Use Case: Online Learning Management System (LMS)

Concept: Demonstrate OOP principles in a training simulation for building an LMS system with Users like Students, Instructors, and Admins.

Task 1: Class and Object Creation

Scenario:

Create a base class `User` to store general information of LMS users.

Objective:

Use `class` to define a structure and create `objects` representing users.

Sample Input:

```
User(name="Alice", email="alice@lms.com", role="Student")
User(name="Bob", email="bob@lms.com", role="Instructor")
User(name="Carol", email="carol@lms.com", role="Admin")
```

Expected Output:

```
User created: Alice (Student)
User created: Bob (Instructor)
User created: Carol (Admin)
```

◇ Task 2: Inheritance & Method Overriding

Scenario:

Instructors can create courses, and students can enroll.

Objective:

Use `inheritance` to create `Student` and `Instructor` classes from `User`. Add role-specific methods.

Sample Input:

```
Bob creates course "Python 101"
Alice enrolls in course "Python 101"
```

Expected Output:

```
Bob has created a course: Python 101
Alice has enrolled in: Python 101
```

◇ Task 3: Class and Object Attributes

Scenario:

Maintain a list of courses each instructor created and each student enrolled in.

Objective:

Use class attributes for global course count and object attributes for user-specific courses.

Sample Input:

```
Total courses created
Show Alice’s enrolled courses
Show Bob’s created courses
```

Expected Output:

```
Total Courses Created: 1
Alice's Enrolled Courses: ['Python 101']
Bob's Created Courses: ['Python 101']
```

◇ Task 4: Polymorphism – Method with Different Behaviors

Scenario:

Each user sees a customized dashboard.

Objective:

Implement `view_dashboard()` differently in `Student`, `Instructor`, and `Admin`.

Sample Input:

```
Alice views dashboard
Bob views dashboard
Carol views dashboard
```

Expected Output:

```
Student Dashboard: View Courses, Submit Assignments
Instructor Dashboard: Manage Courses, View Student Progress
```

```
Admin Dashboard: Site Analytics, User Management
```

Task 5: Dunder Methods – `__str__`, `__eq__`

Scenario:

Print user objects in readable format and compare if two users are the same.

Objective:

Override `__str__` for user display and `__eq__` to compare users.

Sample Input:

```
print(Alice)
print(Bob)
Is Alice == another_Alice (same name/email)?
```

Expected Output:

```
User: Alice (Student), Email: alice@lms.com
User: Bob (Instructor), Email: bob@lms.com
Are users equal? True
```

Task 6: Abstract Class and Abstract Method

Scenario:

Every user type must define their own dashboard method.

🎯 *Objective:*

Use an `abstract class` with an `abstract method` `view_dashboard()`.

📝 *Sample Input:*

```
Try to create instance of abstract class UserBase
Create Student, Instructor and call view_dashboard
```

📤 *Expected Output:*

```
TypeError: Can't instantiate abstract class UserBase with abstract
method view_dashboard
Alice's Dashboard Loaded
Bob's Dashboard Loaded
```

✅ Summary of OOP Concepts Demonstrated

Concept	Demonstrated In
Class	`User`, `Student`, `Instructor`, `Admin`
Object	`Alice`, `Bob`, `Carol`
Inheritance	`Student`, `Instructor` inherit from `User`
Attributes	`name`, `email`, `role`, enrolled/created courses
Polymorphism	`view_dashboard()` different in every subclass
Dunder Methods	`__str__` and `__eq__` in User class
Abstract Class	UserBase with `view_dashboard()` as abstract

Answers:

Online Learning Management System (LMS) including:

✅ Concepts: Class, Objects, Inheritance, Attributes, Polymorphism, Dunder Methods, Abstract Classes
✅ Includes: Real-world **scenario**, **sample input/output**, and **Python code**

💼 Use Case: Online Learning Management System (LMS)

◇ Task 1: Class and Object Creation

Scenario:

Create a base class User to define common attributes for LMS users.

Code:

```
class User:
    def __init__(self, name, email, role):
        self.name = name
        self.email = email
        self.role = role
        print(f"User created: {self.name} ({self.role})")

# Creating objects
student1 = User("Alice", "alice@lms.com", "Student")
instructor1 = User("Bob", "bob@lms.com", "Instructor")
```

```
admin1 = User("Carol", "carol@lms.com", "Admin")
```

Output:

```
User created: Alice (Student)
User created: Bob (Instructor)
User created: Carol (Admin)
```

◇ Task 2: Inheritance & Method Overriding

Scenario:

Instructor can create a course, and Student can enroll in it.

Code:

```
class Instructor(User):
    def __init__(self, name, email):
        super().__init__(name, email, "Instructor")
        self.courses_created = []

    def create_course(self, course_name):
        self.courses_created.append(course_name)
        print(f"{self.name} has created a course: {course_name}")

class Student(User):
    def __init__(self, name, email):
        super().__init__(name, email, "Student")
        self.courses_enrolled = []

    def enroll_course(self, course_name):
        self.courses_enrolled.append(course_name)
        print(f"{self.name} has enrolled in: {course_name}")
```

Input:

```
instructor = Instructor("Bob", "bob@lms.com")
student = Student("Alice", "alice@lms.com")
instructor.create_course("Python 101")
student.enroll_course("Python 101")
```

Output:

```
User created: Bob (Instructor)
User created: Alice (Student)
Bob has created a course: Python 101
Alice has enrolled in: Python 101
```

Task 3: Class and Object Attributes

Scenario:

Track total courses created and user-specific course lists.

Enhancement in Code:

```
class CourseManager:
    total_courses = []

    @classmethod
    def add_course(cls, course_name):
        cls.total_courses.append(course_name)

    @classmethod
    def show_total_courses(cls):
        print("Total Courses Created:", len(cls.total_courses))
```

Input:

```
CourseManager.add_course("Python 101")
CourseManager.show_total_courses()
print("Alice's Enrolled Courses:", student.courses_enrolled)
print("Bob's Created Courses:", instructor.courses_created)
```

Output:

```
Total Courses Created: 1
Alice's Enrolled Courses: ['Python 101']
Bob's Created Courses: ['Python 101']
```

◇ Task 4: Polymorphism – Different view_dashboard() behavior

Code:

```
class Admin(User):
    def __init__(self, name, email):
        super().__init__(name, email, "Admin")

    def view_dashboard(self):
        print("Admin Dashboard: Site Analytics, User Management")

# Polymorphic methods
def view_dashboard(user):
    if isinstance(user, Student):
        print("Student Dashboard: View Courses, Submit Assignments")
    elif isinstance(user, Instructor):
        print("Instructor Dashboard: Manage Courses, View Student
Progress")
    elif isinstance(user, Admin):
        user.view_dashboard()
```

Input:

```
admin = Admin("Carol", "carol@lms.com")
view_dashboard(student)
view_dashboard(instructor)
view_dashboard(admin)
```

Output:

```
User created: Carol (Admin)
Student Dashboard: View Courses, Submit Assignments
Instructor Dashboard: Manage Courses, View Student Progress
Admin Dashboard: Site Analytics, User Management
```

◇ Task 5: Dunder Methods – __str__, __eq__

Code:

```
class User:
    def __init__(self, name, email, role):
        self.name = name
        self.email = email
        self.role = role

    def __str__(self):
        return f"User: {self.name} ({self.role}), Email: {self.email}"

    def __eq__(self, other):
        return self.name == other.name and self.email == other.email

# Example usage
alice1 = User("Alice", "alice@lms.com", "Student")
alice2 = User("Alice", "alice@lms.com", "Student")

print(alice1)
print(instructor1)
print("Are users equal?", alice1 == alice2)
```

Output:

```
User: Alice (Student), Email: alice@lms.com
User: Bob (Instructor), Email: bob@lms.com
Are users equal? True
```

◇ Task 6: Abstract Class & Abstract Method

Code:

```
from abc import ABC, abstractmethod

class UserBase(ABC):
    def __init__(self, name, email):
        self.name = name
        self.email = email

    @abstractmethod
    def view_dashboard(self):
        pass

class StudentUser(UserBase):
    def view_dashboard(self):
        print(f"{self.name}'s Dashboard Loaded: Enrolled Courses")

class InstructorUser(UserBase):
    def view_dashboard(self):
        print(f"{self.name}'s Dashboard Loaded: Created Courses")
```

Input:

```
# Uncommenting this will raise an error:
# base_user = UserBase("Test", "test@lms.com")
```

```
student_user = StudentUser("Alice", "alice@lms.com")
instructor_user = InstructorUser("Bob", "bob@lms.com")

student_user.view_dashboard()
instructor_user.view_dashboard()
```

Output:

```
Alice's Dashboard Loaded: Enrolled Courses
Bob's Dashboard Loaded: Created Courses
```

If we attempt to instantiate the abstract class:

```
base_user = UserBase("Test", "test@lms.com")
```

Output:

```
TypeError: Can't instantiate abstract class UserBase with abstract
method view_dashboard
```

✓ Summary of Concepts

OOP Concept	Covered In
Class & Object	User, Student, Instructor
Inheritance	Student, Instructor inherit from User
Attributes	name, email, role, course lists
Polymorphism	view_dashboard() behavior varies by class
Dunder Methods	__str__, __eq__ for print and compare
Abstract Class	UserBase, enforced view_dashboard()