**OOPS Assignment**

**1. What is Object-Oriented Programming (OOPS)?**

**A.** In python, Object oriented programming is programming paradigm that uses class and objects to create modular programs. there are many concepts such as -

1. Classes and Objects
2. Polymorphism
3. encapsulation
4. inheritance
5. abstraction

**2.Explain the concept of Polymorphism in OOP with an example.**

**A.** In python, polymorphism is a concept of OOPs.

Polymorphism means “many forms.”

- `same function is being used for different types `

 # for example,

len("abcd") --> 4

len([1,2,3,4,5]) --> 5

- It is the ability of an object to behave differently depending on the context.

- In Python, polymorphism can be achieved by using methods that have the same name but different implementations in different classes.

- For example,

    - the `bark()` method can have different sounds for different subclasses of Dog.

    - the `+ operator` can perform addition or concatenation depending on the operands.

class Cat:

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

self.name = name

self.age = age

def make\_sound(self):

print("Meow")

class Dog:

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

self.name = name

self.age = age

def make\_sound(self):

print("Bark")

cat1 = Cat("Kitty", 2.5)

dog1 = Dog("Fluffy", 4)

for animal in (cat1, dog1):

animal.make\_sound()

**3.How does Encapsulation enhance security in OOP? Provide an explanation.**

**A.** Encapsulation is a technique of hiding some internal data of an object from the outside world. This helps to enhance security.

In Python, we can use underscores (\_) to denote private attributes or methods of a class.

- It's like putting a protective bubble around object. it keeps the data of the object works hidden, only show what necessary.

- example,

- `\_name` is a private attribute of the `class Dog` that cannot be accessed directly by other classes or objects.

When we implement encapsulation, we declare the fields in the class as private to prevent other classes from accessing them directly. However, the encapsulated data can only be accessed using public get () and set () methods. As a result, encapsulation confers several advantages: data security, code flexibility, and application maintainability.

**4.What is Inheritance in OOP? Give an example to illustrate its usage.**

**A.** Inheritance in OOPs is the mechanism of creating a new class from an existing class, inheriting its attributes and methods.

This allows code reuse and polymorphism.

The new class is called a `subclass` or `child class`, and the existing class is called a `superclass` or `parent class`.

- Inheritance can be achieved by using parentheses after the class name and specifying the superclass name.

    - For example,

parent class: class Dog():

child class : class Labrador(Dog):

here, Labrador() inherits most of the characters from Dog() class.

- It's like passing on family traits. a new object can be inheriting the characteristics of old object.

    - ``example``, new launched car(new) and general car(old), most of the characteristics are inherits from general car to new launched car.

**5.Explain the concept of Abstraction in OOP and its importance in software development.**

**A.** Abstraction in Python’s Object-Oriented Programming (OOP) is a crucial concept that allows programmers to hide complex implementation details while exposing only essential information and functionalities to users.

- In Python, abstraction can be achieved by using abstract classes and methods that are not implemented but only defined.

from abc import ABC, abstractmethod

- For example, `class Animal(abc.ABC):` creates an abstract class named Animal that cannot be instantiated.

\* `Abstraction` focuses on `what an object does` rather than `how it does it`.

\* It provides a `high-level view` of an object’s functionality, making it easier to understand and use without getting bogged down in internal details.

- `For example`, when you drive a car, you don’t need to know the intricate workings of the engine, transmission, or braking systems. You interact with the car as a whole, abstracting away the underlying complexity.

﻿Importance of Abstraction:

- Enables modular and well-organized code.

- Promotes code reuse.

- Improves developer collaboration.

**6.Write a Python code example using decorators to log the time taken by a function to execute.**

**A.**

import time

def time\_tester(func):

    def time\_log():

        start\_time = time.time()

        print(start\_time)

        func()

        end\_time = time.time()

        print(end\_time)

    return time\_log()

@time\_tester

def add():

    print(4+5) # 1.29 9 0.0

**7.Differentiate between class methods and static methods in Python.**

**A.** Difference between class methods and static methods in python are:

1. Class methods take 'cls' as the first parameter which refers to the class. Static methods don’t take any specific parameters.
2. Class methods can modify class state, but static method can’t modify class state,
3. Class methods are useful for factory methods that return class objects for different use cases. static methods serve as utility functions  that don’t rely on instance-specific data.

Similarity:

* A class method is associated with the class itself, not with a specific instance (object). A static method is also bound to the class itself, not to any specific instance.

**8.What are special (magic/dunder) methods in Python classes? Provide examples of their usage.**

**A.** `Magic methods`, also known as `dunder methods` (short for “`double underscore`”), are special methods in Python that start and end with double underscores. These methods provide a way to customize the behavior of built-in Python operations and make user-defined classes more intuitive and powerful.

* Magic methods are most frequently used to define `overloaded behaviors` of predefined operators in Python. For instance, arithmetic operators by default operate upon numeric operands. To make this overloaded behavior available in your own custom class, you need to `override the corresponding magic method`.
* For example, to use the `+` operator with objects of a user-defined class, you should include the` \_\_add\_\_()` method in your class.

**9.Explain the usage and significance of property decorators in Python.**

**A.** property decorator is a built-in decorator for the `property()` function in Python.

- which can expose your property of the class to the outer world.

- We use it to give special functionality to certain methods, making them act as:

    - `Getters (fget)`: To retrieve the value of an attribute.

    - `Setters (fset)`: To set the value of an attribute.

    - `Deleters (fdel)`: To delete an instance attribute.

To define a property, follow this pattern:

class Portal:

    # Defining \_\_init\_\_ method

    def \_\_init\_\_(self):

        self.\_\_name ='' # private variable

    # 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 = 'ineuron'

# Prints name

print (p.name) ## output: ineuron

# Deletes name

del p.name

Significance of @property:

* Encapsulation: It allows you to encapsulate attribute access and modification logic within the class.
* Read-Only Properties: You can create read-only attributes by omitting the setter method.
* Validation: You can validate input values before setting an attribute (e.g., checking temperature bounds).
* Cleaner Syntax: Accessing properties feels like accessing regular attributes (no need for parentheses).

**10.Why are getters, setters, and deleters used in Python? Provide a practical example to illustrate their importance.**

**A.**  Getters : To retrieve the value of an attribute.

* Use : They ensure data encapsulation by providing controlled access to an attribute. Instead of directly accessing the attribute, users call the getter method.

class Student:

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

self.\_name = name

self.\_age = age

def get\_name(self):

return self.\_name

def get\_age(self):

return self.\_age

student1 = Student("Alice", 20)

print(student1.get\_name()) # Output: "Alice"

`Setters `: To set the value of an attribute.

* use: They add validation logic around setting a value. By using setters, you can enforce rules or constraints before updating an attribute.

class BankAccount:

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

self.\_balance = balance

def set\_balance(self, new\_balance):

if new\_balance >= 0:

self.\_balance = new\_balance

else:

print("Invalid balance. Cannot set a negative value.")

account = BankAccount(1000)

account.set\_balance(1500) # Valid update

account.set\_balance(-200) # Invalid update

 `Deleters`: To delete an instance attribute.

* use: They provide a controlled way to remove an attribute. You can perform additional actions (e.g., logging) before deletion.

class ShoppingCart:

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

self.\_items = []

def add\_item(self, item):

self.\_items.append(item)

def del\_item(self, item):

if item in self.\_items:

self.\_items.remove(item)

else:

print(f"{item} not found in the cart.")

cart = ShoppingCart()

cart.add\_item("Shoes")

cart.add\_item("Books")

cart.del\_item("Books") # Removes "Books" from the cart

cart.del\_item("Hat")

**11.How does polymorphism contribute to code reusability in OOP? Discuss with an example.**

**A.**  **OOPS polymorphism** allows using a single interface to represent a general action across multiple classes. This means common functionality can be defined in a superclass, and subclasses can override these methods to provide their specialized implementations. This reusability reduces redundant code and makes maintenance easier.

class India:

    def capital(self):

        print("New Delhi is the capital of India.")

    def language(self):

        print("Hindi is the most widely spoken language of India.")

    def type(self):

        print("India is a developing country.")

class USA:

    def capital(self):

        print("Washington, D.C. is the capital of USA.")

    def language(self):

        print("English is the primary language of USA.")

    def type(self):

        print("USA is a developed country.")

obj\_ind = India()

obj\_usa = USA()

for country in (obj\_ind, obj\_usa):

    country.capital()

    country.language()

    country.type()

**12.Discuss a real-world scenario where encapsulation in OOP is beneficial for data protection.**

**A.** Consider a banking application written in Python. In this scenario, encapsulation in Object-Oriented Programming (OOP) is crucial for data protection.

Let's say we have a BankAccount class with attributes like account\_number, balance, and owner\_name. Encapsulation allows us to hide the internal state of the BankAccount object and only expose certain methods to interact with it.

By encapsulating the attributes of the BankAccount class and providing access to them through methods like deposit(), withdraw(), and get\_balance(), we can ensure that the data (such as the account balance) is accessed and modified in a controlled manner. This prevents direct manipulation of sensitive data from outside the class, thus enhancing data protection.

For example, the deposit() method might include logic to update the balance only if the deposit amount is positive, while the withdraw() method could check if the withdrawal amount is not greater than the current balance. These encapsulated methods ensure that the data integrity is maintained and prevent unauthorized or erroneous access to sensitive information.

**13.Explain the concept of inheritance and its advantages in the context of software maintenance.**

**A.** Inheritance is a fundamental concept in Object-Oriented Programming (OOP) that allows a new class (called a "subclass" or "child class") to inherit attributes and methods from an existing class (called a "superclass" or "parent class"). This means that the subclass can reuse and extend the functionality of the superclass.

In the context of software maintenance, inheritance offers several advantages:

1. **Code Reusability**: Inheritance promotes code reuse by allowing subclasses to inherit behavior and attributes from a superclass. This reduces redundancy and promotes a modular approach to programming. When a change is required, it often only needs to be made in one place, the superclass, rather than in multiple subclasses.
2. **Modularity and Organization**: Inheritance helps in organizing and structuring code by creating a hierarchy of classes based on their relationships. This makes the codebase easier to understand, navigate, and maintain. Developers can easily identify common functionalities and group them into superclasses, while specific behaviors can be implemented in subclasses.
3. **Ease of Maintenance**: Inheritance facilitates software maintenance by allowing changes to be made in a centralized manner. When a modification is needed, such as adding new features or fixing bugs, it can often be done in the superclass, affecting all subclasses that inherit from it. This reduces the risk of introducing errors and ensures consistency across the codebase.
4. **Enhanced Extensibility**: Inheritance enables the extension of existing functionalities without modifying the original code. Subclasses can add new methods or override existing ones to tailor the behavior according to specific requirements, while still inheriting the common functionalities from the superclass. This promotes flexibility and scalability in software development.
5. **Polymorphism**: Inheritance facilitates polymorphism, which allows objects of different classes to be treated uniformly through a common interface. This simplifies code implementation and maintenance by allowing interchangeable use of objects based on their common superclass, without the need for complex conditional logic.

**14.What is the purpose of using decorators in Python? Explain with an example.**

**A** In Python, decorators are functions that modify or enhance the behavior of other functions or methods. They provide a convenient way to add functionality to existing code without modifying it. Decorators are often used for tasks like logging, authorization, caching, and performance monitoring.

Here's a simple example to illustrate the purpose of decorators:

def logger(func):

def wrapper(\*args, \*\*kwargs):

print(f"Calling function: {func.\_\_name\_\_}")

result = func(\*args, \*\*kwargs)

print(f"Function {func.\_\_name\_\_} executed")

return result

return wrapper

@logger

def add(a, b):

return a + b

result = add(2, 3)

print("Result:", result)

In this example, we define a decorator function logger that takes another function func as an argument. Inside the logger function, we define a nested function wrapper that prints a message before and after calling the original function func. Then, we return this wrapper function.

**15.Illustrate the usage of static method special (magic/dunder) methods in Python with an example.**

**A.** Static method special (magic/dunder) methods in Python are used to define behaviors for built-in operations on objects of a class. One such special method is \_\_str\_\_, which is used to return a string representation of an object. Here's an example:

class Point:

def \_\_init\_\_(self, x, y):

self.x = x self.y = y

@staticmethod

def distance(p1, p2):

return ((p1.x - p2.x) \*\* 2 + (p1.y - p2.y) \*\* 2) \*\* 0.5

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

return f"Point({self.x}, {self.y})"

# Creating two Point objects

point1 = Point(1, 2)

point2 = Point(4, 6)

# Calling the static method to calculate distance between points distance = Point.distance(point1, point2)

print(f"Distance between {point1} and {point2} is {distance}.")

In this example, we have a Point class with \_\_str\_\_ special method defined to return a string representation of a Point object. Additionally, we have a static method distance defined to calculate the distance between two points. The @staticmethod decorator indicates that the method is static and does not require access to instance attributes.

When we call Point.distance(point1, point2), it calculates the distance between the two points using the static method distance. Finally, we print the result using the \_\_str\_\_ special method to display the points in a readable format.