oops_Assignment_2_New

November 26, 2023

[ ]:
```
1.What is a constructor in Python? Explain its purpose and usage.

Constructors are generally used for instantiating an object.
The task of constructors is to initialize(assign values) to the data members of␣
 ↪the class when an object of the class is created.
In Python the __init__() method is called the constructor and is always called␣
 ↪when an object is created.



def __init__(self):
    # body of the constructor


Types of constructors :

(a)default constructor :The default constructor is simple constructor which␣
 ↪doesn't accept any arguments.It's definition has only one argument which is␣
 ↪a reference to the instance being constructed.
(b)parameterized constructor :constructor with parameters is known as␣
 ↪parameterized constructor.The parameterized constructor take its first␣
 ↪argument as a reference to the instance being constructed known as self and␣
 ↪the rest of the arguments are provided by the programmer.
```

[ ]:
```
2.Differentiate between a parameterless constructor and a parameterized␣
 ↪constructor in Python.
(a)Parameterized Constructor: A parameterized constructor is a constructor that␣
 ↪takes one or more arguments. It is used to create an object with custom␣
 ↪values for its attributes.
Parameterized constructors are useful when you want to create an object with␣
 ↪custom values for its attributes. They allow you to specify the values of␣
 ↪the object's attributes when the object is created, rather than using␣
 ↪default values.
(b)Non-Parameterized Constructor: A non-parameterized constructor is a␣
 ↪constructor that does not take any arguments. It is a special method in␣
 ↪Python that is called when you create an instance of a class.
```

```
The non-parameterized constructor is used to initialize the default values for
 ↪the instance variables of the object.
There is not necessarily a need for a non-parameterized constructor in Python.
 ↪It is up to the programmer to decide whether to include a non-parameterized
 ↪constructor in a class.
```

[ ]:
```
3.How do you define a constructor in a Python class? Provide an example.
Constructors are generally used for instantiating an object.The task of
 ↪constructors is to initialize(assign values) to the data members of the
 ↪class when an object of class is created.In Python the __init__() method is
 ↪called the constructor and is always called when an object is created.

Syntax of constructor declaration :

def __init__(self):
# body of the constructor

Example:
    class Employee:
    def __init__(self, name, id):
        self.id = id
        self.name = name

    def display(self):
        print("ID: %d \nName: %s" % (self.id, self.name))


emp1 = Employee("John", 101)
emp2 = Employee("David", 102)

# accessing display() method to print employee 1 information

emp1.display()

# accessing display() method to print employee 2 information
emp2.display()
```

[ ]:
```
4.Explain the `__init__` method in Python and its role in constructors.
Constructors are generally used for instantiating an object. The task of
 ↪constructors is to initialize(assign values) to the data members of the
 ↪class when an object of the class is created.
In Python the __init__() method is called the constructor and is always called
 ↪when an object is created.
```

[ ]:
```
5. In a class named `Person`, create a constructor that initializes the `name`
 ↪and `age` attributes. Provide an
example of creating an object of this class.
```

```
 class Person:
    def __init__(self, name, age):
        self.name= name
        self.age = age

    def display(self):
        print("Name: %s \nAge: %s" % (self.name, self.age))


per1 = Person("John", 40)
per2 = Person("David", 45)
```

[ ]:
```
6.How can you call a constructor explicitly in Python? Give an example.
Constructors are always called when an object is created and is simulated by␣
 ↪the __init__() method. It accepts the self-keyword,
which refers to itself (the object), as a first argument which allows accessing␣
 ↪the attributes or method of the class.

class Student:
  def __init__(self, name, roll_no)
    self.name = name
    self.roll_no = roll_no

  def display(self):
    print ("Roll No.: %d \nName: %s" % (self.roll_no, self.name))

# Creating object of the class
stud1 = Student("Alex", 34)
stud2 = Student("Mark", 67)

stud1.display()
stud2.display()
```

[ ]:
```
7.What is the significance of the `self` parameter in Python constructors?␣
 ↪Explain with an example.
 self is not a keyword rather it is more of a coding convention. It represents␣
 ↪the instance or objects of a class and binds the attributes of a class with␣
 ↪specific arguments.
The use of self variable in Python helps to differentiate between the instance␣
 ↪attributes (and methods) and local variables.

The self variable is used to represent the instance of the class which is often␣
 ↪used in object-oriented programming. It works as a reference to the object.␣
 ↪Python uses the self parameter to refer to instance attributes and methods␣
 ↪of the class.
```

```
Unlike other programming languages, Python does not use the "@" syntax to␣
 ↪access the instance attributes. This is the sole reason why you need to use␣
 ↪the self variable in Python.
```

[ ]:
```
8.Discuss the concept of default constructors in Python. When are they used?
 A default constructor is a constructor that takes no arguments.
It is used to create an object with default values for its attributes.
```

[ ]:
```
#9.Create a Python class called `Rectangle` with a constructor that initializes␣
 ↪the `width` and `height`
#attributes. Provide a method to calculate the area of the rectangle.

class rect():
    def __init__(self,width,height):
        self.width=width
        self.height=height
    def area(self):
        return self.width*self.height

a=int(input("Enter length of rectangle: "))
b=int(input("Enter breadth of rectangle: "))
obj=rect(a,b) # Creating an object 'obj' of class rect

print("Area of rectangle:",obj.area())
```

[ ]:
```
10.How can you have multiple constructors in a Python class? Explain with an␣
 ↪example.

The technique of having two (or more) constructors in a class is known as␣
 ↪constructor overloading. A class can have multiple constructors that differ␣
 ↪in the number and/or type of their parameters.
It's not, however, possible to have two constructors with the exact same␣
 ↪parameters.

public Person(String name) {
        this.name = name;
        this.age = 0;
        this.weight = 0;
        this.height = 0;
    }

public Person(String name, int age) {
    this.name = name;
    this.age = age;
    this.weight = 0;
    this.height = 0;
```

```
}


Let's make an alternative constructor. The old constructor can remain in place.

public Person(String name) {
        this.name = name;
        this.age = 0;
        this.weight = 0;
        this.height = 0;
    }


We now have two alternative ways to create objects:

public static void main(String[] args) {
    Person paul = new Person("Paul", 24);
    Person ada = new Person("Ada");

    System.out.println(paul);
    System.out.println(ada);
}
```

[ ]:
```
11.What is method overloading, and how is it related to constructors in Python?

The technique of having two (or more) constructors in a class is known as␣
 ↪constructor overloading. A class can have multiple constructors that differ␣
 ↪in the number and/or type of their parameters.
It's not, however, possible to have two constructors with the exact same␣
 ↪parameters.

public Person(String name) {
        this.name = name;
        this.age = 0;
        this.weight = 0;
        this.height = 0;
    }

public Person(String name, int age) {
    this.name = name;
    this.age = age;
    this.weight = 0;
    this.height = 0;
}


Let's make an alternative constructor. The old constructor can remain in place.
```

```
public Person(String name) {
        this.name = name;
        this.age = 0;
        this.weight = 0;
        this.height = 0;
    }


We now have two alternative ways to create objects:

public static void main(String[] args) {
    Person paul = new Person("Paul", 24);
    Person ada = new Person("Ada");

    System.out.println(paul);
    System.out.println(ada);
}
```

[ ]:
```
12.Explain the use of the `super()` function in Python constructors. Provide an␣
 ↪example.

In Python, the super() function is used to refer to the parent class or␣
 ↪superclass. It allows you to call methods defined in the superclass from the␣
 ↪subclass, enabling you to extend and customize the functionality inherited␣
 ↪from the parent class.

Syntax of super() in Python
Syntax: super()

Return : Return a proxy object which represents the parent's class.

super() function in Python Example
In the given example, The Emp class has an __init__ method that initializes the␣
 ↪id, and name and Adds attributes. The Freelance class inherits from the Emp␣
 ↪class and adds an additional attribute called Emails. It calls the parent␣
 ↪class's __init__ method super() to initialize the inherited attribute.

class Emp():
    def __init__(self, id, name, Add):
        self.id = id
        self.name = name
        self.Add = Add

# Class freelancer inherits EMP
class Freelance(Emp):
    def __init__(self, id, name, Add, Emails):
```

```
        super().__init__(id, name, Add)
        self.Emails = Emails

Emp_1 = Freelance(103, "Suraj kr gupta", "Noida" , "KKK@gmails")
print('The ID is:', Emp_1.id)
print('The Name is:', Emp_1.name)
print('The Address is:', Emp_1.Add)
print('The Emails is:', Emp_1.Emails)
```

[ ]:
```
#13. Create a class called `Book` with a constructor that initializes the␣
 ↪`title`, `author`, and `published_year`
#attributes. Provide a method to display book details.

class Book():
    def __init__(self,title,author,published_year):
        self.title=title
        self.author=author
        self.published_year=published_year
    def display(self):
        print(self.title)
        print(self.author)
        print(self.published_year)

title=input("Enter title of Book: ")
author=input("Enter author of Book: ")
published_year=input("Enter published year of Book: ")
obj=Book(title,author,published_year)

print("Book Details:",obj.display())
```

[ ]:
```
14. Discuss the differences between constructors and regular methods in Python␣
 ↪classes.

Constructors
A Constructor is a block of code that initializes a newly created object.
 ↪
A Constructor can be used to initialize an object.
A Constructor is invoked implicitly by the system.
A Constructor is invoked when a object is created using the keyword new.
A Constructor doesn't have a return type.
A Constructor initializes a object that doesn't exist.
A Constructor's name must be same as the name of the class.
A class can have many Constructors but must not have the same parameters.
 ↪        A class can have many methods but must not have the same parameters.
A Constructor cannot be inherited by subclasses.

Methods:
```

```
A Method is a collection of statements which returns a value upon its execution.
A Method consists of Java code to be executed.
A Method is invoked by the programmer.
A Method is invoked through method calls.
A Method must have a return type.
A Method does operations on an already created object.
A Method's name can be anything.
A Method can be inherited by subclasses.
```

[ ]:
```
15.Explain the role of the `self` parameter in instance variable initialization␣
 ↪within a constructor.
class MyClass:
    def __init__(self, arg1, arg2):
        self.arg1 = arg1
        self.arg2 = arg2

    def my_method(self):
        # accessing instance members using self
        print(self.arg1, self.arg2)
Copy code
In the example above, "self" is used in the constructor method (__init__) and␣
 ↪instance method (my_method). In the constructor, "self" is used to␣
 ↪initialize instance variables (arg1 and arg2) with the values passed as␣
 ↪arguments. In the my_method, "self" is used to access and print the values␣
 ↪of arg1 and arg2.

It's important to note that "self" is a convention and not a keyword in Python.␣
 ↪It's possible to use any other variable name in place of "self," but it's␣
 ↪not recommended for readability and maintainability reasons.
```

19.What is the purpose of the `__del__` method in Python classes, and how does it relate to constructors? In Python, the **del**() method is referred to as a destructor method. It is called after an object's garbage collection occurs, which happens after all references to the item have been destroyed.

Example

class Student: # Initializing def **init**(self): print('Student table created.')

```
# Deleting (Calling destructor)
def __del__(self):
      print('Destructor called, Student table deleted.')
```

Stud1 = Student() del Stud1

Destructors are called when an object gets destroyed. In Python, destructors are not needed as much as in C++ because Python has a garbage collector that handles memory management automatically. The **del**() method is a known as a destructor method in Python. It is called when all references to the object have been deleted i.e when an object is garbage collected.

[ ]:
```
#20Create a Python class called `Car` with a default constructor that␣
 ↪initializes the `make` and `model`
#attributes. Provide a method to display car information.

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

    def display(self):
        print(self.make)
        print(self.model)


title=input("Enter make of Car: ")
author=input("Enter model of Car: ")

obj=Book(make,model)

print("Book Details:",obj.display())
```

[ ]:
```
19.Explain the use of constructor chaining in Python. Provide a practical␣
 ↪example.
Constructors are used for instantiating an object. The task of the constructor␣
 ↪is to assign value to data members when an object of the class is created.

Constructor chaining is the process of calling one constructor from another␣
 ↪constructor. Constructor chaining is useful when you want to invoke multiple␣
 ↪constructors, one after another, by initializing only one instance.

In Python, constructor chaining is convenient when we are dealing with␣
 ↪inheritance. When an instance of a child class is initialized, the␣
 ↪constructors of all the parent classes are first invoked and then, in the␣
 ↪end, the constructor of the child class is invoked.

class Vehicle:
    # Constructor of Vehicle
    def __init__(self, engine):
        print('Inside Vehicle Constructor')
        self.engine = engine

class Car(Vehicle):
    # Constructor of Car
    def __init__(self, engine, max_speed):
        super().__init__(engine)
        print('Inside Car Constructor')
        self.max_speed = max_speed
```

```
class Electric_Car(Car):
    # Constructor of Electric Car
    def __init__(self, engine, max_speed, km_range):
        super().__init__(engine, max_speed)
        print('Inside Electric Car Constructor')
        self.km_range = km_range

# Object of electric car
ev = Electric_Car('1500cc', 240, 750)
print(f'Engine={ev.engine}, Max Speed={ev.max_speed}, Km range={ev.km_range}')
```

[ ]:
```
16.How do you prevent a class from having multiple instances by using␣
 ↪constructors in Python? Provide an
example.
Sometimes, developers need to enforce non-instantiability to a class, i.e., so␣
 ↪class objects cannot be instantiated.

For example, utility classes are classes that contain utility methods where an␣
 ↪instance of the class is not needed. Instead, the methods of the class can␣
 ↪be made static so that the methods can be accessed using the class name.

We can make a class non-instantiable by making the constructor of the class␣
 ↪private. By making the constructor private, the following happens:

The constructor becomes inaccessible outside the class, so class objects can't␣
 ↪be created.
The class can't be made a parent class by extending it, as the constructor␣
 ↪can't be called from the subclass.
```

[ ]:
```
21.What is inheritance in Python? Explain its significance in object-oriented␣
 ↪programming.

Inheritance allows us to define a class that inherits all the methods and␣
 ↪properties from another class.

Parent class is the class being inherited from, also called base class.

Child class is the class that inherits from another class, also called derived␣
 ↪class.

Inheritance provides code reusability, abstraction, etc. Because of␣
 ↪inheritance, we can even inherit abstract classes, classes with␣
 ↪constructors, etc.
For example - Beagle, Pitbull, etc., are different breeds of dogs, so they all␣
 ↪have inherited the properties of class dog.
```

[ ]:
```
22.Differentiate between single inheritance and multiple inheritance in Python.␣
 ↪Provide examples for each.

Types of Inheritance depend upon the number of child and parent classes␣
 ↪involved. There are four types of inheritance in Python:

(A)Single Inheritance:
Single inheritance enables a derived class to inherit properties from a single␣
 ↪parent class, thus enabling code reusability and the addition of new␣
 ↪features to existing code.

# Python program to demonstrate
# single inheritance

# Base class
class Parent:
    def func1(self):
        print("This function is in parent class.")

# Derived class


class Child(Parent):
    def func2(self):
        print("This function is in child class.")


# Driver's code
object = Child()
object.func1()
object.func2()

(B)Multiple Inheritance:
When a class can be derived from more than one base class this type of␣
 ↪inheritance is called multiple inheritances. In multiple inheritances, all␣
 ↪the features of the base classes are inherited into the derived class.

# Python program to demonstrate
# multiple inheritance

# Base class1
class Mother:
    mothername = ""

    def mother(self):
        print(self.mothername)
```

```
# Base class2


class Father:
    fathername = ""

    def father(self):
        print(self.fathername)

# Derived class


class Son(Mother, Father):
    def parents(self):
        print("Father :", self.fathername)
        print("Mother :", self.mothername)


# Driver's code
s1 = Son()
s1.fathername = "RAM"
s1.mothername = "SITA"
s1.parents()
```

[24]:
```
#23.Create a Python class called `Vehicle` with attributes `color` and `speed`.␣
 ↪Then, create a child class called
#`Car` that inherits from `Vehicle` and adds a `brand` attribute. Provide an␣
 ↪example of creating a `Car` object.


class Child():

    # Constructor
    def __init__(self, name, age):
        self.name=name
        self.age = age



class GrandChild(Child):

    # Constructor
    def __init__(self, name, age, address):
        Child.__init__(self, name, age)
        self.address = address

 # Driver code
```

```
g = GrandChild("Geek1", 23, "Noida")
```

[12]:
```
#27.Create a Python class called `Animal` with a method `speak()`. Then, create child classes `Dog` and `Cat` that inherit from `Animal` and override the `speak()` method. Provide an example of using these classes.

class Animal():


    def speak():
        print("speak() method of class Animal")



class Dog(Animal):
 def speak(self):
        print("speak() method of class Dog")


class Cat(Animal):
 def speak(self):
        print("speak() method of class Cat")


d = Dog()
d.speak()
c = Cat()
c.speak()
```

```
speak() method of class Dog
speak() method of class Cat
```

[ ]:
```
28.Explain the role of the `isinstance()` function in Python and how it relates to inheritance.

At its core, isinstance() is a type-checking function, verifying the type of an object in Python and ensuring your code behaves as expected.

The isinstance() function in Python is a built-in function used for type checking. It verifies if an object is of a specified type, returning True if it is, and False otherwise.

For example:

x = 10
print(isinstance(x, int))
```

```
Python
In this case, isinstance() returns True because x is an integer.

This method "isinstance()" considers inheritance and allows one to check if an␣
 ↪object belongs to a particular hierarchy.
```

[ ]:
```
29.What is the purpose of the `issubclass()` function in Python? Provide an␣
 ↪example.

Python issubclass() is built-in function used to check if a class is a subclass␣
 ↪of another class or not. This function returns True if the given class is␣
 ↪the subclass of given class else it returns False.

# Defining Parent class
class Vehicles:

    # Constructor
    def __init__(vehicleType):
        print('Vehicles is a ', vehicleType)

# Defining Child class
class Car(Vehicles):

    # Constructor
    def __init__(self):
        Vehicles.__init__('Car')

# Driver's code
print(issubclass(Car, Vehicles))#True
```

[ ]:
```
30.Discuss the concept of constructor inheritance in Python. How are␣
 ↪constructors inherited in child classes?

In python, Constructor level inheritance also possible as same way as methods␣
 ↪and variables of a parent class.

 Are the constructors of the base class accessible to the sub class or not,Yes␣
 ↪they are.

 Here, we are taking a super class by the name 'Father' and derived a sub class␣
 ↪'Son' from it. The Father class has a constructor where a variable␣
 ↪'property' is declared and initialized with 800000.00. When Son is created␣
 ↪from Father, this constructor is by default available to Son class. When we␣
 ↪call the method of the super class using sub class object, it will display␣
 ↪the value of the 'property' variable.
```

```
#base class constructor is available to sub class
class Father:
   def __init__(self):
      self.property = 800000.00
   def display_property(self):
      print('Father\'s property=', self.property)
class Son(Father):

s = Son()
s.display_property()
```

[ ]:
```
31.Create a Python class called `Shape` with a method `area()` that calculates
 ↪the area of a shape. Then, create child classes `Circle` and `Rectangle`
 ↪that inherit from `Shape` and implement the `area()` method
accordingly. Provide an example.


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

    def area(self):

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

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

   class Circle(Shape):
    def __init__(self,radius):
    self.radius=radius
    def area(self):
     area=self.radius*self.radius*3.142

a=int(input("Enter length of rectangle: "))
b=int(input("Enter breadth of rectangle: "))
c=int(input("Enter radius of circle: "))
obj1=Rectangle(a,b)
obj2=Circle(c)


print("Area of rectangle:",obj1.area())
```

```
print("Area of rectangle:",obj2.area())
```

[ ]:
```
32.Explain the use of abstract base classes (ABCs) in Python and how they␣
 ↪relate to inheritance. Provide an
example using the `abc` module.

In Python, classes are user-defined blueprints for creating objects that have␣
 ↪attributes and methods. When we define a regular Python class, we can add␣
 ↪any number of attributes and methods to it. We can then create instances of␣
 ↪the class and use them to perform operations.

However, one limitation of regular Python classes is that they do not enforce␣
 ↪the implementation of certain methods or attributes in the classes that␣
 ↪inherit from them. This means that if we have two classes that are related␣
 ↪in some way, there is no guarantee that they will have the same methods or␣
 ↪attributes. As a result, objects of different classes may not be able to be␣
 ↪used interchangeably in our code.

This is where Abstract Base Classes (ABCs) come in. An ABC is a special type of␣
 ↪class that contains one or more abstract methods. Abstract methods are␣
 ↪methods that have no implementation in the ABC but must be implemented in␣
 ↪any class that inherits from the ABC. In other words, an ABC provides a set␣
 ↪of common methods or attributes that its subclasses must implement.

The main difference between an ABC and a regular class is that you cannot␣
 ↪create an instance of an ABC. Instead, you can only inherit from it and␣
 ↪implement all the abstract methods it has defined. This ensures that all␣
 ↪subclasses of the ABC have the same set of methods or attributes, making␣
 ↪them interchangeable in our code.

In summary, ABCs are a way of defining a set of common methods or attributes␣
 ↪that must be implemented by any class that inherits from the ABC. This␣
 ↪promotes code reuse, consistency, and modularity in our code.

class Polygon(ABC):

    @abstractmethod
    def noofsides(self):
        pass


class Triangle(Polygon):

    # overriding abstract method
    def noofsides(self):
        print("I have 3 sides")
```

```
class Pentagon(Polygon):

    # overriding abstract method
    def noofsides(self):
        print("I have 5 sides")


class Hexagon(Polygon):

    # overriding abstract method
    def noofsides(self):
        print("I have 6 sides")


class Quadrilateral(Polygon):

    # overriding abstract method
    def noofsides(self):
        print("I have 4 sides")


# Driver code
R = Triangle()
R.noofsides()

K = Quadrilateral()
K.noofsides()

R = Pentagon()
R.noofsides()

K = Hexagon()
K.noofsides()
```

[ ]:
```
33. How can you prevent a child class from modifying certain attributes or␣
 ↪methods inherited from a parent
class in Python?
```

[ ]:
```
34.Create a Python class called `Employee` with attributes `name` and `salary`.␣
 ↪Then, create a child class
`Manager` that inherits from `Employee` and adds an attribute `department`.␣
 ↪Provide an example.


    class Employee():
```

```
    def __init__(self,name,salary):
        self.name=name
        self.salary=salary



class Manager(Employee):
    def __init__(self,name,salary,department):
        self.width=width
        self.height=height
        self.department=department
```

[ ]:
```
35.Discuss the concept of method overloading in Python inheritance. How does it␣
 ↪differ from method
overriding?

(a)Method Overloading:
    It refers to defining multiple methods with the same name but different␣
 ↪parameters

    It can be achieved in Python using default arguments

    It allows a class to have multiple methods with the same name but different␣
 ↪behaviors based on the input parameters

 (b)Method Overriding:
    It refers to defining a method in a subclass that has the same name as the␣
 ↪one in its superclass

    It can be achieved by defining a method in a subclass with the same name as␣
 ↪the one in its superclass

    It allows a subclass to provide its own implementation of a method defined␣
 ↪in its superclass
```

[ ]:
```
36.Explain the purpose of the `__init__()` method in Python inheritance and how␣
 ↪it is utilized in child classes.
```

```
Constructors are generally used for instantiating an object. The task of␣
 ↪constructors is to initialize(assign values) to the data members of the␣
 ↪class when an object of the class is created. In Python the __init__()␣
 ↪method is called the constructor and is always called when an object is␣
 ↪created.
Syntax of constructor declaration :

def __init__(self):
    # body of the constructor

Inheritance allows the child class to inherit the __init__() method of the␣
 ↪parent class along with the other data members and member functions of that␣
 ↪class.  The __init__ method of the parent or the base class is called within␣
 ↪the __init__ method of the child or sub class. In case the parent class␣
 ↪demands an argument, the parameter value must be passed in the __init__␣
 ↪method of the child class as well as at the time of object creation for the␣
 ↪child class.
```

[ ]:
```
37.Create a Python class called `Bird` with a method `fly()`. Then, create␣
 ↪child classes `Eagle` and `Sparrow` that inherit from `Bird` and implement␣
 ↪the `fly()` method differently. Provide an example of using these
classes.

class Bird():


    def fly(self):
         print("I am a bird")


class Eagle(Bird):


    def fly(self):
        print("I am Eagle")


class Sparrow(Bird):


    def fly(self):
        print("I am Sparrow")
```

[ ]:
```
38.What is the "diamond problem" in multiple inheritance, and how does Python␣
 ↪address it?
```

```
The diamond problem occurs when two classes have a common parent class, and␣
 ↪another class has both those classes as base classes. The diamond problem is␣
 ↪the generally used term for an ambiguity that arises when two classes B and␣
 ↪C inherit from a superclass A, and another class D inherits from both B and␣
 ↪C.

To address this issue, Python provides the super() function. At a basic level,␣
 ↪super() is used to call methods in a superclass. In multiple inheritance␣
 ↪scenarios, it ensures that the method in the base class gets executed only␣
 ↪once.
```

[ ]:
```
39. Discuss the concept of "is-a" and "has-a" relationships in inheritance, and␣
 ↪provide examples of each.

An IS-A relationship is inheritance. The classes which inherit are known as sub␣
 ↪classes or child classes. On the other hand, HAS-A relationship is␣
 ↪composition.

In OOP, IS-A relationship is completely inheritance. This means, that the child␣
 ↪class is a type of parent class. For example, an apple is a fruit. So you␣
 ↪will extend fruit to get apple.

class Apple extends Fruit {

}
On the other hand, composition means creating instances which have references␣
 ↪to other objects. For example, a room has a table. So you will create a␣
 ↪class room and then in that class create an instance of type table.

class Room {

    Table table = new Table();

}
A HAS-A relationship is dynamic (run time) binding while inheritance is a␣
 ↪static (compile time) binding. If you just want to reuse the code and you␣
 ↪know that the two are not of same kind use composition. For example, you␣
 ↪cannot inherit an oven from a kitchen. A kitchen HAS-A oven. When you feel␣
 ↪there is a natural relationship like Apple is a Fruit use inheritance.
```

[ ]:
```
40.Create a Python class hierarchy for a university system. Start with a base␣
 ↪class `Person` and create child
classes `Student` and `Professor`, each with their own attributes and methods.␣
 ↪Provide an example of using
these classes in a university context.
```

[ ]:
```
24.Explain the concept of method overriding in inheritance. Provide a practical␣
 ↪example.

Method overriding is an ability of any object-oriented programming language␣
 ↪that allows a subclass or child class to provide a specific implementation␣
 ↪of a method that is already provided by one of its super-classes or parent␣
 ↪classes. When a method in a subclass has the same name, same parameters or␣
 ↪signature and same return type(or sub-type) as a method in its super-class,␣
 ↪then the method in the subclass is said to override the method in the␣
 ↪super-class.

# Defining parent class
class Parent():

    # Constructor
    def __init__(self):
        self.value = "Inside Parent"

    # Parent's show method
    def show(self):
        print(self.value)

# Defining child class
class Child(Parent):

    # Constructor
    def __init__(self):
        self.value = "Inside Child"

    # Child's show method
    def show(self):
        print(self.value)


# Driver's code
obj1 = Parent()
obj2 = Child()

obj1.show()
obj2.show()
```

[ ]:
```
25.How can you access the methods and attributes of a parent class from a child␣
 ↪class in Python? Give an
example.

To access parent class attributes in a child class:
```

```
Use the super() method to call the constructor of the parent in the child.
The __init__() method will set the instance variables.
Access any of the parent class's attributes or methods on the self object.


class Employee():
    cls_id = 'emp-cls'

    def __init__(self, name):
        self.salary = 100
        self.name = name


class Developer(Employee):
    def __init__(self, name):
        #  invoke parent __init__() method
        super().__init__(name)

        #  accessing parent instance variable
        print(self.salary)  #  100

        #  accessing parent class variable
        print(self.cls_id)  #  emp-cls


d1 = Developer('bobbyhadz')
print(d1.salary)  #  100

print(d1.cls_id)  #  'emp-cls'
```

[ ]:
```
26.Discuss the use of the `super()` function in Python inheritance. When and␣
 ↪why is it used? Provide an
example.

The super() function is used to give access to methods and properties of a␣
 ↪parent or sibling class.

The super() function returns an object that represents the parent class.

class Parent:
   def __init__(self):
      self.parent_attribute = "I'm from the parent class"

class Child(Parent):
   def __init__(self):
      super().__init__()
      self.child_attribute = "I'm from the child class"
```

```
child = Child()
print(child.parent_attribute)  # Accessing parent class attribute using instance
```

[ ]:
```
41.Explain the concept of encapsulation in Python. What is its role in␣
↪object-oriented programming?
Encapsulation is one of the fundamental concepts in object-oriented programming␣
↪(OOP). It describes the idea of wrapping data and the methods that work on␣
↪data within one unit. This puts restrictions on accessing variables and␣
↪methods directly and can prevent the accidental modification of data. To␣
↪prevent accidental change, an object's variable can only be changed by an␣
↪object's method. Those types of variables are known as private variables.

A class is an example of encapsulation as it encapsulates all the data that is␣
↪member functions, variables, etc. The goal of information hiding is to␣
↪ensure that an object's state is always valid by controlling access to␣
↪attributes that are hidden from the outside world.
```

[ ]:
```
42.Describe the key principles of encapsulation, including access control and␣
↪data hiding.

Encapsulation is the process of bundling data members and methods inside a␣
↪single class. Bundling similar data elements inside a class also helps in␣
↪data hiding. Encapsulation also ensures that objects of a class can function␣
↪independently.

Why use Encapsulation?
Consider an analogy of a car - I'm sure most of us enjoy going on long drives␣
↪and generally enjoy driving our cars. We know how to change gears, use the␣
↪steering wheel, apply breaks, or accelerate as and when required.

But not most of us need to know how cars are manufactured. We do not know how␣
↪the parts were acquired or how long it took to design that model. All these␣
↪factors are irrelevant to us, and what matters is the final product only.

This is basically how Encapsulation works - Python codes encapsulate and hide␣
↪backend implementation details by restricting the permissions. However, this␣
↪does not hinder the program's execution in any manner. Thus, Encapsulation␣
↪facilitates Data Abstraction.
```

[ ]:
```
43.How can you achieve encapsulation in Python classes? Provide an example.
```

```
Encapsulation is one of the fundamental concepts in object-oriented programming␣
 ↪(OOP). It describes the idea of wrapping data and the methods that work on␣
 ↪data within one unit. This puts restrictions on accessing variables and␣
 ↪methods directly and can prevent the accidental modification of data. To␣
 ↪prevent accidental change, an object's variable can only be changed by an␣
 ↪object's method. Those types of variables are known as private variables.

A class is an example of encapsulation as it encapsulates all the data that is␣
 ↪member functions, variables, etc. The goal of information hiding is to␣
 ↪ensure that an object's state is always valid by controlling access to␣
 ↪attributes that are hidden from the outside world.

Consider a real-life example of encapsulation, in a company, there are␣
 ↪different sections like the accounts section, finance section, sales section␣
 ↪etc. The finance section handles all the financial transactions and keeps␣
 ↪records of all the data related to finance. Similarly, the sales section␣
 ↪handles all the sales-related activities and keeps records of all the sales.␣
 ↪Now there may arise a situation when due to some reason an official from the␣
 ↪finance section needs all the data about sales in a particular month. In␣
 ↪this case, he is not allowed to directly access the data of the sales␣
 ↪section. He will first have to contact some other officer in the sales␣
 ↪section and then request him to give the particular data. This is what␣
 ↪encapsulation is
```

[ ]:
```
44.Discuss the difference between public, private, and protected access␣
 ↪modifiers in Python.

A Class in Python has three types of access modifiers:

Public Access Modifier
Protected Access Modifier
Private Access Modifier


Public Access Modifier:
The members of a class that are declared public are easily accessible from any␣
 ↪part of the program. All data members and member functions of a class are␣
 ↪public by default.

Protected Access Modifier:
The members of a class that are declared protected are only accessible to a␣
 ↪class derived from it. Data members of a class are declared protected by␣
 ↪adding a single underscore '_' symbol before the data member of that class.

Private Access Modifier:
```

```
The members of a class that are declared private are accessible within the␣
 ↪class only, private access modifier is the most secure access modifier. Data␣
 ↪members of a class are declared private by adding a double underscore '__'␣
 ↪symbol before the data member of that class.
```

[ ]:
```
45.Create a Python class called `Person` with a private attribute `__name`.␣
 ↪Provide methods to get and set the
name attribute.

class Person:

    def __init__(self, a):
        ## private varibale or property in Python
        self.__a = a

    ## getter method to get the properties using an object
    def get_a(self):
        return self.__name

    ## setter method to change the value 'a' using an object
    def set_a(self, a):
        self.__name = __name
```

[ ]:
```
46.Explain the purpose of getter and setter methods in encapsulation. Provide␣
 ↪examples.

Getter and Setter in Python differ from those in other Object Oriented␣
 ↪Languages. They are used in Python to achieve data encapsulation.

What is Getter() in Python?
Getters are OOPs functions in python. Getters are functions used to retrieve␣
 ↪the values of private attributes, while setters are functions used to modify␣
 ↪or assign values to private attributes.

In Python, since private variables cannot be accessed directly, the getter␣
 ↪method provides a way to retrieve them from outside the class, thereby␣
 ↪helping in data encapsulation. The getter method typically allows read-only␣
 ↪access to the properties of an object. Getters are named with a 'get_'␣
 ↪prefix followed by the attribute name whose value is to be accessed.

What is Setter() in Python?
Setters are methods in OOPs that are used to modify and update the value of an␣
 ↪attribute of an object. Certain constraints can also be applied along with␣
 ↪setter methods. This ensures that the new values assigned to the attributes␣
 ↪are valid. Setter() are used together with the getter() method in Python.
```

[ ]:
```
47.What is name mangling in Python, and how does it affect encapsulation?

Name mangling in Python is the process of an identifier with two leading␣
 ↪underscores getting replaced with the class name followed by the identifier,␣
 ↪e.g. _classname__identifier.

Where class name is the name of the current class.

It means that any identifier of the form __var_name – at least two leading␣
 ↪underscores or at most one trailing underscore – is replaced with␣
 ↪_classname__var_name.

What is the Use of Name Mangling in Python
Name mangling in Python is used to restrict private class attributes or methods␣
 ↪from being accessed from outside the class.

Most high-level programming languages have a way of making the attributes and␣
 ↪methods of the class to be private or protected.

Python does not have an access modifier, hence, you cannot restrict the class␣
 ↪attributes and methods from getting access from outside the class.

To partially implement this, Python uses the concept of Name Mangling
```

[ ]:
```
48.Create a Python class called `BankAccount` with private attributes for the␣
 ↪account balance (`__balance`)

class BankAccount:
    # create the constuctor with parameters: accountNumber, name and balance
    def __init__(self,accountNumber, name, balance):
        self.__account_number = accountNumber

        self.__balance = __balance

    # create Deposit() method
    def Deposit(self , d ):
        self.__balance = self.__balance + d

    # create Withdrawal method
    def Withdrawal(self , w):
        if(self.__balance < w):
            print("impossible operation! Insufficient balance !")
        else:
            self.__balance = self.__balance - w
```

[ ]:
```
49.Discuss the advantages of encapsulation in terms of code maintainability and␣
 ↪security.

Encapsulation is one of the fundamental concepts of object-oriented programming␣
 ↪and helps to improve code maintainability and security in several ways:

Improved Maintainability: Encapsulation helps to reduce the complexity of the␣
 ↪code by hiding the implementation details of objects and exposing only the␣
 ↪necessary information to the outside world through well-defined interfaces.␣
 ↪This makes the code easier to understand, maintain, and extend, as changes␣
 ↪to the implementation can be made without affecting other parts of the␣
 ↪system.

Better Security: Encapsulation helps to improve security by hiding the␣
 ↪implementation details of objects and exposing only the necessary␣
 ↪information through well-defined interfaces. This makes it more difficult␣
 ↪for attackers to access and exploit the implementation details of objects,␣
 ↪reducing the risk of security vulnerabilities.
```

[ ]:
```
50.How can you access private attributes in Python? Provide an example␣
 ↪demonstrating the use of name
mangling.

The name mangling process helps to access the class variables from outside the␣
 ↪class. The class variables can be accessed by adding _classname to it. The␣
 ↪name mangling is closest to private not exactly private.

class Student:
    def __init__(self, name):
        self.__name = name

s1 = Student("Santhosh")
print(s1._Student__name)

The above class variable is accessed by adding the _classname to it. The class␣
 ↪variable is accessed from outside the class with the name _Student__name.
```

[ ]:
```
51.Create a Python class hierarchy for a school system, including classes for␣
 ↪students, teachers, and courses,
and implement encapsulation principles to protect sensitive information.
```

[ ]:
```
52.Explain the concept of property decorators in Python and how they relate to␣
 ↪encapsulation.
```

```
@property decorator is a built-in decorator in Python which is helpful in␣
 ↪defining the properties effortlessly without manually calling the inbuilt␣
 ↪function property(). Which is used to return the property attributes of a␣
 ↪class from the stated getter, setter and deleter as parameters.

class Portal:

    # Defining __init__ method
    def __init__(self):
        self.__name =''

    # Using @property decorator
    @property

    # Getter method
    def name(self):
        return self.__name

    # Setter method
    @name.setter
    def name(self, val):
        self.__name = val

    # Deleter method
    @name.deleter
    def name(self):
       del self.__name

# Creating object
p = Portal();

# Setting name
p.name = 'GeeksforGeeks'

# Prints name
print (p.name)

# Deletes name
del p.name

# As name is deleted above this
# will throw an error
print (p.name)
```

[ ]:
```
53.What is data hiding, and why is it important in encapsulation? Provide␣
 ↪examples.
```

```
Data hiding in Python is done by using a double underscore before (prefix) the␣
 ↪attribute name. This makes the attribute private/ inaccessible and hides␣
 ↪them from users. Python has nothing secret in the real sense. Still, the␣
 ↪names of private methods and attributes are internally mangled and unmangled␣
 ↪on the fly, making them inaccessible by their given names.

In Python, the process of encapsulation and data hiding works simultaneously.␣
 ↪Data encapsulation hides the private methods on the other hand data hiding␣
 ↪hides only the data components. The robustness of the data is also increased␣
 ↪with data hiding. The private access specifier is used to achieve data␣
 ↪hiding. There are three types of access specifiers, private, public, and␣
 ↪protected.


class JustCounter:

   __secretCount = 0

   def count(self):

      self.__secretCount += 1

      print self.__secretCount

counter = JustCounter()

counter.count()

counter.count()

print counter.__secretCount
```

[ ]:
```
54.Create a Python class called `Employee` with private attributes for salary␣
 ↪(`__salary`) and employee ID

class Employee:

    def __init__(self,accountNumber, salary, empId):
        self.__accountNumber = accountNumber

        self.__salary = __salary


    # Bonus Calculation
    def calculateBonus(self , w):
        current_year=int(input("Enter the Current Year :"))
        join_year=int(input("Enter the Year of Joining :"))
```

```
        diff=current_year-join_year
        if(diff>3):
             print("Bonus of Rs : 2500 /-");
       else:
            print("No Bonus..")
```

[ ]:
```
56.What are the potential drawbacks or disadvantages of using encapsulation in␣
 ↪Python?

Advantages of Encapsulation
Data Protection: The program runner will not be able to identify or see which␣
 ↪methods are present in the code. Therefore he/she doesn't get any chance to␣
 ↪change any specific variable or data and hinder the running of the program.

Flexibility: The code which is encapsulated looks more cleaner and flexible,␣
 ↪and can be changed as per the needs. We can change the code read-only or␣
 ↪write-only by getter and setter methods. This also helps in debugging the␣
 ↪code if needed.

Reusability: The methods can be changed and the code is reusable.

Disadvantages of Encapsulation
Code Size:The length of the code increases drastically in the case of␣
 ↪encapsulation as we need to provide all the methods with the specifiers.

More Instructions: As the size of the code increases, therefore, you need to␣
 ↪provide additional instructions for every method.

Increased code execution: Encapsulation results in an increase in the duration␣
 ↪of the program execution. It is because more instructions are added to the␣
 ↪code therefore they require more time to execute.
```

[ ]:
```
57.Create a Python class for a library system that encapsulates book␣
 ↪information, including titles, authors,
and availability status.


class Library:

    def __init__(self,title, author,status):
        self.__title = title
        self.__author = author
        self.__status = __status
```

[ ]:
```
58.Explain how encapsulation enhances code reusability and modularity in Python␣
↪programs.

Encapsulation is one of the critical features of object-oriented programming,␣
↪which involves the bundling of data members and functions inside a single␣
↪class. Bundling similar data members and functions inside a class also helps␣
↪in data hiding. Encapsulation also ensures that objects are self-sufficient␣
↪functioning pieces and can work independently.

It is easier to reuse code when it is encapsulated. Creating objects that␣
↪contain common functionality allows us to reuse them in many settings.

Encapsulation encourages developers to design classes that are cohesive and␣
↪have a single responsibility, which makes the code more modular and easier␣
↪to maintain.
```

[ ]:
```
59.Describe the concept of information hiding in encapsulation. Why is it␣
↪essential in software development?

Data hiding is also known as information hiding or data encapsulation. The data␣
↪encapsulation is done to hide the application implementation details from␣
↪its users. As the intention behind both is the same, encapsulation is also␣
↪known as data hiding. When a data member is mentioned as private in the␣
↪class, it is accessible only within the same class and inaccessible outside␣
↪that class.

The feature of data hiding, ides the feature of internal data. The feature␣
↪prevents free access and the access is given to limited access. There are␣
↪various benefits to having a data hiding feature, one of those is preventing␣
↪the vulnerability of the data and safeguarding it from potential breaches.

In Python, the data hiding isolates the features, data, class, program, etc␣
↪from the users. The users do not get free access. This feature of data␣
↪hiding enhances the security of the system and initiates better reliability.␣
↪Only a few or very specific people get access.
```

[ ]:
```
60.Create a Python class called `Customer` with private attributes for customer␣
↪details like name, address,
and contact information. Implement encapsulation to ensure data integrity and␣
↪security.


class Customer:
```

```
    def __init__(self,title, author,status):
        self.__name = name
        self.__address = address
        self.__cinfo = __cinfo
```

[ ]:
```
61.What is polymorphism in Python? Explain how it is related to object-oriented␣
 ↪programming.

The word polymorphism means having many forms. In programming, polymorphism␣
 ↪means the same function name (but different signatures) being used for␣
 ↪different types. The key difference is the data types and number of␣
 ↪arguments used in function.

Every object-oriented programming language must include the fundamental concept␣
 ↪of polymorphism, which is inseparable from OOPs. An object or reference can␣
 ↪have distinct forms in various contexts. Polymorphism would be defined as "a␣
 ↪property of having numerous forms" since, as the name suggests, "poly"␣
 ↪denotes "many," and "morph" refers to "forms."

A single interface in the object-oriented programming language handles both␣
 ↪classes and objects. It puts virtual functions, overriding, and function␣
 ↪overloading into practice. Additionally, it is frequently employed in␣
 ↪programming to instrument inheritance.

One of the important OOPS ideas is polymorphism. You can have different or␣
 ↪numerous kinds of objects and variables or use polymorphism methods.␣
 ↪Polymorphism allows for different implementations of the same method␣
 ↪depending on the requirements of the class.
```

[ ]:
```
62.Describe the difference between compile-time polymorphism and runtime␣
 ↪polymorphism in Python.

Polymorphism in Java is classified into command-line diversity & run time␣
 ↪polymorphism. Method overloading is used to achieve compile time␣
 ↪polymorphism. Because call resolution for overloaded methods occurs at build␣
 ↪time, it is known called compile time and static polymorphism. Parameters␣
 ↪determine whether the overloaded version of the method is invoked. Dynamic␣
 ↪polymorphism, on the other hand, refers to runtime polymorphism. Because␣
 ↪method call resolution does not occur at build time; rather, it occurs at␣
 ↪runtime when actual objects are formed. It is accomplished by method␣
 ↪overriding. At runtime, every type of object determines which override␣
 ↪method is called.
```

```
1.Compile Time Polymorphism:
Method overloading produces compile-time polymorphism. The phrase method␣
 ↪overloading refers to the ability to have many methods with the same name.␣
 ↪Because this procedure occurs during build time, it is called Compile-Time␣
 ↪Polymorphism. Build-time polymorphism occurs when an object is coupled,␣
 ↪including its functionality at compile time. By inspecting the method␣
 ↪signatures, Java determines the method to invoke at compile time. As a␣
 ↪result, this is known as compile-time polymorphism, static polymorphism, or␣
 ↪early binding. Method overloading is used to achieve compile-time␣
 ↪polymorphism. Method Overloading states that you can have many functions␣
 ↪that share the same name in the same class, each with a distinct prototype.␣
 ↪Function overloading is a method of achieving polymorphism, although it␣
 ↪depends on technology and the type of polymorphism used. In Java, function␣
 ↪overloading is accomplished at the compile time.

2.Run-Time Polymorphism:
Runtime polymorphism occurs when an object is associated with functionality␣
 ↪during runtime. Method overriding can be used to provide runtime␣
 ↪polymorphism. The Java virtual machine selects the method to invoke during␣
 ↪runtime, not at compile time. It's also known as dynamic and late binding.␣
 ↪Method overriding indicates that the child class utilises the same function␣
 ↪as the parent class. It indicates that method overriding occurs when a child␣
 ↪class offers a custom implementation of a method offered by another of its␣
 ↪parent classes.
```

[ ]:
```
63.Create a Python class hierarchy for shapes (e.g., circle, square, triangle)␣
 ↪and demonstrate polymorphism
through a common method, such as `calculate_area()`.


# Import the math module to access mathematical functions like pi
import math

# Define a base class called Shape to represent a generic shape with methods␣
 ↪for calculating area and perimeter
class Shape:
    # Placeholder method for calculating area (to be implemented in derived␣
 ↪classes)
    def calculate_area(self):
        pass


#Define a derived class called Circle, which inherits from the Shape class
class Circle(Shape):
    # Initialize the Circle object with a given radius
    def __init__(self, radius):
        self.radius = radius
```

```
    # Calculate and return the area of the circle using the formula:   * r^2
    def calculate_area(self):
        return math.pi * self.radius**2



# Define a derived class called Rectangle, which inherits from the Shape class
class Rectangle(Shape):
    # Initialize the Rectangle object with given length and width
    def __init__(self, length, width):
        self.length = length
        self.width = width

    # Calculate and return the area of the rectangle using the formula: length␣
 ↪* width
    def calculate_area(self):
        return self.length * self.width


# Define a derived class called Triangle, which inherits from the Shape class
class Triangle(Shape):
    # Initialize the Triangle object with a base, height, and three side lengths
    def __init__(self, base, height, side1, side2, side3):
        self.base = base
        self.height = height
        self.side1 = side1
        self.side2 = side2
        self.side3 = side3

    # Calculate and return the area of the triangle using the formula: 0.5 *␣
 ↪base * height
    def calculate_area(self):
        return 0.5 * self.base * self.height
```

[ ]:
```
64.Explain the concept of method overriding in polymorphism. Provide an example.

Polymorphism is most commonly associated with inheritance. In Python, child␣
 ↪classes, like other programming languages, inherit methods and attributes␣
 ↪from the parent class. Method Overriding is the process of redefining␣
 ↪certain methods and attributes to fit the child class. This is especially␣
 ↪handy when the method inherited from the parent class does not exactly fit␣
 ↪the child class. In such circumstances, the method is re-implemented in the␣
 ↪child class. Method Overriding refers to the technique of re-implementing a␣
 ↪method in a child class.

Example
```

```
class Vehicle:
    def __init__(self, brand, model, price):
        self.brand = brand
        self.model = model
        self.price = price

    def show(self):
        print('Details:', self.brand, self.model, 'Price:', self.price)

    def max_speed(self):
        print('Vehicle max speed is 160')

    def gear_system(self):
        print('Vehicle has 6 shifter gearbox')

# inherit from vehicle class
class Car(Vehicle):
    def max_speed(self):
        print('Car max speed is 260')

    def gear_system(self):
        print('Car has Automatic Transmission')

# Car Object
car = Car('Audi', 'R8', 9000000)
car.show()
# call methods from Car class
car.max_speed()
car.gear_system()

# Vehicle Object
vehicle = Vehicle('Nissan', 'Magnite', 550000)
vehicle.show()
# call method from a Vehicle class
vehicle.max_speed()
vehicle.gear_system()
```

[ ]:
```
65.How is polymorphism different from method overloading in Python? Provide␣
↪examples for both.
```

The polymorphism is the base of the OOP, the overloading is one of ways to implement to polymorphism, specially when are involved operators. More generally, speaking about polymorphism when there are two or more classes involved. While the overloading can be made also inside the same class, we can overload the name of a method with several signatures (different list of parameters). While overriding is designed exclusively for involving two or more classes. Note that the overrided methods have all the same signature.

[ ]:
```
66.Create a Python class called `Animal` with a method `speak()`. Then, create child classes like `Dog`, `Cat`, and `Bird`, each with their own `speak()` method. Demonstrate polymorphism by calling the `speak()` method
on objects of different subclasses.

class Animal:
    def __init__(self):


    def speak(self):
        print("Parent class")




class Dog(Animal):
    def speak(self):
        print("Child Class Dog")


class Bird(Animal):
    def speak(self):
        print("Child Class Bird")
```

[ ]:
```
67.Discuss the use of abstract methods and classes in achieving polymorphism in Python. Provide an example
using the `abc` module.

The main goal of the abstract base class is to provide a standardized way to test whether an object adheres to a given specification. It can also prevent any attempt to instantiate a subclass that doesn't override a particular method in the superclass. And finally, using an abstract class, a class can derive identity from another class without any object inheritance.
```

```
Python has a module called abc (abstract base class) that offers the necessary␣
 ↪tools for crafting an abstract base class. First and foremost, you should␣
 ↪understand the ABCMeta metaclass provided by the abstract base class. The␣
 ↪rule is every abstract class must use ABCMeta metaclass.

ABCMeta metaclass provides a method called register method that can be invoked␣
 ↪by its instance. By using this register method, any abstract base class can␣
 ↪become an ancestor of any arbitrary concrete class. Let's understand this␣
 ↪process by considering an example of an abstract base class that registers␣
 ↪itself as an ancestor of dict.
```

[ ]:
```
68.Create a Python class hierarchy for a vehicle system (e.g., car, bicycle,␣
 ↪boat) and implement

class Vehicle:
    def __init__(self):


    def start(self):
        print("Parent class")




class Car(Vehicle):
    def start(self):
        print("Child Class Car")


class Bicycle(Vehicle):
    def start(self):
        print("Child Class Bicycle")


 class Boat(Vehicle):
    def start(self):
        print("Child Class Boat")
```

[ ]:
```
69.Explain the significance of the `isinstance()` and `issubclass()` functions␣
 ↪in Python polymorphism.

1.issubclass() in Python
```

```
Python issubclass() is built-in function used to check if a class is a subclass␣
 ↪of another class or not. This function returns True if the given class is␣
 ↪the subclass of given class else it returns False.

2.isinstance(object, classinfo) asks whether an object is an instance of a␣
 ↪class (or a tuple of classes).
```

[ ]:
```
70.What is the role of the `@abstractmethod` decorator in achieving␣
 ↪polymorphism in Python? Provide an
example.

Abstract methods are methods that are declared in a base class but not defined.␣
 ↪They act as placeholders that must be overridden by subclasses that inherit␣
 ↪from the base class. Abstract methods enforce a consistent interface for␣
 ↪subclasses and ensure that they provide their own implementation of the␣
 ↪method. To create an abstract method in Python, you need to import the␣
 ↪abstract base class (ABC) module and use the @abstractmethod decorator.


from abc import ABC, abstractmethod



class Animal(ABC):

  @abstractmethod

  def speak(self):

    pass



class Dog(Animal):

  def speak(self):

    print("Woof")



class Cat(Animal):

  def speak(self):

    print("Meow")
```

[ ]:
```
71.Create a Python class called `Shape` with a polymorphic method `area()` that␣
 ↪calculates the area of different shapes (e.g., circle, rectangle, triangle).

# Import the math module to access mathematical functions like pi
import math

# Define a base class called Shape to represent a generic shape with methods␣
 ↪for calculating area and perimeter
class Shape:
    # Placeholder method for calculating area (to be implemented in derived␣
 ↪classes)
    def area(self):
        pass


#Define a derived class called Circle, which inherits from the Shape class
class Circle(Shape):
    # Initialize the Circle object with a given radius
    def __init__(self, radius):
        self.radius = radius

    # Calculate and return the area of the circle using the formula:   * r^2
    def area(self):
        return math.pi * self.radius**2



# Define a derived class called Rectangle, which inherits from the Shape class
class Rectangle(Shape):
    # Initialize the Rectangle object with given length and width
    def __init__(self, length, width):
        self.length = length
        self.width = width

    # Calculate and return the area of the rectangle using the formula: length␣
 ↪* width
    def area(self):
        return self.length * self.width


# Define a derived class called Triangle, which inherits from the Shape class
class Triangle(Shape):
    # Initialize the Triangle object with a base, height, and three side lengths
    def __init__(self, base, height, side1, side2, side3):
        self.base = base
        self.height = height
        self.side1 = side1
```

```
        self.side2 = side2
        self.side3 = side3

    # Calculate and return the area of the triangle using the formula: 0.5 *␣
 ↪base * height
    def area(self):
        return 0.5 * self.base * self.height
```

[ ]:
```
72.Discuss the benefits of polymorphism in terms of code reusability and␣
 ↪flexibility in Python programs.

A. Code Reusability

By enabling programmers to reuse the same code for various data types,␣
 ↪polymorphism encourages code reuse. As a result, less code needs to be␣
 ↪written and code maintenance is made simpler over time.

B. Flexibility

Code can be more flexible thanks to polymorphism, which enables objects and␣
 ↪functions to take on different forms. Because of this, it is simpler to␣
 ↪write code that works with various data types, enabling developers to␣
 ↪produce more adaptable and reusable software.
```

[ ]:
```
73.Explain the use of the `super()` function in Python polymorphism. How does␣
 ↪it help call methods of parent
classes?

1.
The super() function is used to give access to methods and properties of a␣
 ↪parent or sibling class.

The super() function returns an object that represents the parent class.

2.

class Emp():
    def __init__(self, id, name, Add):
        self.id = id
        self.name = name
        self.Add = Add

# Class freelancer inherits EMP
class Freelance(Emp):
    def __init__(self, id, name, Add, Emails):
        super().__init__(id, name, Add)
```

```
        self.Emails = Emails

Emp_1 = Freelance(103, "Suraj kr gupta", "Noida" , "KKK@gmails")
print('The ID is:', Emp_1.id)
print('The Name is:', Emp_1.name)
print('The Address is:', Emp_1.Add)
print('The Emails is:', Emp_1.Emails)
```

[ ]:
```
74.Create a Python class hierarchy for a banking system with various account␣
 ↪types (e.g., savings, checking,



class Account:

    def withdraw(self):
        pass



class Savings(Account):

    def __init__(self):



    def withdraw(self):
       print("Savings Account")



class Checkings(Account):

    def __init__(self):



    def withdraw(self):
       print("Checkings Account")



class CreditCard(Account):

    def __init__(self):
```

```
    def withdraw(self):
        print("Credit Card")
```

[ ]:
```
75.Describe the concept of operator overloading in Python and how it relates to␣
 ↪polymorphism. Provide
examples using operators like `+` and `*`.

Python operators work for predefined data types like int, str, list, etc, but␣
 ↪we can change the way an operator works depending on the types of operands␣
 ↪that we use. We may use any inbuilt or user-defined operand. This is the␣
 ↪feature of operator overloading in Python that allows the same built-in␣
 ↪operator to behave differently according to the context of the␣
 ↪implementation of a problem.

Operator overloading in Python provides the ability to override the␣
 ↪functionality of a built-in operator in user-defined classes.

For example, the "*" operator can be overloaded not only as a multiplier for␣
 ↪numbers but also as a repetition operator for lists or strings.

Operator overloading is also known as Operator Ad-hoc Polymorphism.

Now suppose we want to add two complex numbers. We know that complex numbers␣
 ↪are added by adding the real parts and imaginary parts separately which␣
 ↪results in a new complex number. So we can simply use the "+" operator with␣
 ↪two objects of ComplexNumber:
```

[ ]:
```
76.What is dynamic polymorphism, and how is it achieved in Python?

Dynamic polymorphism in Java refers to the process when a call to an overridden␣
 ↪process is resolved at the run time. The reference variable of a superclass␣
 ↪calls the overridden method. As the name dynamic connotes, dynamic␣
 ↪polymorphism happens among different classes as opposed to static␣
 ↪polymorphism. Dynamic polymorphism facilitates the overriding of methods in␣
 ↪Java which is core for run-time polymorphism.

class Bike{

 void run(){System.out.println("running");}

}

class Splendor extends Bike{
```

```
  void run(){System.out.println("walking safely with 30km");}

    public static void main(String args[]){

    Bike b = new Splendor();//upcasting

    b.run();

  }

}
```

[ ]:
```
77.Create a Python class hierarchy for employees in a company (e.g., manager,␣
 ↪developer, designer) and

class Employee:

    def calculate_salary(self):
        pass



class Manager(Employee):

    def __init__(self):



    def calculate_salary(self):
       print("Manager")



class Developer(Employee):

    def __init__(self):



    def calculate_salary(self):
       print("Developer")



class Designer(Employee):

    def __init__(self):
```

```
    def calculate_salary(self):
        print("Designer")
```

[ ]:
```
78.Discuss the concept of function pointers and how they can be used to achieve␣
↪polymorphism in Python.
```

[ ]:
```
79.Explain the role of interfaces and abstract classes in polymorphism, drawing␣
↪comparisons between them.

A blueprint for other classes might be thought of as an abstract class. You may␣
↪use it to define a collection of methods that are required for all␣
↪subclasses derived from the abstract class. An abstract class is one that␣
↪includes one or more abstract methods. A method that has a declaration but␣
↪no implementation is said to be abstract. We use an abstract class for␣
↪creating huge functional units. An abstract class is used to offer a␣
↪standard interface for various implementations of a component.

The interface in object-oriented languages like Python is a set of method␣
↪signatures that the implementing class is expected to provide. Writing␣
↪ordered code and achieving abstraction are both possible through interface␣
↪implementation.
```

[ ]:
```
80.Create a Python class for a zoo simulation, demonstrating polymorphism with␣
↪different animal types (e.g.,
class Animal:

    def eating(self):
        pass

     def sleeping(self):
        pass

     def making_soundsy(self):
        pass



class Mammals(Animal):

    def __init__(self):



    def eating(self):
```

```
        print("Developer")

    def sleeping(self):
       print("Developer")

    def making_soundsy(self):
        print("Developer")



class Birds(Animal):

    def __init__(self):



    def eating(self):
         print("Birds")

    def sleeping(self):
       print("Birds")

    def making_soundsy(self):
        print("Birds")



class Reptiles(Animal):

    def __init__(self):



    def eating(self):
         print("Reptiles")

    def sleeping(self):
       print("Reptiles")

    def making_soundsy(self):
        print("Reptiles")
```

[ ]:
```
81.What is abstraction in Python, and how does it relate to object-oriented␣
 ↪programming?
Abstraction in python is defined as a process of handling complexity by hiding␣
 ↪unnecessary information from the user. This is one of the core concepts of␣
 ↪object-oriented programming (OOP) languages. That enables the user to␣
 ↪implement even more complex logic on top of the provided abstraction without␣
 ↪understanding or even thinking about all the hidden background/back-end␣
 ↪complexity.

That's a very generic core topic not only limited to object-oriented␣
 ↪programming. You can observe it everywhere in the real world or in our␣
 ↪surroundings.
```

[ ]:
```
82.Describe the benefits of abstraction in terms of code organization and␣
 ↪complexity reduction.
Abstractions help hide concrete code which can change with abstractions such as␣
 ↪interfaces and abstract classes which are less likely to change.

Using abstraction limits dependencies in code, reduced dependency means␣
 ↪reducing the effect of change in your code.

Two of the key long term goals for developers when writing code

Increase the readability of the code
Minimise dependencies in code, reducing the effects of change
The two goals will make maintaining the code easier

Abstractions make it easier to understand code because it concentrates on core␣
 ↪features/actions and not on the small details.

The code is easy to understand because if you have captured the right␣
 ↪abstractions is obvious what the code is doing.

Abstractions help create loosly coupled code.  The reason is the abstraction␣
 ↪acts like a type and encapsulates the potential changes behind the abstract␣
 ↪class/interface.
```

[ ]:
```
83.Create a Python class called `Shape` with an abstract method␣
 ↪`calculate_area()`. Then, create child classes (e.g., `Circle`, `Rectangle`)␣
 ↪that implement the `calculate_area()` method. Provide an example of
using these classes.

# Import the math module to access mathematical functions like pi
import math
```

```python
# Define a base class called Shape to represent a generic shape with methods␣
 ↪for calculating area and perimeter
class Shape:
    # Placeholder method for calculating area (to be implemented in derived␣
 ↪classes)
    def calculate_area(self):
        pass


#Define a derived class called Circle, which inherits from the Shape class
class Circle(Shape):
    # Initialize the Circle object with a given radius
    def __init__(self, radius):
        self.radius = radius

    # Calculate and return the area of the circle using the formula:   * r^2
    def calculate_area(self):
        return math.pi * self.radius**2



# Define a derived class called Rectangle, which inherits from the Shape class
class Rectangle(Shape):
    # Initialize the Rectangle object with given length and width
    def __init__(self, length, width):
        self.length = length
        self.width = width

    # Calculate and return the area of the rectangle using the formula: length␣
 ↪* width
    def calculate_area(self):
        return self.length * self.width


# Define a derived class called Triangle, which inherits from the Shape class
class Triangle(Shape):
    # Initialize the Triangle object with a base, height, and three side lengths
    def __init__(self, base, height, side1, side2, side3):
        self.base = base
        self.height = height
        self.side1 = side1
        self.side2 = side2
        self.side3 = side3

    # Calculate and return the area of the triangle using the formula: 0.5 *␣
 ↪base * height
```

```
    def calculate_area(self):
        return 0.5 * self.base * self.height
```

[ ]:
```
84.Explain the concept of abstract classes in Python and how they are defined␣
 ↪using the `abc` module. Provide
an example.

In Python, you can create an abstract class using the abc module. This module␣
 ↪provides the infrastructure for defining abstract base classes (ABCs).

Here's a simple example:

from abc import ABC, abstractmethod

class AbstractClassExample(ABC):
    @abstractmethod
    def do_something(self):
        pass
Python
In this example, we import the ABC and abstractmethod from the abc module. We␣
 ↪then define an abstract class AbstractClassExample using the ABC as the base␣
 ↪class. The abstractmethod decorator is used to declare the method␣
 ↪do_something as an abstract method. This means that any class that inherits␣
 ↪from AbstractClassExample must provide an implementation of the do_something␣
 ↪method.

This is a basic way to create an abstract class in Python, but there's much␣
 ↪more to learn about using and understanding abstract classes. Continue␣
 ↪reading for a more detailed understanding and advanced usage scenarios.
```

[ ]:
```
85.How do abstract classes differ from regular classes in Python? Discuss their␣
 ↪use cases.

A class is a construct that enables you to create your own custom types by␣
 ↪grouping together variables of other types, methods and events. A class is␣
 ↪like a blueprint. It defines the data and behavior of a type. If the class␣
 ↪is not declared as static, client code can use it by creating objects or␣
 ↪instances which are assigned to a variable. The variable remains in memory␣
 ↪until all references to it go out of scope. At that time, the CLR marks it␣
 ↪as eligible for garbage collection. If the class is declared as static, then␣
 ↪only one copy exists in memory and client code can only access it through␣
 ↪the class itself, not an instance variable.
```

```
Abstract classes, marked by the keyword abstract in the class definition, are␣
 ↪typically used to define a base class in the hierarchy. What's special about␣
 ↪them, is that you can't create an instance of them - if you try, you will␣
 ↪get a compile error. Instead, you have to subclass them, and create an␣
 ↪instance of your subclass. So when do you need an abstract class? It really␣
 ↪depends on what you do.
```

[ ]:
```
86.Create a Python class for a bank account and demonstrate abstraction by␣
 ↪hiding the account balance and
providing methods to deposit and withdraw funds.

class BankAccount:
def __init__(self, _account_number, _balance):
self._account_number = account_number
self._balance = balance

def deposit(self, amount):
self._balance += amount

def withdraw(self, amount):
if amount <= self._balance:
self._balance -= amount
else:
print("Insufficient funds.")
```

[ ]:
```
87.Discuss the concept of interface classes in Python and their role in␣
 ↪achieving abstraction.
```

[ ]:
```
88.Create a Python class hierarchy for animals and implement abstraction by␣
 ↪defining common methods (e.g., `eat()`, `sleep()`) in an abstract base class.

from abc import ABC, abstractmethod

class Animal(ABC):
    def __init__(self, name):
        self.name = name

    @abstractmethod
    def eat(self):
        pass

    @abstractmethod
    def sleep(self):
        pass
```

```
class Dog(Animal):
    def __init__(self):

    def eat(self):
        print("Dog Eating")

     def sleep(self):
        print("Dog Sleeping")


  class Cat(Animal):
    def __init__(self):

    def eat(self):
        print("Cat Eating")

     def sleep(self):
        print("Cat Sleeping")
```

[ ]:
```
89.Explain the significance of encapsulation in achieving abstraction. Provide␣
 ↪examples.

Abstraction and encapsulation are two crucial concepts of Object Oriented␣
 ↪Programming(OOP) in the field of computer science that are used to make code␣
 ↪more manageable, secure, and efficient.

What is Abstraction in Python?

Abstraction is a technique that involves hiding the implementation details of a␣
 ↪class and only exposing the essential features of the class to the user.␣
 ↪This allows the user to work with objects of the class without having to␣
 ↪worry about the details of how the class works.

For example, let's say we have a class called Car that has methods such as␣
 ↪start_engine(), accelerate(), and brake(). The user only needs to know that␣
 ↪they can start the engine, accelerate, and brake the car, but they don't␣
 ↪need to know how these methods are implemented. Here's what the code would␣
 ↪look like:

class Car:
    def start_engine(self):
        # Implementation details hidden
        print("Engine started")

    def accelerate(self):
        # Implementation details hidden
        print("Accelerating")
```

```
    def brake(self):
        # Implementation details hidden
        print("Braking")
In this example, the user can use the Car class in a straightforward way,␣
 ↪without having to worry about the implementation details.

What is Encapsulation in Python?

Encapsulation is a technique that involves wrapping data and functions into a␣
 ↪single unit or object. This allows you to control the way the data and␣
 ↪functions are accessed and helps to prevent accidental modification of the␣
 ↪data.

For example, let's say we have a class called BankAccount that has attributes␣
 ↪such as balance and account_number and methods such as deposit() and␣
 ↪withdraw(). We want to ensure that the balance can only be modified through␣
 ↪the deposit() and withdraw() methods and not directly. Here's what the code␣
 ↪would look like:

class BankAccount:
    def __init__(self, balance, account_number):
        self.__balance = balance
        self.__account_number = account_number

    def deposit(self, amount):
        self.__balance += amount

    def withdraw(self, amount):
        if self.__balance >= amount:
            self.__balance -= amount
        else:
            print("Insufficient funds")

    def get_balance(self):
        return self.__balance
In this example, the balance and account_number attributes are prefixed with␣
 ↪two underscores, which makes them private. This means that they can only be␣
 ↪accessed within the BankAccount class and not outside of it. The user can␣
 ↪only modify the balance through the deposit() and withdraw() methods,␣
 ↪ensuring that the data is secure.
```

[ ]:
```
90.What is the purpose of abstract methods, and how do they enforce abstraction␣
 ↪in Python classes?
```

```
An abstract method is a method that has a declaration but does not have an␣
↪implementation. While we are designing large functional units we use an␣
↪abstract class. When we want to provide a common interface for different␣
↪implementations of a component, we use an abstract class.


The abstract Method is used for creating blueprints for classes or interfaces.␣
↪Here methods are defined but these methods don't provide the implementation.␣
↪Abstract Methods can only be implemented using subclasses or classes that␣
↪implement the interfaces.
```

[ ]:
```
91.Create a Python class for a vehicle system and demonstrate abstraction by␣
↪defining common methods


class Vehicle(ABC):
    def __init__(self, type):
        self.type = type

    @abstractmethod
    def drive(self):
        pass

    @abstractmethod
    def stop(self):
        pass



class Car(Vehicle):
    def __init__(self):

    def drive(self):
        print("Car")

     def stop(self):
        print("Car")


  class Bus(Animal):
    def __init__(self):

    def drive(self):
        print("Bus")

     def eat(self):
        print("Bus")
```

[ ]:
```
92.Describe the use of abstract properties in Python and how they can be␣
 ↪employed in abstract classes.

Abstract class in Python also provides the functionality of abstract properties␣
 ↪in addition to abstract methods. Properties are Pythonic ways of using␣
 ↪getters and setters. The abc module has a @abstractproperty decorator to use␣
 ↪abstract properties. Just like abstract methods, we need to define abstract␣
 ↪properties in implementation classes. Otherwise, Python will raise the error.

As we have been told, properties are used in Python for getters and setters.␣
 ↪Abstract property is provided by the abc module to force the child class to␣
 ↪provide getters and setters for a variable in Python.

Let's define an abstract property in our Shape class:

from abc import ABC, abstractmethod, abstractproperty

class Shape(ABC):
    def __init__(self, shape_name):
        self.shape_name = shape_name

    @abstractproperty
    def name(self):
        pass

    @abstractmethod
    def draw(self):
        pass
```

[ ]:
```
93.Create a Python class hierarchy for employees in a company (e.g., manager,␣
 ↪developer, designer) and

class Employee(ABC):
    def __init__(self, type):
        self.type = type

    @abstractmethod
    def work(self):
        pass

    @abstractmethod
    def check(self):
        pass



class Manager(Employee):
```

```
    def __init__(self):

    def work(self):
        print("Manager")

     def check(self):
        print("Manager")


  class Developer(Employee):
    def __init__(self):

    def drive(self):
        print("Developer")

     def eat(self):
        print("Developer")
```

[ ]:
```
94.Discuss the differences between abstract classes and concrete classes in␣
 ↪Python, including their
instantiation.

(a)
Abstraction is the concept in object-oriented programming that is used to hide␣
 ↪the internal functionality of the classes from the users. Abstraction is␣
 ↪implemented using the abstract classes. An abstract class in Python is␣
 ↪typically created to declare a set of methods that must be created in any␣
 ↪child class built on top of this abstract class. Similarly, an abstract␣
 ↪method is one that doesn't have any implementation.


(b)

Abstract classes usually have partial or no implementation. On the other hand,␣
 ↪concrete classes always have full implementation of its behavior.Unlike␣
 ↪concrete classes, abstract classes cannot be instantiated. Therefore␣
 ↪abstract classes have to be extended in order to make them useful. Abstract␣
 ↪classes may contain abstract methods, but concrete classes can't. When an␣
 ↪abstract class is extended, all methods (both abstract and concrete) are␣
 ↪inherited. The inherited class can implement
any or all the methods. If all the abstract methods are not implemented,then␣
 ↪that class also becomes an abstract class.
```

[ ]:
```
95.Explain the concept of abstract data types (ADTs) and their role in␣
 ↪achieving abstraction in Python.
```

```
Abstract Data type (ADT) is a type (or class) for objects whose behavior is␣
 ↪defined by a set of values and a set of operations. The definition of ADT␣
 ↪only mentions what operations are to be performed but not how these␣
 ↪operations will be implemented. It does not specify how data will be␣
 ↪organized in memory and what algorithms will be used for implementing the␣
 ↪operations. It is called "abstract" because it gives an␣
 ↪implementation-independent view.

The process of providing only the essentials and hiding the details is known as␣
 ↪abstraction.
```

[ ]:
```
96.Create a Python class for a computer system, demonstrating abstraction by␣
 ↪defining common methods

class ComputerSystem(ABC):
    def __init__(self, type):
        self.type = type

    @abstractmethod
    def power_on(self):
        pass

    @abstractmethod
    def shutdown(self):
        pass
```

[ ]:
```
97.Discuss the benefits of using abstraction in large-scale software␣
 ↪development projects.
What is Abstraction in Python?

Abstraction in python is defined as a process of handling complexity by hiding␣
 ↪unnecessary information from the user. This is one of the core concepts of␣
 ↪object-oriented programming (OOP) languages. That enables the user to␣
 ↪implement even more complex logic on top of the provided abstraction without␣
 ↪understanding or even thinking about all the hidden background/back-end␣
 ↪complexity.

That's a very generic core topic not only limited to object-oriented␣
 ↪programming. You can observe it everywhere in the real world or in our␣
 ↪surroundings.
```

[ ]:
```
98.Explain how abstraction enhances code reusability and modularity in Python␣
 ↪programs.

Benefits of Data Abstraction:
```

```
Modularity: Data abstraction promotes modularity by encapsulating data and␣
 ↪behavior within a class. This makes it easier to organize code and␣
 ↪collaborate with others, as classes provide clear boundaries and␣
 ↪well-defined interfaces.
Reusability: Once you've defined an abstract data type, you can reuse it␣
 ↪throughout your program or in other projects. This reusability reduces code␣
 ↪duplication and saves development time.
Maintenance: By encapsulating implementation details, data abstraction allows␣
 ↪you to change the internal workings of a class without affecting the code␣
 ↪that uses it. This makes maintenance and updates less error-prone.
Complexity Management: Data abstraction helps manage the complexity of large␣
 ↪software projects. It enables developers to focus on high-level design and␣
 ↪problem-solving while abstracting away low-level details.
```

[ ]:
```
99.Create a Python class for a library system, implementing abstraction by␣
 ↪defining common methods (e.g.,

class Library(ABC):
    def __init__(self, type):
        self.type = type

    @abstractmethod
    def add_book(self):
        pass

    @abstractmethod
    def borrow_book(self):
        pass
```

[ ]:
```
100.Describe the concept of method abstraction in Python and how it relates to␣
 ↪polymorphism.

(a)
What is Polymorphism?
The literal meaning of polymorphism is the condition of occurrence in different␣
 ↪forms.

Polymorphism is a very important concept in programming. It refers to the use␣
 ↪of a single type entity (method, operator or object) to represent different␣
 ↪types in different scenarios.

(b)Polymorphism is the ability to have objects of different types understanding␣
 ↪the same message, so you can use those objects the same way without worrying␣
 ↪about their concrete type.
```

```
(c)Another way to use polymorphism in Python is by combining abstract methods␣
 ↪and inheritance. This means that you can define an abstract method in a base␣
 ↪class and then override it with different implementations in subclasses.␣
 ↪This way, you can create a common interface for subclasses that share some␣
 ↪functionality but differ in some details. For example, if you have a base␣
 ↪class called DataProcessor that has an abstract method called process, you␣
 ↪can override this method in subclasses like CSVProcessor, JSONProcessor, and␣
 ↪XMLProcessor and provide different ways of processing data files. Here is an␣
 ↪example of how to do this:
from abc import ABC, abstractmethod



class DataProcessor(ABC):

  @abstractmethod

  def process(self, file):

    pass



class CSVProcessor(DataProcessor):

  def process(self, file):

    print(f"Reading and parsing {file} as CSV")



class JSONProcessor(DataProcessor):

  def process(self, file):

    print(f"Reading and parsing {file} as JSON")



class XMLProcessor(DataProcessor):

  def process(self, file):

    print(f"Reading and parsing {file} as XML")
```

[ ]:
```
101.Explain the concept of composition in Python and how it is used to build␣
 ↪complex objects from simpler ones.

Compositions-It is a programming technique used when establishing relationships␣
 ↪between classes and objects. Understanding compositions is important in␣
 ↪object-oriented programming. Compositions can be a good idea in Python and␣
 ↪software language to make a particular structure dynamic. Because in this␣
 ↪way, we fill the place of complex code blocks with more compact material.

Why Composition?
Through the compositions, we actually state the relationship between the two␣
 ↪classes. And in this way, the code can be reused. This has a common␣
 ↪similarity with the concept of inheritance. As a result, we obtain more␣
 ↪complex structures by combining objects between different types.

What we need to understand with quality; is our ability to reach a clear␣
 ↪decision thanks to the conformity with the definitions made about a class␣
 ↪member. Values and methods are created about the breed, such as an apple is␣
 ↪a fruit and a fawn being an animal. And the attributes and methods created␣
 ↪for these main classes are exactly applied to the lower classes. After the␣
 ↪application, the results are expected not to conflict with the upper class.␣
 ↪Of course, this will save you from code repetitions.
```

[ ]:
```
102.Describe the difference between composition and inheritance in␣
 ↪object-oriented programming.

1.Inheritance :

 Inheritance represents the IS-A relationship, which is also called as␣
 ↪parent-child relationship.

 In inheritance, the child class is dependent upon parent class.

 In inheritance, we can extend only one class so that we can reuse the code of␣
 ↪only one class.

2.Composition:

   The Composition is a way to design or implement the "has-a" relationship.

   Whereas in composition, both child class and parent class are independent.

   In Composition, we can use the functionalities from multiple classes.

   Whereas, using composition, we can easily achieve multiple inheritance
```

[ ]:
```
103.Create a Python class called `Author` with attributes for name and␣
 ↪birthdate. Then, create a `Book` class
that contains an instance of `Author` as a composition. Provide an example of␣
 ↪creating a `Book` object.


class Author:
    def __init__(self, name,birthdate):
        self.name = name
        self.birthdate = birthdate


 class Book:
    def __init__(self, address, salary, auth_obj):
        self.address = address
        self.salary = salary

       # creating object of Author class
       auth_obj= Author()
```

[ ]:
```
104.Discuss the benefits of using composition over inheritance in Python,␣
 ↪especially in terms of code flexibility
and reusability.

1.It is one of the fundamental concepts of Object-Oriented Programming. In this␣
 ↪concept, we will describe a class that references to one or more objects of␣
 ↪other classes as an Instance variable. Here, by using the class name or by␣
 ↪creating the object we can access the members of one class inside another␣
 ↪class. It enables creating complex types by combining objects of different␣
 ↪classes. It means that a class Composite can contain an object of another␣
 ↪class Component. This type of relationship is known as Has-A Relation.

2.Why Composition?
Through the compositions, we actually state the relationship between the two␣
 ↪classes. And in this way, the code can be reused. This has a common␣
 ↪similarity with the concept of inheritance. As a result, we obtain more␣
 ↪complex structures by combining objects between different types.
```

```
What we need to understand with quality; is our ability to reach a clear␣
 ↪decision thanks to the conformity with the definitions made about a class␣
 ↪member. Values and methods are created about the breed, such as an apple is␣
 ↪a fruit and a fawn being an animal. And the attributes and methods created␣
 ↪for these main classes are exactly applied to the lower classes. After the␣
 ↪application, the results are expected not to conflict with the upper class.␣
 ↪Of course, this will save you from code repetitions.

In the composition, the code is reused by adding other objects to objects␣
 ↪instead of inheriting properties, created methods between classes. And␣
 ↪indeed, classes created by composition are more flexible than the structure␣
 ↪created by heritage. This is because adding new features without changing␣
 ↪the existing code structure is more effective.
```

[ ]:
```
105.How can you implement composition in Python classes? Provide examples of␣
 ↪using composition to create
complex objects.

We can access the member of one class inside a class using these 2 concepts:

By Composition(Has-A Relation)
By Inheritance(Is-A Relation)

By using the class names or by creating an object we can access the member of␣
 ↪one class inside another class.

Example:

class Engine:
    # engine specific functionality
    '''
    '''
    '''


class Car:
    e = Engine()
    e.method1()
    e.method2()
    '''
    '''
In these above example class Car has-A Engine class reference. Here inside␣
 ↪class Car also we can create different variables and methods. Using object␣
 ↪reference of Engine class inside Car class we can easily access each and␣
 ↪every member of Engine class inside Car class.
```

```
Code-

class Employee:

    # constructor for initialization
    def __init__(self, name, age):
        self.name = name
        self.age = age

    # instance method
    def emp_data(self):
        print('Name of Employee : ', self.name)
        print('Age of Employee : ', self.age)


class Data:
    def __init__(self, address, salary, emp_obj):
        self.address = address
        self.salary = salary

        # creating object of Employee class
        self.emp_obj = emp_obj

    # instance method
    def display(self):

        # calling Employee class emp_data()
        # method
        self.emp_obj.emp_data()
        print('Address of Employee : ', self.address)
        print('Salary of Employee : ', self.salary)

# creating Employee class object
emp = Employee('Ronil', 20)

# passing obj. of Emp. class during creation
# of Data class object
data = Data('Indore', 25000, emp)

# call Data class instance method
data.display()
```

[ ]:
```
106.Create a Python class hierarchy for a music player system, using␣
 ↪composition to represent playlists and
songs.
```

```
class Song:

    # constructor for initialization
    def __init__(self,song_language):

      # instance method
    def songLanguage(self):
        print('Language of song : ', self.song_language)


     class PlayList:

    # constructor for initialization
    def __init__(self, NumberOfSongs,obj_songs):
        self.NumberOfSongs = NumberOfSongs
        self.obj_songs = Song



    # instance method
    def numberOfSongs(self):
        print('Name of Employee : ', self.NumberOfSongs)


     # instance method
    def display(self):
      self.obj_song.songLanguage()
```

[ ]:
```
107.Explain the concept of "has-a" relationships in composition and how it␣
 ↪helps design software systems.

1.Composite pattern is a partitioning design pattern and describes a group of␣
 ↪objects that is treated the same way as a single instance of the same type␣
 ↪of object. The intent of a composite is to "compose" objects into tree␣
 ↪structures to represent part-whole hierarchies. It allows you to have a tree␣
 ↪structure and ask each node in the tree structure to perform a task.

As described by Gof, "Compose objects into tree structure to represent␣
 ↪part-whole hierarchies. Composite lets client treat individual objects and␣
 ↪compositions of objects uniformly".
```

```
When dealing with Tree-structured data, programmers often have to discriminate␣
 ↪between a leaf-node and a branch. This makes code more complex, and␣
 ↪therefore, error prone. The solution is an interface that allows treating␣
 ↪complex and primitive objects uniformly.

In object-oriented programming, a composite is an object designed as a␣
 ↪composition of one-or-more similar objects, all exhibiting similar␣
 ↪functionality. This is known as a "has-a" relationship between objects.

2.
The key concept is that you can manipulate a single instance of the object just␣
 ↪as you would manipulate a group of them. The operations you can perform on␣
 ↪all the composite objects often have a least common denominator relationship.
```

[ ]:
```
108.Create a Python class for a computer system, using composition to represent␣
 ↪components like CPU, RAM,
and storage devices.

class ComputerSystem:

   # composite class constructor
    def __init__(self):
        print('ComputerSystem class object created...')

    # composite class instance method
    def m1(self):
        print('ComputerSystem class m1() method executed...')


class CPU:

    # composite class constructor
    def __init__(self):

        # creating object of component class
        self.obj1 = ComputerSystem()

        print('CPU class object also created...')

     # composite class instance method
    def m2(self):

        print('CPU class m2() method executed...')

        # calling m1() method of component class
        self.obj1.m1()
```

```
class RAM:

    # composite class constructor
    def __init__(self):

        # creating object of component class
        self.obj2 = ComputerSystem()

        print('RAM class object also created...')

     # composite class instance method
    def m2(self):

        print('RAM class m2() method executed...')

        # calling m1() method of component class
        self.obj2.m1()
```

[ ]:
```
109.Describe the concept of "delegation" in composition and how it simplifies␣
 ↪the design of complex systems.

Delegation is a design pattern in which an object, called the delegate, is␣
 ↪responsible for performing certain tasks on behalf of another object, called␣
 ↪the delegator. This can be done by the delegator forwarding method calls and␣
 ↪attribute access to the delegate. In its most basic form, delegation can be␣
 ↪implemented using the following approach: the delegator passes requests for␣
 ↪certain actions to the delegate, which then performs the actions on behalf␣
 ↪of the delegator.
```

[ ]:
```
110.Create a Python class for a car, using composition to represent components␣
 ↪like the engine, wheels, and
transmission.

class Car:

    # composite class constructor
    def __init__(self):
        print('Car class object created...')

    # composite class instance method
    def m1(self):
        print('Car class m1() method executed...')
```

```
class Engine:

    # composite class constructor
    def __init__(self):

        # creating object of component class
        self.obj1 = Car()

        print('Engine class object also created...')

     # composite class instance method
    def m2(self):

        print('Engine class m2() method executed...')

        # calling m1() method of component class
        self.obj1.m1()


   class Wheels:

    # composite class constructor
    def __init__(self):

        # creating object of component class
        self.obj2 = Car()

        print('Wheels class object also created...')

     # composite class instance method
    def m2(self):

        print('Wheels class m2() method executed...')

        # calling m1() method of component class
        self.obj2.m1()
```

[ ]:
```
111.How can you encapsulate and hide the details of composed objects in Python␣
 ↪classes to maintain
abstraction?
```

[ ]:
```
112.Create a Python class for a university course, using composition to␣
 ↪represent students, instructors, and
course materials.
```

```
class UniversityCourse:

   # composite class constructor
    def __init__(self):
        print('UniversityCourse class object created...')

    # composite class instance method
    def m1(self):
        print('UniversityCourse class m1() method executed...')


class Student:

    # composite class constructor
    def __init__(self):

        # creating object of component class
        self.obj1 = UniversityCourse()

        print('Student class object also created...')

     # composite class instance method
    def m2(self):

        print('Student class m2() method executed...')

        # calling m1() method of component class
        self.obj1.m1()


   class Instructors:

    # composite class constructor
    def __init__(self):

        # creating object of component class
        self.obj2 = UniversityCourse()

        print('Instructors class object also created...')

     # composite class instance method
    def m2(self):

        print('Instructors class m2() method executed...')

        # calling m1() method of component class
        self.obj2.m1()
```

```
  class CourseMaterial:

    # composite class constructor
    def __init__(self):

        # creating object of component class
        self.obj3 = UniversityCourse()

        print('CourseMaterial class object also created...')

     # composite class instance method
    def m2(self):

        print('CourseMaterial class m2() method executed...')

        # calling m1() method of component class
        self.obj3.m1()
```

[ ]:
```
113.Discuss the challenges and drawbacks of composition, such as increased␣
 ↪complexity and potential for
tight coupling between objects.

# Code to demonstrate Aggregation

# Salary class with the public method
# annual_salary()
class Salary:
    def __init__(self, pay, bonus):
        self.pay = pay
        self.bonus = bonus

    def annual_salary(self):
        return (self.pay*12)+self.bonus


# EmployeeOne class with public method
# total_sal()
class EmployeeOne:

    # Here the salary parameter reflects
    # upon the object of Salary class we
    # will pass as parameter later
```

```
    def __init__(self, name, age, sal):
        self.name = name
        self.age = age

        # initializing the sal parameter
        self.agg_salary = sal   # Aggregation

    def total_sal(self):
        return self.agg_salary.annual_salary()

# Here we are creating an object
# of the Salary class
# in which we are passing the
# required parameters
salary = Salary(10000, 1500)

# Now we are passing the same
# salary object we created
# earlier as a parameter to
# EmployeeOne class
emp = EmployeeOne('Geek', 25, salary)

print(emp.total_sal())

From the above code,  we will get the same output as we got before using the␣
 ↪Composition concept. But the difference is that here we are not creating an␣
 ↪object of the Salary class inside the EmployeeOne class, rather than that we␣
 ↪are creating an object of the Salary class outside and passing it as a␣
 ↪parameter of EmployeeOne class which yields the same result.
```

[ ]:
```
114.Create a Python class hierarchy for a restaurant system, using composition␣
 ↪to represent menus, dishes,
and ingredients.


class Restaurant:

   # composite class constructor
    def __init__(self):
        print('Restaurant class object created...')

    # composite class instance method
    def m1(self):
        print('Restaurant class m1() method executed...')


class Menu:
```

```
 # composite class constructor
 def __init__(self):

     # creating object of component class
     self.obj1 = Restaurant()

     print('Menu class object also created...')

  # composite class instance method
 def m2(self):

     print('Menu class m2() method executed...')

     # calling m1() method of component class
     self.obj1.m1()


class Dishes:

 # composite class constructor
 def __init__(self):

     # creating object of component class
     self.obj2 = Restaurant()

     print('Dishes class object also created...')

  # composite class instance method
 def m2(self):

     print('Dishes class m2() method executed...')

     # calling m1() method of component class
     self.obj2.m1()



   class ingredients:

 # composite class constructor
 def __init__(self):

     # creating object of component class
     self.obj3 = Restaurant()

     print('ingredients class object also created...')
```

```
    # composite class instance method
    def m2(self):

        print('ingredients class m2() method executed...')

        # calling m1() method of component class
        self.obj3.m1()
```

[ ]:
```
115.Explain how composition enhances code maintainability and modularity in␣
↪Python programs.
```

[ ]:
```
116.Create a Python class for a computer game character, using composition to␣
↪represent attributes like
weapons, armor, and inventory.

class Character:

   # composite class constructor
    def __init__(self):
        print('Character class object created...')

    # composite class instance method
    def m1(self):
        print('Character class m1() method executed...')


class Weapons:

    # composite class constructor
    def __init__(self):

        # creating object of component class
        self.obj1 = Character()

        print('Weapons class object also created...')

    # composite class instance method
    def m2(self):

        print('Weapons class m2() method executed...')

        # calling m1() method of component class
        self.obj1.m1()


   class Armour:
```

```
    # composite class constructor
    def __init__(self):

        # creating object of component class
        self.obj2 = Character()

        print('Armour class object also created...')

     # composite class instance method
    def m2(self):

        print('Armour class m2() method executed...')

        # calling m1() method of component class
        self.obj2.m1()



      class Inventory:

    # composite class constructor
    def __init__(self):

        # creating object of component class
        self.obj3 = Character()

        print('Inventory class object also created...')

     # composite class instance method
    def m2(self):

        print('Inventory class m2() method executed...')

        # calling m1() method of component class
        self.obj3.m1()
```

[ ]:
```
117.Describe the concept of "aggregation" in composition and how it differs␣
 ↪from simple composition.


Key Difference between Aggregation and Composition
Aggregation is one type of association between two objects describing the "have␣
 ↪a" relationship, while Composition is a specific type of Aggregation which␣
 ↪implies ownership.
```

```
Aggregation is indicated using a straight line with an empty arrowhead at one␣
 ↪end. On the other hand, the composition is indicated using a straight line␣
 ↪with a filled arrowhead at any one of the ends.
In an aggregation relationship, objects that are associated with each other can␣
 ↪remain in the scope of a system without each other. But in a composition␣
 ↪relationship, objects that are associated with each other cannot remain in␣
 ↪the scope without each other.
In Aggregation, linked objects are not dependent upon the other object, whereas␣
 ↪in composition, objects are highly dependent upon each other.
In Aggregation, deleting a single element does not affect another associated␣
 ↪element. On the contrary, in composition, deleting a single element affects␣
 ↪another associated element.
Aggregation is denoted by a filled diamond, while an empty diamond denotes␣
 ↪composition.
```

[ ]:
```
118.Create a Python class for a house, using composition to represent rooms,␣
 ↪furniture, and appliances.

class House:

   # composite class constructor
    def __init__(self):
        print('House class object created...')

    # composite class instance method
    def m1(self):
        print('House class m1() method executed...')


class Rooms:

    # composite class constructor
    def __init__(self):

        # creating object of component class
        self.obj1 = House()

        print('Rooms class object also created...')

     # composite class instance method
    def m2(self):

        print('Rooms class m2() method executed...')

        # calling m1() method of component class
        self.obj1.m1()
```

```
class Furniture:

 # composite class constructor
 def __init__(self):

     # creating object of component class
     self.obj2 = House()

     print('Furniture class object also created...')

  # composite class instance method
 def m2(self):

     print('Furniture class m2() method executed...')

     # calling m1() method of component class
     self.obj2.m1()



   class Appliances:

 # composite class constructor
 def __init__(self):

     # creating object of component class
     self.obj3 = House()

     print('Appliances class object also created...')

  # composite class instance method
 def m2(self):

     print('Appliances class m2() method executed...')

     # calling m1() method of component class
     self.obj3.m1()
```

[ ]:
```
119.How can you achieve flexibility in composed objects by allowing them to be␣
 ↪replaced or modified
dynamically at runtime?
```

[ ]:
```
120.Create a Python class for a social media application, using composition to␣
 ↪represent users, posts, and
comments.
```

```
class App:

   # composite class constructor
    def __init__(self):
        print('App class object created...')

    # composite class instance method
    def m1(self):
        print('App class m1() method executed...')


class Users:

    # composite class constructor
    def __init__(self):

        # creating object of component class
        self.obj1 = App()

        print('Users class object also created...')

     # composite class instance method
    def m2(self):

        print('Users class m2() method executed...')

        # calling m1() method of component class
        self.obj1.m1()


   class Posts:

    # composite class constructor
    def __init__(self):

        # creating object of component class
        self.obj2 = App()

        print('Posts class object also created...')

     # composite class instance method
    def m2(self):

        print('Posts class m2() method executed...')

        # calling m1() method of component class
```

```
        self.obj2.m1()



      class Comments:

    # composite class constructor
    def __init__(self):

        # creating object of component class
        self.obj3 = App()

        print('Comments class object also created...')

     # composite class instance method
    def m2(self):

        print('Comments class m2() method executed...')

        # calling m1() method of component class
        self.obj3.m1()
```

[ ]:
```
55.Discuss the use of accessors and mutators in encapsulation. How do they help␣
 ↪maintain control over
attribute access?

Accessors and Mutators that are helpful in accessing and altering respectively,␣
 ↪internally stored data.


Accessor Method: This method is used to access the state of the object i.e, the␣
 ↪data hidden in the object can be accessed from this method. However, this␣
 ↪method cannot change the state of the object, it can only access the data␣
 ↪hidden. We can name these methods with the word get.

Mutator Method: This method is used to mutate/modify the state of an object i.
 ↪e, it alters the hidden value of the data variable. It can set the value of␣
 ↪a variable instantly to a new value. This method is also called as update␣
 ↪method. Moreover, we can name these methods with the word set.
```