**EXPLANATION OF THE OOPS CONCEPT APPLIED IN CODE**

Here's a breakdown of how I have applied oops concept in code:

**1. Creating Classes**

* In the program, there are three main classes: Task, PriorityTask, and TaskList.
* **Task class** : Represents a single task with attributes like title, description, and status.
* **PriorityTask Class**:It is a subclass of Task, which extends Task by adding a priority attribute (e.g., low, medium, high).
* **TaskList class** : It has a collection of tasks and provides methods to add, remove, list, and search tasks.

**Why using class matters**: according to my point of view Classes help to organize the code into reusable, logical units ( called as objects). Each task is an instance of the Task class, and tasks with priorities are instances of PriorityTask.

**2. Encapsulation**

* Encapsulation refers to the bindling of data ( ie attributes) and methods that operate on that data into a single unit (class). It also helps restrict access to certain parts of the object, usually by making attributes private or by providing public methods for access.

**Example from program**: In the Task class, the attributes (title, description, status) are encapsulated inside the class. I have provided public methods like mark\_complete() and \_\_str\_\_() to interact with the data, but the data Cannot be directly accessed outside the class. The status can only be changed through the mark\_complete() method, enforcing control over how tasks are updated.

**Why it matters**: Encapsulation keeps the code cleaner and safer. Aa in program it don’t allow anyone to change the status of a task directly; they have to use a method (mark\_complete()), which controls how and when the status is updated.

**3. Method Overloading**

* Method overloading means that the same method or name can be used to perform different actions depending on the arguments provided. In Python, overloading is typically done by using default arguments or by checking the number of arguments.

**Example**: The add\_task() method in the TaskList class demonstrates overloading. If no description is given, it creates a task with just a title. If a description is provided, the method creates a task with both a title and a description.

**CODE**

def add\_task(self, title, description=""):

if description:

task = Task(title, description)

else:

task = Task(title)

**Why it matters**: This makes the method flexible and user friendly The user can add a task with or without a description, depending on what they provide when calling the method.

**4. Method Overriding**

* Method overriding occurs when a subclass Defines its own implementation or a new implementation of a method that is already defined in its superclass or parent class.

**For Example**: In the PriorityTask class, I have override the \_\_str\_\_() method. The Task class has a \_\_str\_\_() method that returns a basic string representation, but the PriorityTask class provides or defines a more detailed string that includes the task’s priority.

**CODE**

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

return f"Priority Task: {self.title} - Status: {self.status} - Priority: {self.priority}"

**Why using overriding matters**: By overriding \_\_str\_\_(), I have customized the string representation of priority tasks. This way, when we print a priority task, it includes the priority level, which is specific to that particular class.

**5. Inheritance**

* Inheritance allows one class (child class) to inherit attributes and methods from another class (parent class). It’s a effective way to reuse code and create a hierarchy of classes.

**Example**: The PriorityTask class inherits from the Task class. This means PriorityTask has all the attributes and methods of Task (like title, description, and status), but it also added its own attribute, priority.

**CODE**

class PriorityTask(Task):

def \_\_init\_\_(self, title, description="", priority="medium"):

super().\_\_init\_\_(title, description)

self.priority = priority

**Why using inheritance matters**: Inheritance allows us to create specialized versions of a class (like PriorityTask) without rewriting all the basic task functionality again . We can reuse the code and We only add something new what’s specific to the child class.

**6. Polymorphism**

* Polymorphism is the ability to use methods or objects of different classes in the same way. The idea is that a method can act on objects of different types (classes), and the correct method is called based on the object’s type.

**Example**: The TaskList class can handle both Task and PriorityTask objects in a similar way. When you list all tasks, both Task and PriorityTask are handled by the \_\_str\_\_() method of their respective classes, but the list\_tasks() method in TaskList treats them the same way by calling the \_\_str\_\_() method for each task.

**CODE**

def list\_tasks(self):

if not self.tasks:

print("No tasks available.")

else:

print("Tasks:")

for task in self.tasks:

print(task) # Polymorphism in action

**Why it matters**: Polymorphism allows us to treat objects from different classes (like Task and PriorityTask) uniformly, making the code easier to extend and more flexible.

**A short Summary of OOPs Concepts in the Code:**

* **Classes**: I have created classes like Task, PriorityTask, and TaskList to represent objects and their behaviors.
* **Encapsulation**: The task details are encapsulated inside the class, and methods are used to interact with them.
* **Method Overloading**: add\_task() is overloaded to accept tasks with or without descriptions.
* **Method Overriding**: The \_\_str\_\_() method is overridden in PriorityTask to display the priority of the task.
* **Inheritance**: PriorityTask inherits from Task, inheriting all the properties and methods of Task.
* **Polymorphism**: The same list\_tasks() method can handle both Task and PriorityTask objects, showcasing polymorphism in action.

This is how I applied the core OOP concepts in my code while making it modular, reusable, and easy to extend.