**Day-8 Morning Assessment**

**Inheritance**

**1.Single Inheritance:**

class Person:  
   def display\_name(self):  
       print("Name: Amitha")  
  
class Student(Person):  
   pass    
s = Student()  
s.display\_name()

o/p: Name: Amitha

**2. Multilevel Inheritance**:  
class Animal:  
 def eat(self):  
 print("Animal eats")  
  
class Mammal(Animal):  
 def walk(self):  
 print("Mammal walks")  
  
class Dog(Mammal):  
 def bark(self):  
 print("Dog barks")  
  
d = Dog()  
d.eat()  
d.walk()  
d.bark()

o/p:

Animal eats

Mammal walks

Dog barks

**3. Multiple Inheritance:**  
class Flyable:  
   def fly(self):  
       print("Can fly")  
  
class Swimmable:  
   def swim(self):  
       print("Can swim")  
  
class Duck(Flyable, Swimmable):  
   pass  
  
duck = Duck()  
duck.fly()  
duck.swim()

o/p:

Can fly

Can swim  
  
**4. Hierarchical Inheritance**:  
class Vehicle:  
   def move(self):  
       print("Vehicle moves")  
  
class Car(Vehicle):  
   def drive(self):  
       print("Car drives")  
  
class Bike(Vehicle):  
   def ride(self):  
       print("Bike rides")  
  
car = Car()  
car.move()  
car.drive()  
  
bike = Bike()  
bike.move()  
bike.ride()

o/p:

Vehicle moves

Car drives

Vehicle moves

Bike rides

5. class A:  
   def show(self):  
       print("This is from class A")  
  
class B(A):  
   def show(self):  
       super().show()    
       print("This is from class B")  
  
b = B()  
b.show()

o/p:

This is from class A

This is from class B  
super() calls the parent class method even if both have methods with the same name.  
  
 6. class A:  
   def show(self):  
       print("A")  
  
class B(A):  
   def show(self):  
       print("B")  
  
class C(A):  
   def show(self):  
       print("C")  
  
class D(B, C):  
   pass  
  
d = D()  
d.show()  
print(D.mro())

o/p:

B

[<class '\_\_main\_\_.D'>, <class '\_\_main\_\_.B'>, <class '\_\_main\_\_.C'>, <class '\_\_main\_\_.A'>, <class 'object'>]  

7. class Person:  
   def \_\_init\_\_(self, name):  
       self.name = name  
  
class Student(Person):  
   def \_\_init\_\_(self, name, grade):  
       super().\_\_init\_\_(name)

self.grade = grade      
  
s = Student("Amitha", "A")  
print(s.name)  
print(s.grade)

o/p:

Amitha

A  
  
8. class Shape:  
   def area(self):  
       return 0  
  
class Circle(Shape):  
   def \_\_init\_\_(self, radius):  
       self.radius = radius  
  
   def area(self):  
       return 3.14 \* self.radius \*\* 2  
  
c = Circle(5)  
print("Area of circle:", c.area())

o/p: Area of circle: 78.5

**Polymorphism**

**9. Method Overriding**class Animal:  
   def speak(self):  
       print("Animal speaks")  
  
class Dog(Animal):  
   def speak(self):  
       print("Dog barks")  
  
class Cat(Animal):  
   def speak(self):  
       print("Cat meows")  
  
a = Animal()  
d = Dog()  
c = Cat()  
a.speak()  
d.speak()  
c.speak()

o/p:

Animal speaks

Dog barks

Cat meows  
  
**10. Polymorphic Behavior**:  
class Cow(Animal):  
   def speak(self):  
       print("Cow moos")  
  
  
print("\n Polymorphic Behavior:")  
animals = [Dog(), Cat(), Cow()]  
for animal in animals:  
   animal.speak()  
Python calls the appropriate method at runtime — this is runtime polymorphism.  
  
**11. Simulated Method Overloading:**   
Python doesn't support method overloading, but we can mimic it.  
class Demo:  
   def show(self, \*args):  
       if not args:  
           print("No arguments")  
       else:  
           print("Arguments:", args)  
  
d = Demo()  
d.show()  
d.show(10)  
d.show("A", "B", "C")

o/p:

No arguments

Arguments: (10,)

Arguments: ('A', 'B', 'C')  
  
12. class Calculator:  
   def add(self, a, b, c=0):  
       return a + b + c  
  
class FlexibleCalculator:  
   def add(self, \*args):  
       return sum(args)  
  
print("\nCalculator Add:")  
calc = Calculator()  
print("Sum of 2 args:", calc.add(10, 5))  
print("Sum of 3 args:", calc.add(10, 5, 2))  
  
flex = FlexibleCalculator()  
print("Flexible sum (2 args):", flex.add(1, 2))  
print("Flexible sum (3 args):", flex.add(1, 2, 3))

o/p:

Calculator Add:

Sum of 2 args: 15

Sum of 3 args: 17

Flexible sum (2 args): 3

Flexible sum (3 args): 6  
  
13. class Book:  
   def \_\_init\_\_(self, title, author):  
       self.title = title  
       self.author = author  
  
   def \_\_str\_\_(self):  
       return f"'{self.title}' by {self.author}"  
  
  
print("\nOverride \_\_str\_\_():")  
b = Book("1984", "George Orwell")  
print(b)

o/p: Override \_\_str\_\_():

'1984' by George Orwell

14. class Car:  
   def start(self):  
       print("Car engine started")  
  
class Boat:  
   def start(self):  
       print("Boat engine started")  
  
def start\_engine(vehicle):  
   vehicle.start()  
  
start\_engine(Car())  
start\_engine(Boat())

o/p:

Car engine started

Boat engine started  
 No need to check types — works as long as the object has a start() method.  
  
15. class PDF:  
   def open(self):  
       print("Opening PDF file")  
class WordDoc:  
   def open(self):  
       print("Opening Word document")  
def open\_file(doc):  
   doc.open()  
docs = [PDF(), WordDoc()]  
for d in docs:  
   open\_file(d)

o/p: Opening PDF file

Opening Word document  
 Polymorphism lets us use the same function (open\_file) to work with any object that has an open() method.