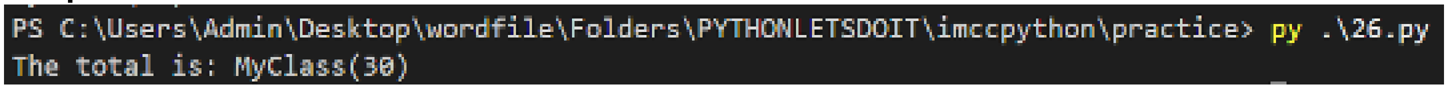
PYTHON HANDS-ON(HARD)

Name:-Sarita Paliwal
Roll no:- 2401146
Div:- B

26. Implement operator overloading for the + operator to add two objects of a userdefined class.
Input:-

```
class MyClass:
    def __init__(self, value):
        self.value = value
    def __add__(self, other):
        if isinstance(other, MyClass):
            # Adding the values of two MyClass objects
            return MyClass(self.value + other.value)
        return NotImplemented

    def __repr__(self):
        return f"MyClass({self.value})"

# Example usage
obj1 = MyClass(10)
obj2 = MyClass(20)

# Using the overloaded + operator
obj3 = obj1 + obj2

print("The total is:",obj3)  # Output: MyClass(30)
```

Output:-

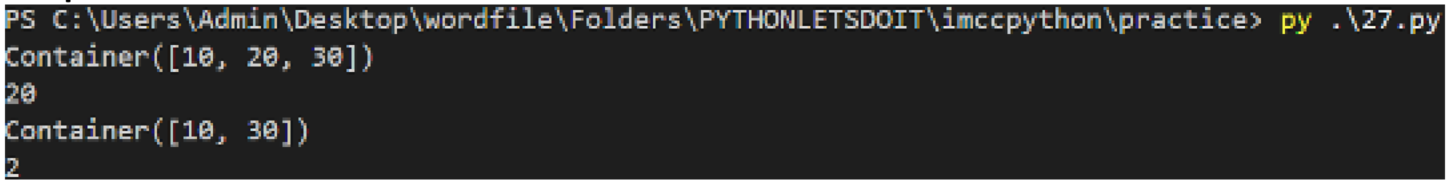


```
PS C:\Users\Admin\Desktop\wordfile\Folders\PYTHONLETSDOIT\imccpython\practice> py .\26.py
The total is: MyClass(30)
```

27. Write a Python program to demonstrate the use of delegation in a class with container-like behavior
Input:-

```
class Container:
    def __init__(self):
        # Internal list to hold items
        self._items = []

    # Delegate the append operation to the internal list
    def add(self, item):
        self._items.append(item)

    # Delegate the removal operation to the internal list
    def remove(self, item):
        if item in self._items:
            self._items.remove(item)
        else:
            print(f"{item} not found in Container.")

    # Delegate the length operation to the internal list
    def __len__(self):
        return len(self._items)

    # Delegate the item retrieval operation to the internal list
    def __getitem__(self, index):
        return self._items[index]

    # Delegate the string representation of the object
    def __repr__(self):
        return f"Container({self._items})"


# Example usage:
Container = Container()
Container.add(10)  # Adding item
Container.add(20)
Container.add(30)

print(Container)  # Output: Container([10, 20, 30])

# Accessing items through delegation
print(Container[1])  # Output: 20

# Removing an item
Container.remove(20)
print(Container)  # Output: Container([10, 30])

# Using len() function, which delegates to __len__
print(len(Container))  # Output: 2
```

Output:-



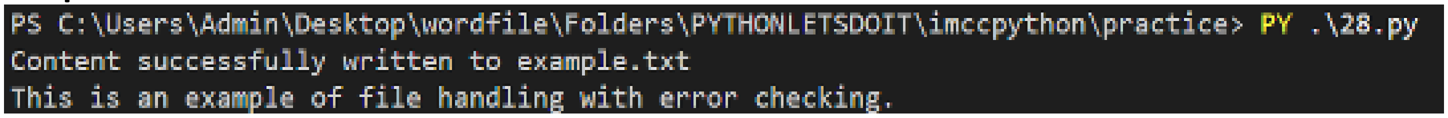
```
PS C:\Users\Admin\Desktop\wordfile\Folders\PYTHONLETSDOIT\imccpython\practice> py .\27.py
Container([10, 20, 30])
20
Container([10, 30])
2
```

28.Create a class with methods for reading and writing files, and ensure proper error handling using exceptions.

Input:-

```
class FileHandler:
    def __init__(self, filename):
        self.filename = filename
    def write_to_file(self, content):
        #Write content to a file with error handling.
        try:
            with open(self.filename, "w") as file:
                file.write(content)
            print(f"Content successfully written to {self.filename}")
        except PermissionError:
            print(f"Error: You do not have permission to write to {self.filename}")
        except FileNotFoundError:
            print(f"Error: The file {self.filename} was not found.")
        except Exception as e:
            print(f"An unexpected error occurred while writing: {e}")

    def read_from_file(self):
        "Read content from a file with error handling."
        try:
            with open(self.filename, "r") as file:
                Content = file.read()
            return Content
        except FileNotFoundError:
            print(f"Error: The file {self.filename} was not found.")
        except PermissionError:
            print(f"Error: You do not have permission to read {self.filename}")
        except Exception as e:
            print(f"An unexpected error occurred while reading: {e}")
        return None

# Example usage:
File_handler = FileHandler("example.txt")

# Writing to a file
File_handler.write_to_file("This is an example of file handling with error checking.")

# Reading from a file
Content = File_handler.read_from_file()
print(Content)
```

Output:-



```
PS C:\Users\Admin\Desktop\wordfile\Folders\PYTHONLETSDOIT\imccpython\practice> PY .\28.py
Content successfully written to example.txt
This is an example of file handling with error checking.
```

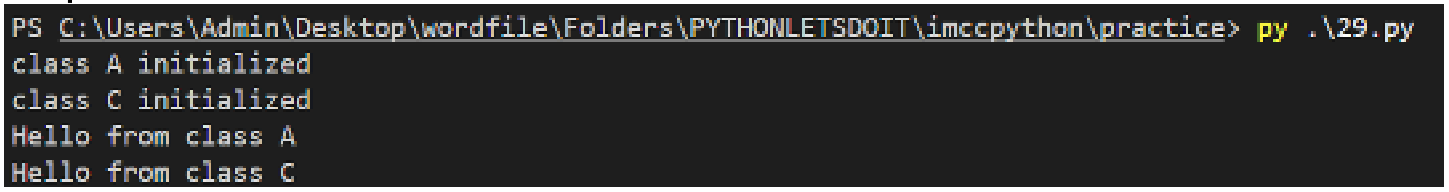
29.Define a class hierarchy to demonstrate multiple inheritance and handle potential conflicts with the super keyword.

Input:-

```
class A:
    def __init__(self):
        print("class A initialized")

    def greet(self):
```

```python
        print("Hello from class A")

class B:
    def __init__(self):
        print("class B initialized")

    def greet(self):
        print("Hello from class B")

class C(A, B):
    def __init__(self):
        super().__init__() #calling A's constructor
        print("class C initialized")

    def greet(self):
        # Using super() to ensure the correct greet method is called
        super().greet()
        print("Hello from class C")

# Create an object of class C
Obj = C()
Obj.greet() #method resolution order
```

Output:-



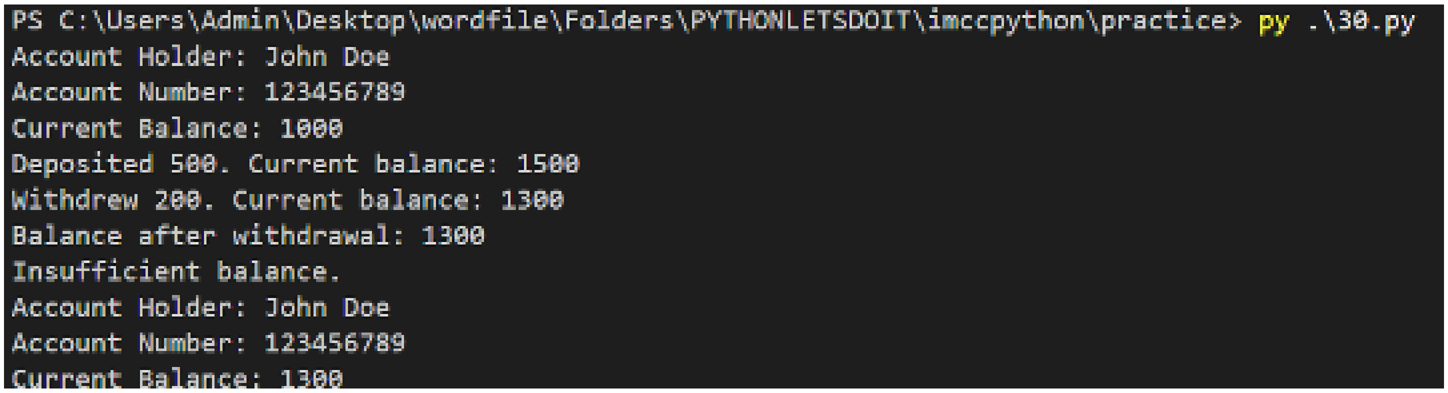
```
PS C:\Users\Admin\Desktop\wordfile\Folders\PYTHONLETSDOIT\imccpython\practice> py .\29.py
class A initialized
class C initialized
Hello from class A
Hello from class C
```

30. Implement a Python program to simulate a simple banking system with features like account creation, deposit, withdrawal, and balance inquiry using OOP principles.

Input:-

```python
class BankAccount:
    def __init__(self, account_holder, account_number, initial_balance=0):
        # Initialize the account with account holder’s name, account number, and balance
        self.account_holder = account_holder
        self.account_number = account_number
        self.balance = initial_balance

    def deposit(self, amount):
        "Deposit a specified amount into the account."
        if amount > 0:
            self.balance += amount
            print(f"Deposited {amount}. Current balance: {self.balance}")
        else:
            print("Deposit amount must be positive.")

    def withdraw(self, amount):
        "Withdraw a specified amount from the account."
        if amount <= 0:
            print("Withdrawal amount must be positive.")
        elif amount > self.balance:
            print("Insufficient balance.")
        else:
            self.balance -= amount
            print(f"Withdrew {amount}. Current balance: {self.balance}")

    def get_balance(self):
        "return the current balance of the account."
        return self.balance

    def display_account_info(self):
        "Display account information."
        print(f"Account Holder: {self.account_holder}")
        print(f"Account Number: {self.account_number}")
        print(f"Current Balance: {self.balance}")

# Example usage:

# Creating an account
Account1 = BankAccount("John Doe", "123456789", 1000)

# Display account info
Account1.display_account_info()

# Deposit money into the account
Account1.deposit(500)

# Withdraw money from the account
Account1.withdraw(200)

# Check balance
print(f"Balance after withdrawal: {Account1.get_balance()}")

# Try to withdraw more than the balance
Account1.withdraw(2000)

# Display final account info
Account1.display_account_info()
```

Output:-



```
PS C:\Users\Admin\Desktop\wordfile\Folders\PYTHONLETSDOIT\imccpython\practice> py .\30.py
Account Holder: John Doe
Account Number: 123456789
Current Balance: 1000
Deposited 500. Current balance: 1500
Withdrew 200. Current balance: 1300
Balance after withdrawal: 1300
Insufficient balance.
Account Holder: John Doe
Account Number: 123456789
Current Balance: 1300
```