PYTHON

OOPS:-

OOPs (Object-Oriented Programming System) is a programming paradigm based on the concept of **objects** and **classes**, which help organize code for better reusability, scalability, and maintainability.

SECTION 1: BASIC CLASS DEFINITION AND OBJECTS:-

1. **Class and Objects**: A class is a blueprint for creating objects, defining attributes (data) and methods (behavior). Objects are instances of a class.
2. **Instance vs. Class Attributes**: Instance attributes (e.g., name, age) are unique to each object, while class attributes (e.g., species) are shared by all instances.
3. **__init__ Method**: This is the constructor method in Python, automatically called when an object is created to initialize instance attributes.
4. **Methods in a Class**: Instance methods (e.g., bark, get_info) define behaviors that objects can perform and can access instance attributes via self.
5. **Creating and Using Objects**: Objects are instantiated using the class name, and attributes/methods are accessed using dot notation (e.g., fido.name, fido.bark()).

SECTION 2: INHERITANCE

1. **Inheritance Concept**: A subclass inherits attributes and methods from a parent class, allowing code reuse and hierarchy creation.
2. **super() Function**: Calls the parent class's __init__ method to initialize inherited attributes in the child class.
3. **Method Overriding**: A subclass can override a parent class's method (e.g., speak() in Cat overrides speak() in Pet).
4. **Instance Checking**: isinstance(object, Class) checks if an object belongs to a specific class or its parent class.

5. **Subclass-Specific Methods**: A subclass can define its own unique attributes and methods (e.g., purr() in Cat).

SECTION 3: ENCAPSULATION

1. **Encapsulation Concept**: It restricts direct access to an object's data, ensuring better control and security.
2. **Private Attributes (__var)**: Double underscore (__balance) makes attributes harder to access directly, enforcing data protection.
3. **Protected Attributes (_var)**: Single underscore (_transaction_count) signals that the attribute should not be modified outside the class, but it's still accessible.
4. **Public Methods**: Methods like deposit(), withdraw(), and get_balance() provide controlled access to private data.
5. **Name Mangling**: Private attributes can still be accessed using _ClassName__attribute, but this is discouraged in practice.

SECTION 4: POLYMORPHISM

1. **Polymorphism Concept**: It allows different classes to use the same interface (e.g., speak() method in multiple subclasses).
2. **Method Overriding**: Subclasses (Dog, Cat, Duck) override the speak() method from the parent Animal class to provide specific implementations.
3. **Abstract Method**: The speak() method in Animal is defined but raises NotImplementedError, forcing subclasses to implement it.
4. **Function Polymorphism**: The animal_sound() function works with different objects (Dog, Cat, Duck) without modifying its implementation.
5. **Class-Level Polymorphism**: The introduce() method in Animal uses speak(), enabling each subclass to provide its own behavior dynamically.

SECTION 5: ABSTRACTION

1. **Abstraction Concept**: Hides implementation details and provides a blueprint for derived classes.
2. **Abstract Base Class (ABC)**: Used to define an abstract class that cannot be instantiated directly.

3. **Abstract Methods (@abstractmethod)**: Must be implemented by subclasses (Circle, Rectangle) to ensure a consistent interface.
4. **Concrete Methods in Abstract Classes**: describe() is implemented in Shape but relies on subclass implementations of area() and perimeter().
5. **Enforcement of Implementation**: Attempting to instantiate Shape directly results in an error, ensuring all required methods are defined in subclasses.

@abstractmethod Decorator

The @abstractmethod decorator in Python is used to define **abstract methods** inside an **abstract base class (ABC)**.

Key Points:

1. **Enforces Implementation**: Any subclass **must** implement the method marked with @abstractmethod, or it cannot be instantiated.
2. **Defines Interface**: Ensures a consistent interface across multiple subclasses.
3. **Part of ABC Module**: Requires importing ABC from abc (Abstract Base Class module).
4. **Prevents Instantiation**: A class with at least one @abstractmethod **cannot** be instantiated directly.