4, 5)

print(v1) # Output: Vector(2, 3) print(v2) # Output: Vector(4, 5) v3 = v1 + v2 print(v3) # Output: Vector(6, 8)

try: v4 = v1 + 5 except TypeError as e: print(e) # Output: Unsupported operand type for + In this example:

* + The str method allows you to customize the string representation of Vector objects. When you print a Vector object, Python calls its str method to generate the string

representation.

* + The add method enables you to overload the + operator for Vector objects. When you use the + operator with two Vector objects, Python calls their add method to perform the addition. The method returns a new Vector object representing the sum of the

two input vectors.

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

ANS.Here's an example implementation of a decorator that measures and prints the execution time of a function:

import time from functools import wraps

def timer\_decorator(func): @wraps(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"Function {func. name } executed in {execution\_time:.4f} seconds") return result return wrapper

Example usage @timer\_decorator def example\_function(): time.sleep(1) # Simulate some work print("Function executed")

example\_function() Output:

Function executed Function example\_function executed in 1.0011 seconds In this example:

1. We define a decorator function timer\_decorator that takes a function func as input.
2. Inside the decorator, we define a wrapper function that measures the execution time of the input function func.
3. The wrapper function uses the time.time() function to record the start and end times of the function execution.
4. The execution time is calculated by subtracting the start time from the end time.
5. The wrapper function prints the execution time and returns the result of the input function func.
6. We apply the timer\_decorator to an example function example\_function using the @timer\_decorator syntax.
7. When we call the decorated example\_function, it executes and prints the execution time
8. Explain the concept of the Diamond Problem in multiple inheritance. How does Python resolve it?

ANS.The Diamond Problem is a well-known issue that arises in multiple inheritance, where a class inherits from two classes that share a common base class. This creates a diamond-shaped inheritance graph, with the common base class at the top, the two intermediate classes in the middle, and the subclass at the bottom.

Here's an example of the Diamond Problem in Python:

A

/ B C / D

In this example, class D inherits from both classes B and C, which in turn inherit from class A. This creates a diamond-shaped inheritance graph.

The problem arises when classes B and C override a method from class A, and class D inherits from both classes B and C. In this case, class D will have two conflicting versions of the overridden method.

Python resolves the Diamond Problem using a technique called C3 Linearization, also known as the "Method Resolution Order" (MRO). The MRO is a standard for resolving the order of methods in a class hierarchy.

Here's how Python's MRO resolves the Diamond Problem:

1. List the classes in the inheritance graph, starting from the subclass (D) and moving up to the common base class (A).
2. Remove any duplicates from the list, preserving the original order.
3. If a class has already been visited, skip it and move on to the next class. Using this algorithm, Python's MRO resolves the Diamond Problem as follows: D -> B -> C -> A

This means that when class D looks for a method, it will first look in class B, then in class C, and finally in class A.

Here's some example code that demonstrates the Diamond Problem and Python's MRO: class A: def method(self): print("A's method")

class B(A): def method(self): print("B's method") class C(A): def method(self): print("C's method") class D(B, C): pass

d = D() d.method() # Output: B's method

In this example, class D inherits from both classes B and C, which in turn inherit from class A. When we call the method() on an instance of class D, Python's MRO resolves the method call to class B's implementation of the method().

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

ANS.Here's an example implementation of a class method that keeps track of the number of instances created from a class:

class InstanceTracker: num\_instances = 0

def init (self):

InstanceTracker.num\_instances += 1

@classmethod

def get\_num\_instances(cls):

return cls.num\_instances

Example usage print(InstanceTracker.get\_num\_instances()) # Output: 0

obj1 = InstanceTracker() print(InstanceTracker.get\_num\_instances()) # Output: 1 obj2 = InstanceTracker() print(InstanceTracker.get\_num\_instances()) # Output: 2 obj3 = InstanceTracker() print(InstanceTracker.get\_num\_instances()) # Output: 3 In this example:

1. We define a class InstanceTracker with a class attribute num\_instances initialized to 0.
2. In the **init** method, we increment the num\_instances attribute by 1 each time a new instance is created.
3. We define a class method get\_num\_instances that returns the current value of num\_instances.
4. We demonstrate the usage of the InstanceTracker class by creating multiple instances and printing the number of instances created after each instantiation.
5. Implement a static method in a class that checks if a given year is a leap year.

ANS.Here's an example implementation of a static method in a class that checks if a given year is a leap year:

class DateUtils: @staticmethod def is\_leap\_year(year): return year % 4 == 0 and (year % 100 != 0 or year % 400 == 0)

Example usage print(DateUtils.is\_leap\_year(2020)) # Output: True print(DateUtils.is\_leap\_year(2019)) # Output: False print(DateUtils.is\_leap\_year(2000)) # Output: True print(DateUtils.is\_leap\_year(1900)) # Output: False

In this example:

1. We define a class DateUtils with a static method is\_leap\_year.
2. The is\_leap\_year method takes an integer year as input and returns a boolean indicating whether the year is a leap year.
3. The method uses the standard leap year rules: a year is a leap year if it is divisible by 4, but not by 100, unless it is also divisible by 400.
4. We demonstrate the usage of the is\_leap\_year method by calling it with different year values and printing the results.