PW Assignment Oops (26th-sept)

1. What is Object-Oriented Programming (OOP)?

Ans> OOP is a programming paradigm (style of coding) that is based on the concept of objects.
An object represents a real-world entity (like a car, student, bank account, employee, etc.) and combines data (attributes) and functions (methods) into one unit.

Instead of writing everything in functions or procedures (as in procedural programming), OOP helps you structure code in a more modular, reusable, and real-world way.

Concept Opps -> Class, Object, Encapsulation, Inheritance, Abstraction, Polymorphisum.

2. What is a class in OOP?

Ans> Class is a blueprint of Object or creating object

```
Class Car :
    Pass
#Blueprint of car Object
C1 = Car() # Object of Car Class
```

3. What is an object in OOP?

Ans>Object is a real-world physical entity . E.g., Car, Mobile, Laptop, Humans etc.

4. What is the difference between abstraction and encapsulation?

Ans>

Aspect	**Abstraction**	**Encapsulation**
Definition	Hiding implementation details and showing only essential features.	Wrapping data (variables) and methods into a single unit and restricting direct access.
Focus	Focuses on **what** the object does.	Focuses on **how** the data is hidden and controlled.
Achieved By	Using **abstract classes** and **interfaces**.	Using **access modifiers** (public, protected, private).
Level	**Design level** concept.	**Implementation level** concept.
Real-life Example	Driving a car → you know to use the steering and pedals, but not how the engine works.	Bank account → your balance is private, only accessible via deposit/withdraw methods.
In Python	Achieved using abc module (@abstractmethod).	Achieved using naming conventions like __private_var.

5. What are dunder methods in Python?

Ans> Dunder method are the method defined by built-in class in python.

Classes define these type of method for creating object implementing operators overloading in python

All method surrounding with double underscore is a Dunder method.

a = “Ankur”

b=”Verma”

a.__add__(b) # Ankur Verma

6. Explain the concept of inheritance in OOP?

Ans> Inheritance is a process of child class receiving the properties of parent class.

<u>Types of Inheritance:</u>

1.Single Level : Child class acquire the properties of single parent.

```
class Parent:
    Body:
Class Child(Parent):
    Body:
```

2.Multi Level : Where child class acquire the property of parent and grand parent also.

```
Class Grandparent:
    Body:
Class Parent(Grandparent):
    Body:
Class Child(Parent):
    Body:
```

3.Multiple : One child having multiple parent.

```
Class p1:                    Class p2:
    Body :                       Body:
```

```
Class child(p1,p2):
    Body:
```

It can cause Dimond problem to remove this in python use method resolution order (MRO) , The class inherited 1st that method will be called.

4.Hierarchical Inheritance : Single parent multiple child.

$$\begin{tikzcd}
 & \text{Parent} \arrow[dl] \arrow[d] \arrow[dr] & \\
\text{Child1} & \text{Child2} & \text{Child3}
\end{tikzcd}$$

5.Hybrid Inheritance: Combination of more than one inheritance.

$$\begin{tikzcd}
 & \text{Base Class} \arrow[d] & \\
 & \text{Parent} \arrow[dl] \arrow[d] \arrow[dr] & \\
\text{Child1} & \text{Child2} & \text{Child3}
\end{tikzcd}$$

Sigle Level Between Base Class and Parent class

Hierarchical between Parent and Child1,Child2 and Child3

7. What is polymorphism in OOP?

Ans> Any thing which can take multiple form is known as polymorphism .

We can achieve it using Method Overloading and Method Overriding.

Method Overloading:

```
Class Student:
            def Student(Self):
                    Print(“Welcome”)
            def Student(Self, name):
                    Print(“Welcome”, name)
            def Student(Self, name=””, Rollno =””):
                    Print(“Welcome”, name, Rollno)



Stud = student()
Stud.student() # Welcome
Stud.student(“Ankur”) # Welcome Ankur
Stud.student(“Ankur”,9) # Welcome Ankur 9
```

Method Overriding:

```
Class Animal:
    def sound(self):
            Print(“Sound”)

Class Cat(Animal):
    def sound(self):
            Print(“Meows”)

a = Animal()
a.sound # Sound
c = cat()
```

```
c.sound # Meows
```

8. How is encapsulation achieved in Python?

Ans> Binding Data member and method of a class.

We can achieve it using Modifiers like Public, Protected and private.

In Python, we don't have strict keywords like private or protected (like Java/C++), but it's achieved using **naming conventions**.

```
class Student:
    def __init__(self, name):
        self.name = name   # Public attribute

s = Student("Ankur")
print(s.name)   # Ankur

class Student:
    def __init__(self, name):
        self._name = name   # Protected attribute

s = Student("Ankur")
print(s._name)  # Can access, but not recommended

class Student:
    def __init__(self, name):
        self.__name = name   # Private attribute
    def get_name(self):
        return self.__name   # Controlled access

s = Student("Ankur")
# print(s.__name)    # AttributeError
print(s.get_name())  # Ankur

# Still can be accessed using name mangling (not recommended)
print(s._Student__name)  # "Ankur"
```

9. What is a constructor in Python?

Ans> 1. The **constructor is __init__()** in Python.

2.It is called **automatically** when you create an object.

3. It is used to **assign initial values** to object attributes.

4.A class can have only **one __init__ constructor** (if you define multiple, the last one overrides).

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

# When object is created, constructor is called automatically
s1 = Student("Ankur", 21)
s2 = Student("Riya", 20)
```

```
print(s1.name, s1.age)  # Ankur 21
print(s2.name, s2.age)  # Riya 20
```

10. What are class and static methods in Python?

Ans> @classmethod > the method that makes the class itself as the 1st arguments .

It takes reference to the class it self to modify and access the class level arguments.

```
Class Math:
    @classmethod
    def add(cls ,x ,y):
        return cls.__name__,x+y
```

you don’t need inti method to take data

Math.add(3,5) # (Math,8)

Class method is bound to class and not the instance of the class

Class it self is 1st arguments.

Static Method

@staticmethod -> the method which can be called without creating any instance of a class and without using any self or cls.

```
Class Math:
    @staticmethod
    def add(x,y)
        return x+y
Math.add(2,3) # 5
```

11. What is method overloading in Python?

Ans> Method Overloading : performing the same tasks using different inputs is called as method overloading.

```
Class Mobile:
    def unloack(self,PIN):
        print(“PIN required to unlock”)
    def unloack(self, FaceID):
        print(“FaceID required to unlock”)
    def unloack(self, Fingerprint):
        print(“Fingerprint required to unlock”)
```

12. What is method overriding in OOP?

Ans> Method overriding happens when a **child class (subclass)** defines a method with the **same name, parameters, and return type** as a method in its **parent class (superclass)**.

The **child’s method replaces (overrides)** the parent’s method when called using a child object.

```
class Payment:
  def pay(self):
    print("Processing generic payment")

class CreditCardPayment(Payment):
  def pay(self):
    print("Processing payment via Credit Card")

class UpiPayment(Payment):
  def pay(self):
        print("Processing payment via UPI")


# Objects
p1 = CreditCardPayment()
p2 = UpiPayment()


p1.pay()   # Processing payment via Credit Card
p2.pay()   # Processing payment via UPI
```

13. What is a property decorator in Python?

Ans> The **@property decorator** allows you to use **methods like attributes**.

- It is used for **getter, setter, and deleter** methods.
- It helps in **encapsulation** (hiding data) while giving a **clean interface**.

  @property → defines a getter (access like an attribute).
  @<propertyname>.setter → defines a setter for validation/modification.
  @<propertyname>.deleter → defines what happens when the property is deleted.
  Provides a clean interface (no need to call methods explicitly).

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

    @property
    def name(self):        # getter
        return self._name

    @name.setter
    def name(self, value):  # setter
```

```
        if len(value) < 3:
            raise ValueError("Name is too short!")
        self._name = value

    @name.deleter
    def name(self):        # deleter
        print("Deleting name...")
        del self._name

# Usage
s = Student("Ankur")
print(s.name)      # Calls getter like an attribute
s.name = "Riya"     # Calls setter
print(s.name)
del s.name         # Calls delete
```

14. Why is polymorphism important in OOP?

Ans> **Polymorphism = "many forms"**
It allows the **same method name or operator** to behave **differently** depending on the object or context.

Code Reusability – One function/method can work with different object types.

- **Flexibility** – Same method name gives different behavior based on the object.
- **Extensibility** – New classes can be added without changing existing code.
- **Runtime Decision Making** – Method behavior is decided at **runtime** (dynamic).
- **Cleaner Code** – Avoids long if-else conditions; just call the same method.
- **Operator Overloading** – Same operator (+, *, etc.) works with different data types.
- **Consistency** – Provides a common interface for different objects (e.g., all shapes have draw()).

15. What is an abstract class in Python?

Ans> An **abstract class** in Python is a class that **cannot be instantiated directly** and is designed to **serve as a blueprint** for other classes. It may contain **abstract methods**, which are methods that are declared but **must be implemented by any subclass**. Abstract classes are used to enforce a certain structure in subclasses.

In Python, abstract classes are implemented using the abc module.

```
from abc import ABC, abstractmethod

class Animal(ABC):

    @abstractmethod

    def sound(self):

        pass


a = Animal()  # This will raise an error


class Dog(Animal):

    def sound(self):

        return "Bark"


d = Dog()

print(d.sound())  # Output: Bark
```

16. What are the advantages of OOP?

- Ans> **Modularity** – Organizes code into classes/objects, making it easier to maintain.
- **Reusability** – Classes can be reused in other programs via inheritance.

- **Scalability** – Easier to add new features and handle large programs.
- **Encapsulation** – Protects data by combining attributes and methods in a single unit.
- **Abstraction** – Hides complexity and shows only necessary details.
- **Polymorphism** – Allows objects of different classes to be treated uniformly.
- **Maintainability** – Easy to update or modify code without affecting other parts.
- **Real-World Modeling** – Helps simulate real-world entities in programs.

17. What is the difference between a class variable and an instance variable?

Ans>

Feature	**Class Variable**	**Instance Variable**
Definition	A variable **shared by all instances** of a class.	A variable **unique to each instance** of a class.
Declaration	Declared **inside the class** but **outside any method**.	Declared **inside a method**, usually __init__(), using self.
Memory	Stored **once** in memory, shared across all objects.	Each object gets its **own copy** in memory.
Access	Can be accessed via **class name** or object.	Accessed via **object only**.

Feature	Class Variable	Instance Variable
Modification Effect	Changing it via class affects **all instances** (unless overridden in an instance).	Changing it affects **only that specific object**.
Use Case	For data that should be **common to all objects** (e.g., company_name).	For data that is **specific to each object** (e.g., employee_name).

18. What is multiple inheritance in Python?

Ans> Types of Inheritance:

1.Single Level : Child class acquire the properties of single parent.

```
class Parent:
   Body:
Class Child(Parent):
   Body:
```

2.Multi Level : Where child class acquire the property of parent and grand parent also.

```
Class Grandparent:
   Body:
Class Parent(Grandparent):
   Body:
Class Child(Parent):
   Body:
```

3.Multiple : One child having multiple parent.

```
Class p1:                    Class p2:

   Body :                       Body:


         Class child(p1,p2):

               Body:
```

It can cause Dimond problem to remove this in python use method resolution order (MRO) , The class inherited 1st that method will be called.

4.Hierarchical Inheritance : Single parent multiple child.

$$\begin{tikzcd}
 & \text{Parent} \arrow[dl] \arrow[d] \arrow[dr] & \\
\text{Child1} & \text{Child2} & \text{Child3}
\end{tikzcd}$$

5.Hybrid Inheritance: Combination of more than one inheritance.

$$\begin{tikzcd}
 & \text{Base Class} \arrow[d] & \\
 & \text{Parent} \arrow[dl] \arrow[d] \arrow[dr] & \\
\text{Child1} & \text{Child2} & \text{Child3}
\end{tikzcd}$$

Sigle Level Between Base Class and Parent class

Hierarchical between Parent and Child1,Child2 and Child3

19. Explain the purpose of ‘’__str__’ and ‘__repr__’ ‘ methods in Python?

Ans> __str__ -> Will return a String representation of method.

```
Class Student:

    def init(self):
        self.phone=9999999
def __str__(self):
        return “This is __str__ method”


Student() # <_main_.Student at 0x7……………..>


Print(Student()) # This is __str__ method
```

__repr__ -> it will return unambiguous String representation of the object as it is, that can be used to recreate the object.

```
Class MyClass:
        def __init__(self,x):
                self.x= x
        def __rerp__(self):
                return f “My class({self.x})
        obj = MyClass(5)
        print(rerp(obj)) -> MyClass(5)
```

20.What is the significance of the ‘super()’ function in Python?

Ans> The **super()** function in Python is used to **call methods from a parent (or superclass) in a child (subclass)**. It is especially useful in **inheritance** to avoid explicitly naming the parent class and to ensure proper method resolution in multiple inheritance scenarios.

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

```
class Child(Parent):
    def __init__(self, name, age):
        super().__init__(name)  # Initialize Parent attributes
        self.age = age
```

21. What is the significance of the __del__ method in Python?

Ans> The **__del__ method** in Python is called a **destructor**. It is a special method that is invoked **when an object is about to be destroyed** or garbage collected. Its main purpose is to **perform cleanup actions**, like releasing resources (files, network connections, database connections) before the object is removed from memory.

```
class Person:
    def __init__(self, name):
        self.name = name
        print(f"{self.name} is created.")

    def __del__(self):
        print(f"{self.name} is destroyed.")

# Usage
p1 = Person("Alice")
p2 = Person("Bob")

del p1  # Calls __del__ for Alice
```

22. What is the difference between @staticmethod and @classmethod in Python?

Ans>

Feature	**@staticmethod**	**@classmethod**
Access to class/instance	Cannot access instance (self) or class (cls).	Can access the class (cls) but not instance (self).
Method parameter	No default parameter; behaves like a regular function inside a class.	Takes cls as the first parameter, referring to the class.
Use case	For utility functions related to a class but **not dependent** on class or instance data.	For methods that **need to access or modify class state**.
Callable via	Both Class.method() and instance.method().	Both Class.method() and instance.method().
Decorator	@staticmethod	@classmethod

23. How does polymorphism work in Python with inheritance?

Ans> Polymorphism means **“many forms”** — the same method or operator behaves **differently for different objects**.
In Python, polymorphism with inheritance allows a **subclass to override a method of its parent class**, and the correct method is called depending on the **object type**.

```
class Animal:
    def speak(self):
        print("Animal makes a sound")

class Dog(Animal):
    def speak(self):
        print("Dog barks")

class Cat(Animal):
    def speak(self):
        print("Cat meows")

# Using polymorphism
animals = [Dog(), Cat(), Animal()]

for a in animals:
    a.speak()
```

Output

```
Dog barks
Cat meows
Animal makes a sound
```

Dog and Cat inherit from Animal.

Both **override** the speak() method.

When we loop through animals, Python **calls the appropriate method** based on the **actual object type**, not the reference type.

This is called **runtime polymorphism** or **dynamic dispatch**.

24. What is method chaining in Python OOP?

Ans> **Method chaining** in Python OOP is a programming technique where **multiple methods are called on the same object in a single line**, one after another. Each method **returns the object itself** (self), allowing the next method to be invoked immediately.

---

1. Improves **code readability** and **compactness**.
2. Requires methods to **return self**.
3. Commonly used in **fluent interfaces** (e.g., configuring objects, building queries).

25. What is the purpose of the __call__ method in Python?

Ans> The **__call__** method in Python allows an **instance of a class to be called like a function**. By defining __call__, you can make objects behave like **callable functions**, which can be useful for flexible and intuitive interfaces.

Make objects callable like regular functions.

Encapsulate behavior inside an object while allowing functional syntax.

Useful in **decorators, functional programming, or creating configurable objects**.

```
class MyDecorators :
    def __init__(self,func):
        self.func=func
    def __call__(self):
        print("Something is happening before function")
        self.func()
        print("Something is heppening after function")
```

```
@MyDecorators
def say_hello():
    print("Hello")

say_hello()
```