Day – 8 Morning Assessment:

1. class Person:

def \_\_init\_\_(self,name):

self.name = name

def display\_name(self):

print(“Name : {self.name}”)

class Student(Person):

def display\_name(self):

super().display\_name()

print(“This is a student”)

st = Student(“Ganesh”)

st.display\_name()

1. class Animal:

def behavior(self):

print(“This is an Animal”)

class Mammal(Animal):

def behavior(self):

super().behavior()

print(“This is a mammal too.”)

class Dog(Mammal):

def behavior(self)”

super().behavior()

print(“It is called Dog”)

d = Dog()

d.behavior()

1. class Flyable:

def fly(self):

print(“can fly”)

class Swimmable:

def swim(self):

print(“can swim”)

class Duck(Flyable, Swimmable):

def action():

print(“Duck”)

super().fly()

super().swim()

d = Duck()

d.action()

1. class Vehicle:

def name(self):

print(“This is a vehicle”)

class Car(Vehicle):

def name(self):

super().name()

print(“called car”)

class Bike(Vehicle):

def name:

super().name()

print(“called Bike”)

c = Car()

b = Bike()

c.name()

b.name()

1. class Person:

def \_\_init\_\_(self,name):

self.name = name

def display\_name(self):

print(“Name : {self.name}”)

class Student(Person):

def display\_name(self):

super().display\_name()

print(“This is a student”)

st = Student(“Ganesh”)

st.display\_name()

As we use super(), we have inherited and have access to use the inherited attributes from the parent class. If both the methods have the same name, then first it prints the data from the inherited method as we used super() first and then it prints the additional data from the child method.

1. When a class inherits attributes and methods from 2 different classes with same names but different values, then the child class faces a situation where it has to choose a single class to inherit the properties. This situation is known as Diamond Problem Structure. To tackle this problem MRO comes into picture. It defines an order to follow and inherit the properties.

Eg:

class A:

def show(self):

print("class A")

class B(A):

def show(self):

print("class B")

class C(A):

def show(self):

print("class C")

class D(B,C):

def show(self):

pass

obj = D()

obj.show()

obj1 = C()

obj1.show()

1. class A:

def \_\_init\_\_(self,name):

self.name = name

def show(self):

print(f"{name}”)

class B(A):

def \_\_init\_\_(self,age):

super().\_\_init\_\_()

self.age = age

def show(self):

print(f”{name}:{age}")

a = A()

b=B()

a.show()

b.show()

1. class Shape:

def area(self):

print(“No area is defined”)

class Circle(Shape):

def \_\_init\_\_(self,radius):

self.radius = radius

def area(self):

print(f"Area : {2\*3.14\*radius\*radius}”)

s = Shape()

c = Circle()

s.area()

c.area()

Polymorphism

1. class Animal:

def speak(self):

print(“Animal can speak”)

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()

1. class Animal:

def speak(self):

print(“Animal can speak”)

class Dog(Animal):

def speak(self):

super().speak()

print(“Dog Barks”)

class Cat(Animal):

def speak(self):

super().speak()

print(“Cat Meows”)

a= Animal()

d = Dog()

c = Cat()

lis = [“a”,”d”,”c”]

for i in lis:

i.speak()

1. class Shape:

def area(self):

print(“No area is defined”)

class Square(Shape):

def \_\_init\_\_(self,side):

self.side = side

def area(self):

print(f"Area : {side\*side}”)

s = Shape()

sq = Square()

s.area()

sq.area()

1. class Calculator:

def add(\*args):

if len(\*args)>3:

raise ValueError(“ Enter 3 or less than 3 parameters)

return(sum(\*args))

calc = Calculator()

calc.add(3,6,9)

calc.add(4,2,5,4)

1. Yes, we can override the \_\_str\_\_() in python

class Book:

def \_\_str\_\_(self):

return f”Hello, this is Book!”

b = Book()

b.\_\_str\_\_()

14. class Car:

def start(self):

print(“Car started”)

class Bike:

def start(self):

print(“Bike started”)

def start\_engine(vehicle):

vehicle.start()

c = Car()

b = Bike()

start\_engine(c)

start\_engine(b)

15. So, in Python, Polymorphism helps us in writing more generic functions because, in polymorphism, we use same method name in different classes but with different parameters like different no.of parameters or different data type parameters, etc.. So, based on the type of parameters the user gives, the interpreter choses the method to execute.

Ex:

class Shape:

def area(self):

print(“No area is defined”)

class Square(Shape):

def \_\_init\_\_(self,side):

self.side = side

def area(self):

print(f"Area : {side\*side}”)

s = Shape()

sq = Square()

s.area()

sq.area()

Here, if user input, the output will be displayed. So, if user give any parameter then area in square class would be executed, if not, area from the shape class will be executed.