**OOPS Assignment 2**

**Constructor in Python**

1. **What is a constructor in Python?**  
   A constructor is a special method in Python used to initialize objects when they are created. It is defined using the \_\_init\_\_() method in a class.
2. **Differentiate between a parameterless constructor and a parameterized constructor in Python.**
   * **Parameterless Constructor:** A constructor that does not take any parameters except self.
   * **Parameterized Constructor:** A constructor that takes arguments to initialize instance variables.
3. **How do you define a constructor in a Python class? Provide an example.**

class Example:

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

self.value = value # Constructor initializes an attribute

obj = Example(10)

print(obj.value) # Output: 10

1. **Explain the \_\_init\_\_ method in Python and its role in constructors.**  
   The \_\_init\_\_() method is automatically called when an object is created. It initializes object attributes.
2. **In a class named Person, create a constructor that initializes the name and age attributes. Provide an example of creating an object of this class.**

class Person:

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

self.name = name

self.age = age

p = Person("Alice", 25)

print(p.name, p.age) # Output: Alice 25

1. **How can you call a constructor explicitly in Python? Give an example.**

p = Person.\_\_init\_\_(Person, "Bob", 30) # Explicit call

1. **What is the significance of the self parameter in Python constructors? Explain with an example.**  
   The self parameter represents the instance of the class and allows access to instance attributes.

class Demo:

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

self.x = x # self refers to instance attribute

1. **Discuss the concept of default constructors in Python. When are they used?**  
   If no constructor is explicitly defined, Python provides a default constructor that initializes an empty object.
2. **Create a Python class called Rectangle with a constructor that initializes the width and height attributes. Provide a method to calculate the area of the rectangle.**

class Rectangle:

def \_\_init\_\_(self, width, height):

self.width = width

self.height = height

def area(self):

return self.width \* self.height

r = Rectangle(5, 10)

print(r.area()) # Output: 50

1. **How can you have multiple constructors in a Python class? Explain with an example.**  
   Python does not support multiple constructors directly, but we can use default arguments or class methods.

class Demo:

def \_\_init\_\_(self, x=None):

self.x = x if x else 0

**Inheritance in Python**

1. **What is inheritance in Python? Explain its significance in object-oriented programming.**  
   Inheritance allows a child class to inherit attributes and methods from a parent class, promoting code reusability.
2. **Differentiate between single inheritance and multiple inheritance in Python. Provide examples for each.**
   * **Single Inheritance:** One class inherits from another.
   * **Multiple Inheritance:** A class inherits from more than one base class.

class Parent:

pass

class Child(Parent): # Single Inheritance

pass

class Parent1:

pass

class Parent2:

pass

class Child(Parent1, Parent2): # Multiple Inheritance

pass

1. **Create a Python class called Vehicle with attributes color and speed. Then, create a child class called Car that inherits from Vehicle and adds a brand attribute.**

class Vehicle:

def \_\_init\_\_(self, color, speed):

self.color = color

self.speed = speed

class Car(Vehicle):

def \_\_init\_\_(self, color, speed, brand):

super().\_\_init\_\_(color, speed)

self.brand = brand

1. **Explain the concept of method overriding in inheritance. Provide a practical example.**  
   Method overriding allows a subclass to redefine a method from its parent class.

class Parent:

def show(self):

print("Parent class")

class Child(Parent):

def show(self):

print("Child class") # Overrides Parent's show()

1. **How can you access the methods and attributes of a parent class from a child class in Python?**  
   Using super().

class Parent:

def display(self):

print("Parent method")

class Child(Parent):

def call\_parent(self):

super().display() # Calls Parent's method

1. **Discuss the use of the super() function in Python inheritance. When and why is it used?**  
   super() allows calling a parent class method without explicitly naming the parent class.
2. **Create a Python class called Animal with a method speak(). Then, create child classes Dog and Cat that override speak().**

class Animal:

def speak(self):

pass

class Dog(Animal):

def speak(self):

print("Bark")

class Cat(Animal):

def speak(self):

print("Meow")

1. **Explain the role of the isinstance() function in Python and how it relates to inheritance.**  
   isinstance(obj, Class) checks if an object belongs to a class or its subclass.
2. **What is the purpose of the issubclass() function in Python? Provide an example.**  
   issubclass(SubClass, ParentClass) checks if a class is derived from another class.

print(issubclass(Dog, Animal)) # Output: True

1. **Discuss the concept of constructor inheritance in Python. How are constructors inherited in child classes?**  
   If a child class does not define its own constructor, it automatically inherits the parent’s constructor.

class Parent:

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

print("Parent Constructor")

class Child(Parent):

pass # Inherits Parent's constructor

c = Child() # Output: Parent Constructor

**Encapsulation in Python**

1. **What is encapsulation in Python? Explain its significance.**  
   Encapsulation is the bundling of data and methods that operate on that data within a single unit (a class) while restricting direct access to some details. It helps in data hiding and maintaining integrity.
2. **How can you implement encapsulation in Python? Provide an example.**  
   Using private and protected attributes:

class Example:

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

self.\_protected = "Protected"

self.\_\_private = "Private"

obj = Example()

print(obj.\_protected) # Can be accessed but conventionally should not

print(obj.\_\_private) # Error: Cannot be accessed directly

1. **Differentiate between public, protected, and private access modifiers in Python.**
   * **Public (variable)**: Accessible from anywhere.
   * **Protected (\_variable)**: Should not be accessed outside the class but can be.
   * **Private (\_\_variable)**: Cannot be accessed directly outside the class.
2. **How do you access private attributes in Python? Provide an example.**  
   Using name mangling (\_ClassName\_\_attribute):

class Demo:

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

self.\_\_secret = "Hidden"

obj = Demo()

print(obj.\_Demo\_\_secret) # Accessing private attribute

1. **Why is encapsulation important in object-oriented programming?**
   * Protects data integrity
   * Prevents unintended modifications
   * Encourages modularity and abstraction
2. **What is the difference between encapsulation and abstraction in Python?**
   * **Encapsulation**: Hides data within a class to restrict direct access.
   * **Abstraction**: Hides implementation details and exposes only the necessary functionality.
3. **How can encapsulation help in preventing accidental data modification?**  
   By using private attributes and providing getter/setter methods to control access.
4. **Create a class BankAccount with encapsulated attributes for balance. Provide methods to deposit and withdraw money securely.**

class BankAccount:

def \_\_init\_\_(self, balance=0):

self.\_\_balance = balance

def deposit(self, amount):

if amount > 0:

self.\_\_balance += amount

def withdraw(self, amount):

if 0 < amount <= self.\_\_balance:

self.\_\_balance -= amount

def get\_balance(self):

return self.\_\_balance

1. **What are getter and setter methods in Python? Why are they used?**  
   Getters retrieve values, and setters modify values while maintaining data integrity.
2. **Provide an example of using getter and setter methods in a class.**

class Student:

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

self.\_\_name = name

def get\_name(self):

return self.\_\_name

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

if isinstance(new\_name, str):

self.\_\_name = new\_name

**Polymorphism in Python**

1. **What is polymorphism in Python? Explain with an example.**  
   Polymorphism allows different objects to use the same interface. Example:

class Cat:

def sound(self):

return "Meow"

class Dog:

def sound(self):

return "Bark"

for animal in (Cat(), Dog()):

print(animal.sound()) # Same method, different behavior

1. **Differentiate between compile-time and run-time polymorphism.**
   * **Compile-time Polymorphism:** Python does not support method overloading.
   * **Run-time Polymorphism:** Achieved using method overriding.
2. **Explain method overloading in Python and provide an example.**  
   Python does not support true method overloading, but we can use default arguments:

class Example:

def display(self, msg="Hello"):

print(msg)

1. **What is method overriding in Python? Provide an example.**  
   A child class redefines a method from its parent class.

class Parent:

def show(self):

print("Parent")

class Child(Parent):

def show(self):

print("Child") # Overrides parent's method

1. **How does Python achieve operator overloading? Provide an example.**  
   By defining special methods like \_\_add\_\_():

class Vector:

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

self.x = x

def \_\_add\_\_(self, other):

return Vector(self.x + other.x)

1. **What is duck typing in Python? Provide an example.**  
   If an object behaves like a certain type, it is treated as such:

class Bird:

def fly(self):

print("Flying")

class Plane:

def fly(self):

print("Gliding")

def takeoff(obj):

obj.fly()

takeoff(Bird()) # Works because both have a fly() method

1. **Explain function polymorphism in Python with an example.**  
   The same function can work with different types.

print(len("hello")) # Works on strings

print(len([1, 2, 3])) # Works on lists

1. **What are magic methods in Python? Provide an example.**  
   Special methods prefixed with \_\_, like \_\_str\_\_(), \_\_add\_\_().

class Example:

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

return "This is an example class"

1. **How does polymorphism improve code reusability?**  
   It allows a common interface for multiple classes, reducing redundant code.
2. **Provide an example of polymorphism using an abstract class in Python.**

from abc import ABC, abstractmethod

class Animal(ABC):

@abstractmethod

def speak(self):

pass

class Dog(Animal):

def speak(self):

return "Bark"

**Abstraction in Python**

1. **What is abstraction in Python?**  
   Abstraction hides implementation details and exposes only essential features.
2. **How do you implement abstraction in Python?**  
   Using abstract classes with the abc module.
3. **What is an abstract class in Python? Provide an example.**  
   A class with at least one abstract method.

from abc import ABC, abstractmethod

class Animal(ABC):

@abstractmethod

def sound(self):

pass

1. **Why is abstraction important in object-oriented programming?**  
   It helps in hiding unnecessary details, making code easier to use and maintain.
2. **How does an abstract method differ from a regular method in Python?**  
   Abstract methods have no implementation in the base class and must be implemented in derived classes.
3. **What happens if a class contains an abstract method but does not implement it?**  
   The class becomes abstract and cannot be instantiated.
4. **Can an abstract class have concrete (normal) methods in Python?**  
   Yes, abstract classes can have both abstract and concrete methods.
5. **Explain the difference between abstraction and encapsulation.**
   * **Encapsulation:** Hides data.
   * **Abstraction:** Hides implementation details.
6. **Provide an example where abstraction is useful in real-world applications.**
   * An ATM class hides implementation but exposes withdraw() and deposit().
7. **Can we instantiate an abstract class in Python? Why or why not?**  
   No, because abstract classes are meant to be inherited and must have their abstract methods implemented.