9/6/25

UNIT-5:

Object Oriented Programming (Oop) Concepts:

- Oop in python is a way of structuring your code using classes and objects to model real volt entities.
- Following are the Oop concepts:

1. class
2. object
3. encapsulation
4. Polymorphism
5. Inheritance
6. Abstraction

1. class:

- A class is a blue print, for creating objects.

Syntax:
```
class className:
    def __init__(self, attribute):
        self.attribute = value
    def method(self):
        pass
```

2. object:

- object is an instance of the class.
- Once the class is defined, next job is to create object
- The object can access class variables and class methods using (.) operator.

Syntax: object_name = classname()

Ex:
```
class Dog:
    def __init__ (self, name):
        self.name = name
    def bark(self):
        print(f"{self.name} says woof!")
my_dog = Dog("Buddy")
my-dog.bark() #output: Buddy says woof!
```

__init__ method :

-This method automatically executed when an object of a class is created.

-This method is useful to initialise variable of class object.

self:

-self is reference to current object.

-It is how an object refers to its own attributes and methods.

Ex1: class variables :

```
class Car :
        a=10 # class variable
        def __init__(self, name):
                self.name=name
        def display(self):
                print("This is car")
c= Car("ok")
print(c.a)
c.display()
```

o/p: 10
      This is car

Ex2: object variable :

```
class Car:
        def __init__(self, name):
        self.name=name
        self.a = 20  #object variable
        def display(self):
                print("This is car")
c= Car("ok")
print(c.a)
c.display()
```

Task17: a) WAP to create a class that perform basic calculator operations.

```
class Calculator:
      def __init__(self a,b):
            self.a=a
            self.b=b
      def add(self):
            print("Addition=", self.a+self.b)
      def substract(self):
            print("Substraction=", self.a-self.b)
      def mul(self):
            print("Multiplication=", self.a*self.b)
      def div(self):
            if self.b != 0:
                  print("Division=", self.a/self.b)
            else:
                  print("ZeroDivisionError")
num1 = int(input("Enter first number:"))
num2 = int(input("Enter second number:"))
calc=Calculator(num1,num2)
calc.add()
calc.substract()
calc.mul()
calc.div()
```

Task17: b) WAP to create a class in which one method accepts a string from user and and/or print it

```
class Message:
    def inputstr(self):
        self.text=input("Enter string:")
    def printmsg(self):
        print(f"Message is:{self.text}")
msg= Message()
msg.inputstr()
msg.printmsg()
```

Ex:-

```
class Company:
    def __init__(self,name,age,salary):
        self.name=name
        self.age=age
        self.salary=salary
    def printmsg(self):
        print(f"Employee Name:{self.name}")
        print(f"Employee Age:{self.age}")
        print(f"Employee salary:{self.salary}")
e1= Company("Pooja",18,80,000)
e2= Company("Navya",18,85,000)
e3 = Company("Srinidhi",17,75,000)
```

Destructor Method:

--del-- is the destructor method. It is called automatically when an object is about to be destroye

- If object is going out of space, this method will be automatically called.
- When destructor method is called, object occupied resour are returned back to the system.

Ex:

```
class Car:
        def --init(self,name):
                self.name=name
                print(f"car {self.name} is created")

        def --del--(self):
                print(f"car {self.name} is destroyed")

c= Car("Skoda")
del c # Explicitly calling the destructor
```

O/P:

```
car Skoda is created
car Skoda is destroyed
```

3. Encapsulation:

- The process of hiding internal details of an object and only exposing what is necessary.
- It helps to protect the data and control how it is accessed or changed.

Ex:

```
class Person:
        def --init--(self,name):
                self.name =name          # Public
                self._email = "hidden"  # Protected
                self.--ssn= "123-45-6789  # Private

p=Person ("CVR")
```

```
print(p.name)          # Allowed (public)
print(p._email)        # Possible, but not recommended
print(p._Person__ssn)  # Works, but breaks encapsulation
```

4. Polymorphism:

- It allows objects of different classes to be treated as objects of common superclass.
- Polymorphism is an ability of different objects to respond to same function or method called in different ways.

<u>Ex:</u>
```
class Bird:
        def sound(self):
                print("Bird makes a sound")

   class Dog:
        def sound(self):
                print ("Dog barks")
  def make_sound(animal):  → # polymorphic function
        animal.sound()
 # Create objects
 b = Bird()
 d = Dog()
 # Make a function call
 make_sound(b)
 make_sound(d)
```

O/P:
```
Bird makes a sound
Dog barks
```

5. Inheritance:

- Inheritance allows child class (sub class) to inherit properties and methods from parent class (super class)

Ex:
```
class A:  # parent class
    def sound(self):
        print("This is A")
class B(A):  #child class
    def display(self):
        print("This is B")
b1 = B()
b1.sound()
b1.display()
```

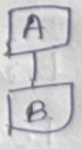
O/P:
```
This is A
This is B
```

Types of Inheritance:

1) Single Inheritance
2) Multiple Inheritance
3) Multilevel Inheritance
4) Hierarchical Inheritance
5) Multipath Inheritance

1) Single Inheritance:

- A child class inherits from only one parent class.

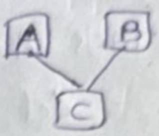


Ex:
```
class Animal:
    def speak(self):
        print("Animal speaks\n")
class Dog(Animal):
    def bark(self):
        print("Dog barks\n")
d = Dog()
d.speak()
d.bark()
```

O/P:
```
Animal speaks
Dog barks
```

2) Multiple Inheritance:

- A child class inherits from more than one parent class.

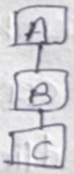


Ex:
```
class Engine:
        def start(self):
                print("-Engine starting")
    class Radio:
        def play_music(self):
                print("Playing music")

    ~~my_car = Car~~
    class Car(Engine, Radio):
            pass
    my_car = Car()
    my_car = start()
    my_car.play_music()
```

O/p: Engine starting
Playing music

3) Multilevel Inheritance:

- A class inherits from child class which itself inherits from another parent class.

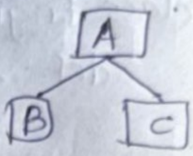


Ex:
```
class Animal:
        def speak(self):
                print("Animal speaks")

    class Dog(Animal):
            def bark(self):
                    print("Dog barks")
    class Puppy(Dog):
            def weep(self):
                    print("Puppy weeps")

    p=Puppy()
    p.speak()
    p.bark()
    p.weep()
```

O/p: Animal speaks
Dog barks
Puppy weeps

4) Hierarchical Enheritance:

- Multiple child classes inherits from single parent class.



Ex:
```
class A:
        def display(self):
                print("class A")
    class B(A):
        def display(self):
                print("class B")
    class C(A):
        def display(self):
                print("class B")
    b=B()
    c=C()
    b.display()
    c.display()
```

o/p:
```
class A
class A
```

5) Multipath Inheritance:

- A combination of 2 or more types of inheritance. It lea to complex relationships. It may requires careful handling of method resolution order (mro).

Ex:
```
class A:
    def show(self):
            print("from class A")
class B(A):
    def show(self):
            print("from class B")
class C(A):
    def show(self):
            print("from class C")
class D(B,C):
    def show(self):
            print("from class D")
```

```
d.D( )
d.show()
```

O/p: from class D

Ex:
```
class A:
        def show(self):
                print("from class A")
   class B(A):
        def show(self):
                super().show()
                print("from class B")
   class C(A):
        def show(self):
                super().show()
                print("From class C")
   class D(B,C):
        def show(self):
                super().show()
                print("from class D")
  d=D()
  d.show()
  print(D.__mro__)
```

O/p:
```
From class A
from class C
from class B
```

Abstraction:

- Principles that hides internal details and only show essential features of an object.
- It enhances code reusability and encourages modular design.
- Python achieves abstraction through abstract base classes and interfaces using ABC(Abstract Base class)
- ABC is the baseclass for defining abstract classes.
- @ abstact method is a decorator to mark methods that must be implemented in child classes.
- Abstract class cannot be instantiated directly.

Ex:

```
from abc import ABC, abstract method
class vehicle(ABC):
        @ abstractmethod
        def start_engine(self):
              pass
class   car (vehicle):
        def start_engine(self):
              print("car engine started with a key")
class Bike( vehicle):
        def start_engine(self):
              print('Bike engine started with a button')
V1= Car()
V2= Bike()
V1. start_engine()
V2. start_engine()
```

Shallow copy:

- It creates new object but inserts references to the objects found in the original.
- changes to mutable nested objects will reflect in both original and its copy.

Ex:
```
import copy
original = [[1,2], [3,4]]
shallow=copy.copy(original)
shallow[0][0]=100
print("Original = ",original)
print("shallow= ", shallow)
```

o/p:
```
original =[[100, 2] ,[3,4]]
shallow = [[100,2],[3,4]]
```

Deep copy:

- It creates new object and recursively copies all objects inside the original.
- changes to nested objects do not affect the original.
- Everything is independent.

Ex:
```
import copy
original =[[1,2],[3,4]]
deep=copy.deepcopy(original)
deep[0][0]=100
print(" original= ",original)
print(" deep= ", deep)
```

o/p:
```
original = [[1,2], [3,4]]
deep = [[100,2], [3,4]]
```