**Python Programming Course: Object-Oriented Programming (OOP)**

**Module 1: Introduction to OOP**

**1.1 Introduction to OOPs**

* **What is OOP?**
  + OOP is a programming paradigm that uses "objects" to design applications and computer programs.
  + The core idea is to model real-world entities as software objects, which have both data (attributes) and behavior (methods).
  + This approach helps in managing complexity, improving code reusability, and making programs more organized.
* **Key Principles of OOP:** We will introduce the four pillars of OOP:
  1. **Encapsulation:** Bundling data and methods together.
  2. **Inheritance:** Creating new classes from existing ones.
  3. **Polymorphism:** A single interface for different types of objects.
  4. **Abstraction:** Hiding implementation details.

**Module 2: The Building Blocks - Classes and Objects**

**2.1 Classes and Objects**

* **Classes:** A class is a blueprint or a template for creating objects. It defines the attributes and methods that all objects of that class will have.
  + **Example:** A Car class is a blueprint. It might define attributes like color and speed, and methods like accelerate() and brake().
* **Objects:** An object is an instance of a class. When you create an object, you are creating a specific, tangible item based on the class blueprint.
  + **Example:** my\_car = Car('red', 60) creates a specific car object named my\_car, which is red and has a speed of 60.

**Code Example:**

# A simple Class definition

class Dog:

# A class attribute, shared by all instances

species = "Canis familiaris"

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

# The \_\_init\_\_ method is a special method called when a new object is created.

# It initializes the object's instance attributes.

self.name = name # Instance attribute

self.age = age # Instance attribute

def bark(self):

# An instance method that operates on the object's data

print(f"{self.name} says woof!")

# Creating objects (instances) of the Dog class

dog1 = Dog("Buddy", 5)

dog2 = Dog("Lucy", 3)

# Accessing object attributes and methods

print(f"{dog1.name} is a {dog1.species} and is {dog1.age} years old.")

dog1.bark()

print(f"{dog2.name} is also a {Dog.species}.")

**Module 3: Data Hiding and Access - Encapsulation and Access Modifiers**

**3.1 Encapsulation**

* **Definition:** The principle of bundling data and the methods that operate on that data into a single unit (the class). It's about protecting the data from outside interference.
* **Data Hiding:** Python doesn't have strict private keywords like other languages. Encapsulation is achieved by convention.
  + **Public members:** Standard attributes and methods (e.g., self.name). Accessible from anywhere.
  + **Protected members:** Conventionally, these start with a single underscore (e.g., \_protected\_member). They should not be accessed directly from outside the class, but are accessible by subclasses.
  + **Private members:** Conventionally, these start with a double underscore (e.g., \_\_private\_member). Python's name mangling makes it harder (though not impossible) to access them from outside the class.

**Code Example:**

class BankAccount:

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

self.owner = owner # Public attribute

self.\_\_balance = balance # Private attribute (by convention)

def deposit(self, amount):

if amount > 0:

self.\_\_balance += amount

print(f"Deposited ${amount}. New balance: ${self.\_\_balance}")

else:

print("Deposit amount must be positive.")

def get\_balance(self):

# A public method to safely access the private balance

return self.\_\_balance

# Creating an object

account = BankAccount("Alice", 1000)

# Accessing public attribute

print(f"Account owner: {account.owner}")

# Attempting to access the "private" attribute will cause an AttributeError

try:

print(account.\_\_balance)

except AttributeError as e:

print(f"Error: {e}")

# Accessing the private data through a public method

print(f"Current balance: ${account.get\_balance()}")

account.deposit(500)

**Module 4: Advanced Class Features**

**4.1 Decorators**

* **Definition:** A decorator is a function that takes another function as an argument, adds some functionality, and returns a new function. They are often used with @ syntax to modify the behavior of a function or class method.
* **Property Decorator (@property):** This is a key decorator in OOP. It allows you to define a method that can be accessed like an attribute. This is great for creating "getters" and "setters" to control access to instance attributes.

**Code Example:**

class Celsius:

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

self.\_temperature = temperature

@property

def temperature(self):

print("Getting value...")

return self.\_temperature

@temperature.setter

def temperature(self, value):

if value < -273.15:

raise ValueError("Temperature below absolute zero is not possible.")

print("Setting value...")

self.\_temperature = value

# Creating an object

c = Celsius(25)

# Accessing the "temperature" attribute, which calls the @property getter

print(c.temperature)

# Setting the "temperature" attribute, which calls the @temperature.setter

c.temperature = 30

print(c.temperature)

# This will raise a ValueError

try:

c.temperature = -300

except ValueError as e:

print(e)

**4.2 Class Methods and Static Members**

* **Class Methods:**
  + Defined with the @classmethod decorator.
  + The first argument is conventionally cls (for class) instead of self.
  + They can access and modify class attributes, but not instance attributes. They are often used as alternative constructors.
* **Static Methods:**
  + Defined with the @staticmethod decorator.
  + They don't receive self or cls as their first argument.
  + They are utility functions that belong to a class but don't operate on any specific instance or the class itself.

**Code Example:**

Python

class MyCar:

num\_wheels = 4 # A class attribute

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

self.color = color

@classmethod

def get\_num\_wheels(cls):

# A class method to access a class attribute

return cls.num\_wheels

@staticmethod

def honk():

# A static method that doesn't need a specific car instance

print("Honk honk!")

# Accessing the class method using the class name

print(f"All cars have {MyCar.get\_num\_wheels()} wheels.")

# Accessing the static method using the class name

MyCar.honk()

**Module 5: Code Reusability - Inheritance**

**5.1 Inheritance**

* **Definition:** The process of creating a new class (a "child" or "subclass") from an existing class (a "parent" or "superclass"). The child class inherits all the attributes and methods of the parent.
* **Purpose:** Promotes code reusability and establishes a "is-a" relationship (e.g., a Dog "is a" Animal).

**5.2 Types of Inheritance**

* **Single Inheritance:** A class inherits from only one parent class. (This is the most common type.)
* **Multi-level Inheritance:** A child class inherits from a parent, and that parent inherits from another parent. (e.g., Grandparent -> Parent -> Child)
* **Hierarchical Inheritance:** Multiple child classes inherit from a single parent class.
* **Hybrid Inheritance:** A combination of two or more types of inheritance.

**5.3 Multiple Inheritance**

* **Definition:** A class inherits from multiple parent classes.
* **Method Resolution Order (MRO):** Python uses a C3 linearization algorithm to determine the order in which base classes are searched for a method. This is important to understand when dealing with multiple inheritance. The super() function helps with calling methods from the correct parent in the MRO.

**Code Example:**

# Parent class

class Animal:

def speak(self):

raise NotImplementedError("Subclass must implement abstract method")

# Child class inheriting from Animal (Single Inheritance)

class Dog(Animal):

def speak(self):

return "Woof!"

# Multiple Inheritance Example

class Runner:

def run(self):

return "Running..."

class Swimmer:

def swim(self):

return "Swimming..."

class Athlete(Runner, Swimmer):

pass

athlete = Athlete()

print(f"Athlete is {athlete.run()} and {athlete.swim()}.")

**Module 6: Flexibility and Abstraction - Polymorphism**

**6.1 Polymorphism**

* **Definition:** The ability of an object to take on many forms. It allows you to use a single interface for different types of objects.
* **Duck Typing:** A key concept in Python polymorphism. "If it walks like a duck and it quacks like a duck, then it must be a duck." In Python, we care about what an object *can do* (its methods), not what its specific type is.

**6.2 More Examples of Polymorphism**

* **Function Polymorphism:** A single function can work with different types of objects, as long as they implement the required methods.
* **Method Overriding:** A subclass can provide its own implementation of a method that is already defined in its parent class.

**Code Example:**

class Cat:

def speak(self):

return "Meow!"

class Duck:

def speak(self):

return "Quack!"

def animal\_sound(animal):

# This function is polymorphic because it works with any object that has a 'speak' method.

print(animal.speak())

animal\_sound(Cat())

animal\_sound(Duck())

**Module 7: Advanced Polymorphism and Interfaces**

**7.1 Operator Overloading**

* **Definition:** The ability to change the way an operator (like +, -, \*) works for a custom object. This is achieved by defining special methods like \_\_add\_\_, \_\_sub\_\_, etc.
* **Example:** Making the + operator work to add two custom Vector objects.

**Code Example:**

Python

class Vector:

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

self.x = x

self.y = y

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

# Defines the behavior of the '+' operator

return Vector(self.x + other.x, self.y + other.y)

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

# Defines the string representation of the object

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

v1 = Vector(2, 3)

v2 = Vector(5, 7)

v3 = v1 + v2 # Calls v1.\_\_add\_\_(v2)

print(v3)

**7.2 Abstract Class**

* **Definition:** A class that cannot be instantiated on its own. It is meant to be a blueprint for other classes, forcing them to implement certain methods.
* **Purpose:** Enforces a contract on all subclasses, ensuring they have a common interface.
* **Python's abc Module:** Python uses the abc (Abstract Base Classes) module to create abstract classes. The @abstractmethod decorator marks a method that must be implemented by any concrete subclass.

**Code Example:**

Python

from abc import ABC, abstractmethod

class Shape(ABC):

@abstractmethod

def area(self):

pass

@abstractmethod

def perimeter(self):

pass

# This will raise a TypeError because you can't instantiate an abstract class

try:

s = Shape()

except TypeError as e:

print(e)

# A concrete class that implements the abstract methods

class Rectangle(Shape):

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

self.length = length

self.width = width

def area(self):

return self.length \* self.width

def perimeter(self):

return 2 \* (self.length + self.width)

rectangle = Rectangle(5, 4)

print(f"Area: {rectangle.area()}")

print(f"Perimeter: {rectangle.perimeter()}")