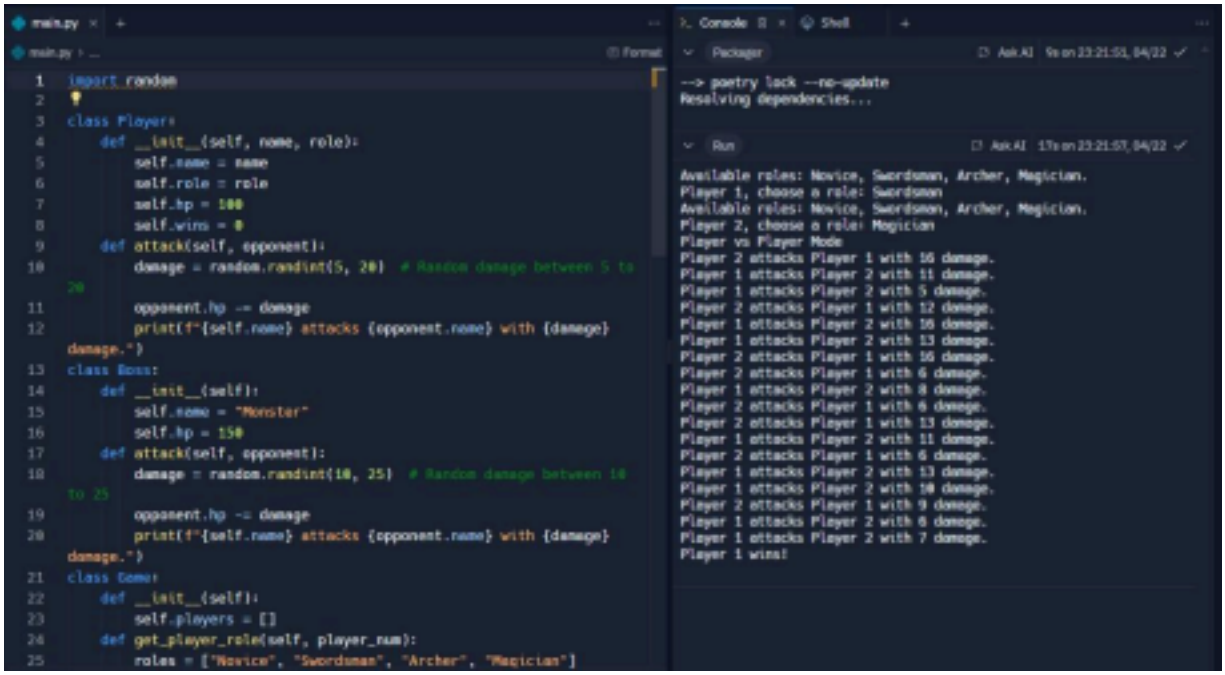
Laboratory Activity No. 2	
Inheritance, Encapsulation, and Abstraction	
Course Code: CPE009	**Program:** Computer Engineering
Course Title: OBJECT-ORIENTED PROGRAMING	**Date Performed: SEPTEMBER 26, 2024**
Section: CPE21S1	**Date Submitted: OCTOBER 3, 2021**
Name(s): GASPAR, AARON ROWEN O.	**Instructor: MA'AM MARIA RIZETTE SAYO**
6. Supplementary Activity	

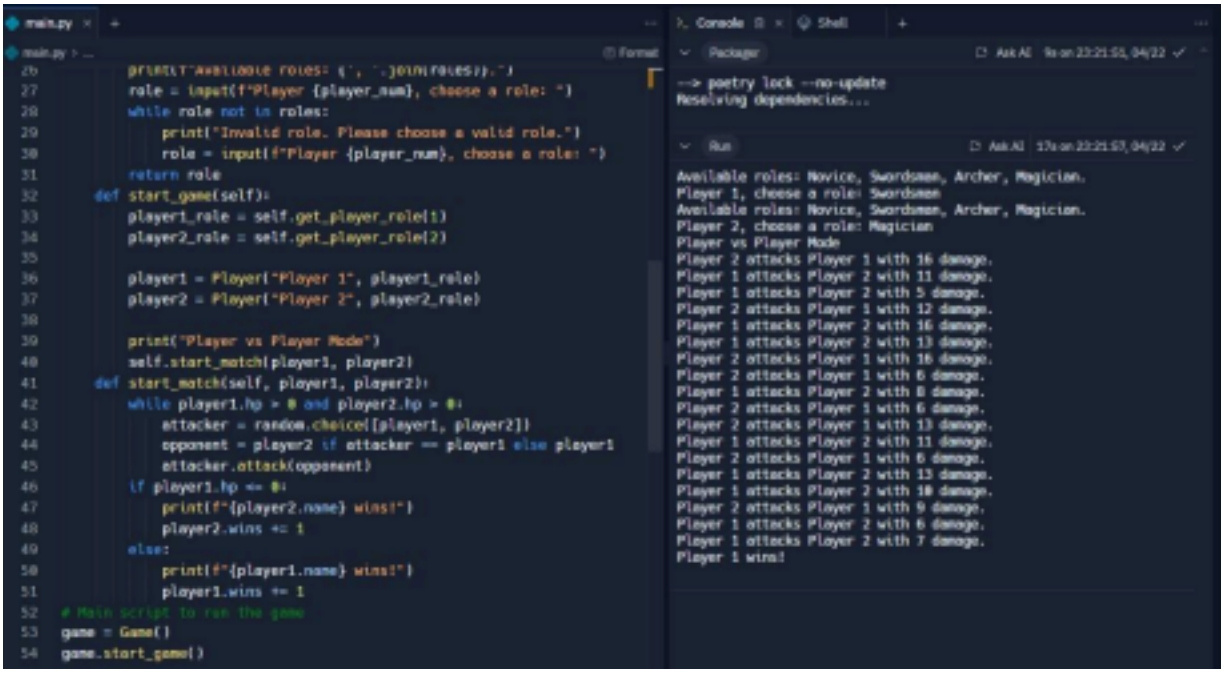


```
import random

class Player:
    def __init__(self, name, role):
        self.name = name
        self.role = role
        self.hp = 100
        self.wins = 0
    def attack(self, opponent):
        damage = random.randint(5, 20)  # Random damage between 5 to 20
        opponent.hp -= damage
        print(f"{self.name} attacks {opponent.name} with {damage} damage.")
class Boss:
    def __init__(self):
        self.name = "Monster"
        self.hp = 150
    def attack(self, opponent):
        damage = random.randint(10, 25)  # Random damage between 10 to 25
        opponent.hp -= damage
        print(f"{self.name} attacks {opponent.name} with {damage} damage.")
class Game:
    def __init__(self):
        self.players = []
    def get_player_role(self, player_num):
        roles = ["Novice", "Swordsman", "Archer", "Magician"]
```

```
--> poetry lock --no-update
Resolving dependencies...

Available roles: Novice, Swordsman, Archer, Magician.
Player 1, choose a role: Swordsman
Available roles: Novice, Swordsman, Archer, Magician.
Player 2, choose a role: Magician
Player vs Player Mode
Player 2 attacks Player 1 with 16 damage.
Player 1 attacks Player 2 with 11 damage.
Player 1 attacks Player 2 with 5 damage.
Player 2 attacks Player 1 with 12 damage.
Player 1 attacks Player 2 with 16 damage.
Player 1 attacks Player 2 with 13 damage.
Player 2 attacks Player 1 with 16 damage.
Player 2 attacks Player 1 with 6 damage.
Player 1 attacks Player 2 with 8 damage.
Player 2 attacks Player 1 with 6 damage.
Player 2 attacks Player 1 with 13 damage.
Player 1 attacks Player 2 with 11 damage.
Player 2 attacks Player 1 with 6 damage.
Player 1 attacks Player 2 with 13 damage.
Player 1 attacks Player 2 with 10 damage.
Player 2 attacks Player 1 with 9 damage.
Player 1 attacks Player 2 with 6 damage.
Player 1 attacks Player 2 with 7 damage.
Player 1 wins!
```



```
        print(f"Available roles: {', '.join(roles)}.")
        role = input(f"Player {player_num}, choose a role: ")
        while role not in roles:
            print("Invalid role. Please choose a valid role.")
            role = input(f"Player {player_num}, choose a role: ")
        return role
    def start_game(self):
        player1_role = self.get_player_role(1)
        player2_role = self.get_player_role(2)

        player1 = Player("Player 1", player1_role)
        player2 = Player("Player 2", player2_role)

        print("Player vs Player Mode")
        self.start_match(player1, player2)
    def start_match(self, player1, player2):
        while player1.hp > 0 and player2.hp > 0:
            attacker = random.choice([player1, player2])
            opponent = player2 if attacker == player1 else player1
            attacker.attack(opponent)
        if player1.hp <= 0:
            print(f"{player2.name} wins!")
            player2.wins += 1
        else:
            print(f"{player1.name} wins!")
            player1.wins += 1
# Main script to run the game
game = Game()
game.start_game()
```

```
--> poetry lock --no-update
Resolving dependencies...

Available roles: Novice, Swordsman, Archer, Magician.
Player 1, choose a role: Swordsman
Available roles: Novice, Swordsman, Archer, Magician.
Player 2, choose a role: Magician
Player vs Player Mode
Player 2 attacks Player 1 with 16 damage.
Player 1 attacks Player 2 with 11 damage.
Player 1 attacks Player 2 with 5 damage.
Player 2 attacks Player 1 with 12 damage.
Player 1 attacks Player 2 with 16 damage.
Player 1 attacks Player 2 with 13 damage.
Player 2 attacks Player 1 with 16 damage.
Player 2 attacks Player 1 with 6 damage.
Player 1 attacks Player 2 with 8 damage.
Player 2 attacks Player 1 with 6 damage.
Player 2 attacks Player 1 with 13 damage.
Player 1 attacks Player 2 with 11 damage.
Player 2 attacks Player 1 with 6 damage.
Player 1 attacks Player 2 with 13 damage.
Player 1 attacks Player 2 with 10 damage.
Player 2 attacks Player 1 with 9 damage.
Player 1 attacks Player 2 with 6 damage.
Player 1 attacks Player 2 with 7 damage.
Player 1 wins!
```

7. Questions

1. Why is Inheritance important?
   - Inheritance is important because it is a key concept in object-oriented programming that helps make code more reusable, easier to maintain, and flexible. By using inheritance, developers can build software that is more efficient, organized, and scalable. Knowing how to use inheritance well is crucial for any programmer who wants to succeed in software development.

2. Explain the advantages and disadvantages of applying inheritance in an Object-Oriented Program.
   - Inheritance also has some downsides, such as creating strong dependencies between classes, potential issues when the base class changes, performance overhead, and added complexity. Knowing these pros and cons helps developers use inheritance effectively, making the most of its advantages while avoiding its pitfalls.

3. Differentiate single inheritance, multiple inheritance, and multi-level inheritance
   - Single inheritance is when a class inherits from just one parent class. Multiple inheritance lets a class inherit from more than one parent class. Multilevel inheritance is when a class inherits from another derived class, forming a chain. Each type of inheritance has its own benefits and challenges. Knowing how these types work and their effects is important for creating and managing object-oriented systems effectively.

4. Why is super(). init (username) added in the codes of Swordsman, Archer, Magician, and Boss?
   -

5. How do you think Encapsulation and Abstraction helps in making good Object-Oriented Programs?
   -

8. Conclusion

Understanding the concept of Inheritance, Encapsulation, and Abstraction in Python is key to becoming proficient in Object-Oriented Programming (OOP). These concepts are the foundation of OOP and are crucial for building software that is robust, maintainable, and scalable. Together, these principles help you write well-structured, maintainable, and scalable code. They make it easier to manage large codebases and improve collaboration among developers. By understanding and applying these concepts, you can create software that is not only efficient but also easier to understand, debug, and extend. This ultimately leads to the development of high-quality software that can adapt to changing requirements and technologies.