ASSIGNMENT -3
BANKING SYSTEM

Task 1: Conditional Statements

In a bank, you have been given the task is to create a program that checks if a customer is eligible for a loan based on their credit score and income. The eligibility criteria are as follows:

- Credit Score must be above 700.
- Annual Income must be at least $50,000.

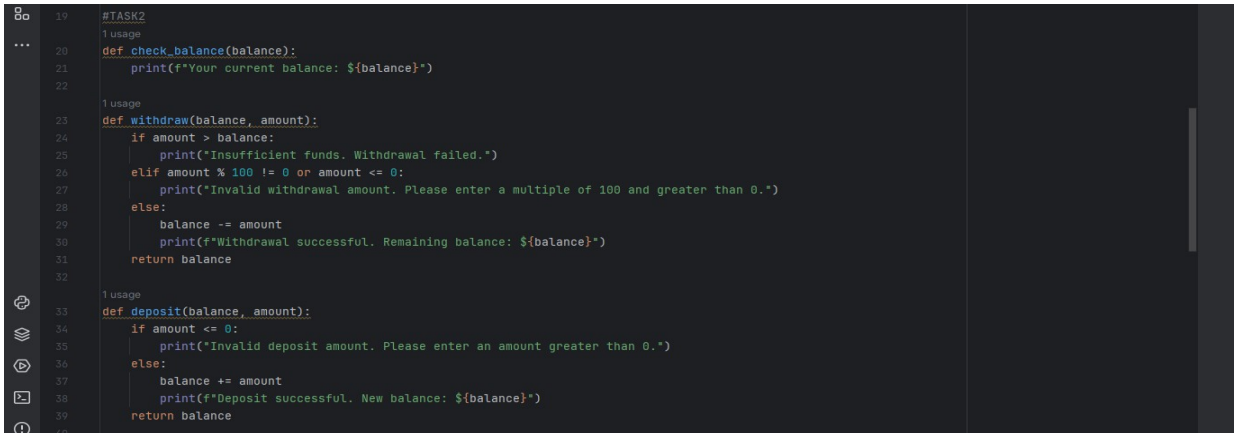
Tasks:
1. Write a program that takes the customer's credit score and annual income as input.
2. Use conditional statements (if-else) to determine if the customer is eligible for a loan.
3. Display an appropriate message based on eligibility

```python
def check_loan_eligibility(credit_score, annual_income):
    if credit_score > 700 and annual_income >= 50000:
        return True
    else:
        return False

1 usage
def main():

    credit_score = int(input("Enter your credit score: "))
    annual_income = float(input("Enter your annual income: $"))

    eligibility = check_loan_eligibility(credit_score, annual_income)

    if eligibility:
        print("Congratulations! You are eligible for a loan.")
    else:
        print("Sorry, you are not eligible for a loan at this time.")

if __name__ == "__main__":
    main()
```

```
Enter your credit score: 725
Enter your annual income: $100000
Congratulations! You are eligible for a loan.

Process finished with exit code 0
```

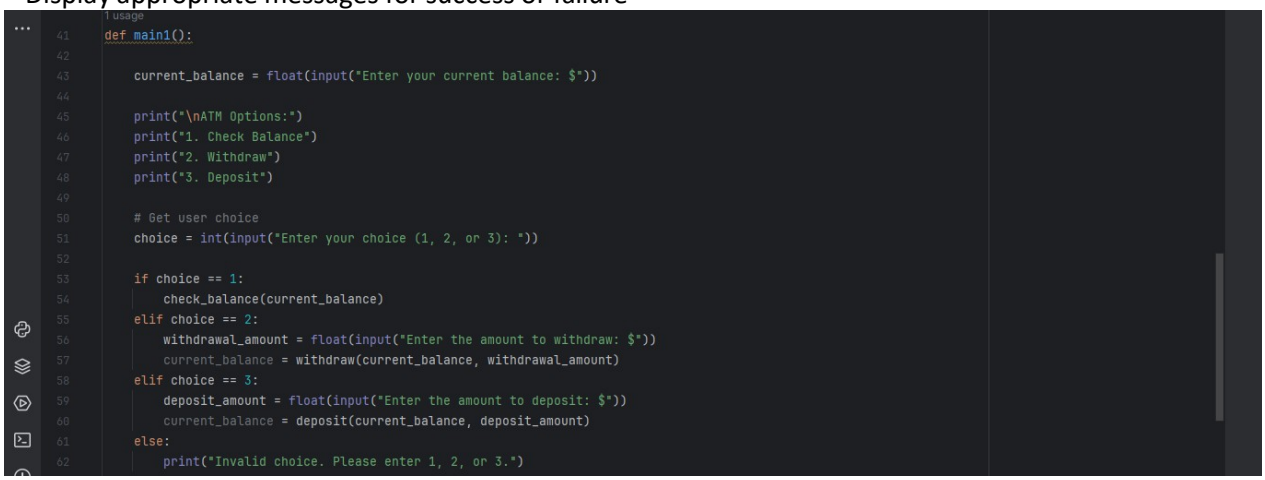
Task 2: Nested Conditional Statement

Create a program that simulates an ATM transaction. Display options such as "Check Balance," "Withdraw," "Deposit,".



```python
#TASK2
1 usage
def check_balance(balance):
    print(f"Your current balance: ${balance}")

1 usage
def withdraw(balance, amount):
    if amount > balance:
        print("Insufficient funds. Withdrawal failed.")
    elif amount % 100 != 0 or amount <= 0:
        print("Invalid withdrawal amount. Please enter a multiple of 100 and greater than 0.")
    else:
        balance -= amount
        print(f"Withdrawal successful. Remaining balance: ${balance}")
    return balance

1 usage
def deposit(balance, amount):
    if amount <= 0:
        print("Invalid deposit amount. Please enter an amount greater than 0.")
    else:
        balance += amount
        print(f"Deposit successful. New balance: ${balance}")
    return balance
```

Ask the user to enter their current balance and the amount they want to withdraw or deposit.
Implement checks to ensure that the withdrawal amount is not greater than the available balance and that the withdrawal amount is in multiples of 100 or 500.
Display appropriate messages for success or failure



```python
1 usage
def main1():

    current_balance = float(input("Enter your current balance: $"))

    print("\nATM Options:")
    print("1. Check Balance")
    print("2. Withdraw")
    print("3. Deposit")

    # Get user choice
    choice = int(input("Enter your choice (1, 2, or 3): "))

    if choice == 1:
        check_balance(current_balance)
    elif choice == 2:
        withdrawal_amount = float(input("Enter the amount to withdraw: $"))
        current_balance = withdraw(current_balance, withdrawal_amount)
    elif choice == 3:
        deposit_amount = float(input("Enter the amount to deposit: $"))
        current_balance = deposit(current_balance, deposit_amount)
    else:
        print("Invalid choice. Please enter 1, 2, or 3.")
```

```
Enter your current balance: $20000

ATM Options:
1. Check Balance
2. Withdraw
3. Deposit
Enter your choice (1, 2, or 3): 2
Enter the amount to withdraw: $10000
Withdrawal successful. Remaining balance: $10000.0

Process finished with exit code 0
```

Task 3: Loop Structures

You are responsible for calculating compound interest on savings accounts for bank customers. You need to calculate the future balance for each customer's savings account after a certain number of years.

Tasks:

1. Create a program that calculates the future balance of a savings account.
2. Use a loop structure (e.g., for loop) to calculate the balance for multiple customers
3. Prompt the user to enter the initial balance, annual interest rate, and the number of years.
4. Calculate the future balance using the formula: future_balance = initial_balance * (1 + annual_interest_rate/100) years.

```python
#TASK 3
1 usage
def calculate_future_balance(initial_balance, annual_interest_rate, years):
    future_balance = initial_balance * (1 + annual_interest_rate / 100) ** years
    return future_balance


1 usage
def main2():
    num_customers = int(input("Enter the number of customers: "))

    for customer in range(1, num_customers + 1):
        print(f"\nCustomer {customer}:")

        initial_balance = float(input("Enter the initial balance: $"))
        annual_interest_rate = float(input("Enter the annual interest rate (%): "))
        years = int(input("Enter the number of years: "))

        future_balance = calculate_future_balance(initial_balance, annual_interest_rate, years)

        print(f"Future Balance for Customer {customer}: ${future_balance:.2f}")
```

5. Display the future balance for each customer.

```
Enter the number of customers: 3

Customer 1:
Enter the initial balance: $1000
Enter the annual interest rate (%): 5
Enter the number of years: 5
Future Balance for Customer 1: $1276.28

Customer 2:
Enter the initial balance: $2000
Enter the annual interest rate (%): 5
Enter the number of years: 5
Future Balance for Customer 2: $2552.56

Customer 3:
Enter the initial balance: $5000
Enter the annual interest rate (%): 5
Enter the number of years: 5
Future Balance for Customer 3: $6381.41

Process finished with exit code 0
```

Task 4: Looping, Array and Data Validation

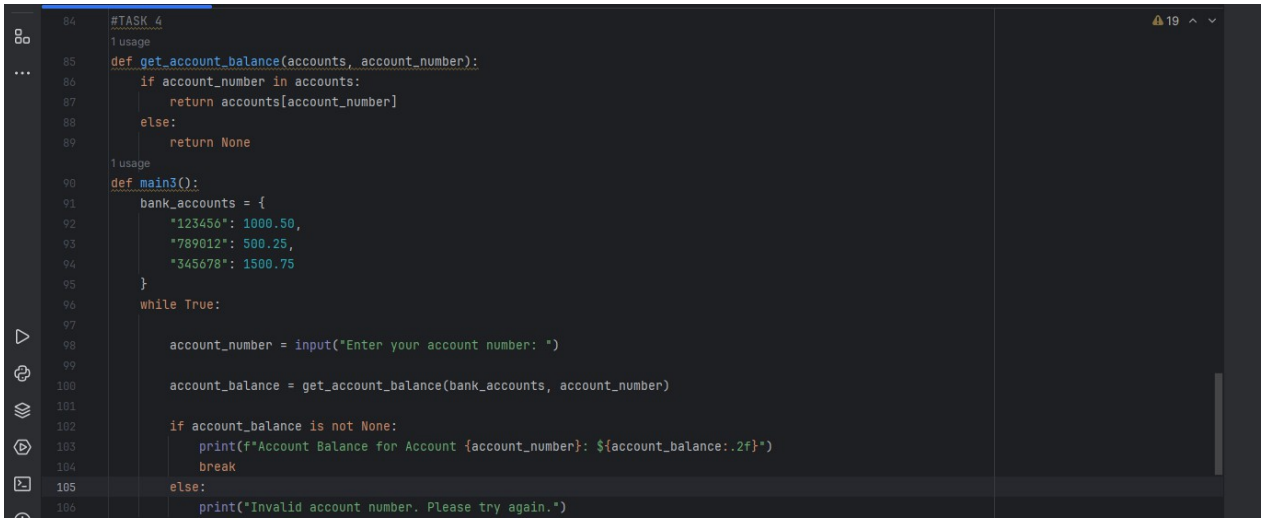
You are tasked with creating a program that allows bank customers to check their account balances. The program should handle multiple customer accounts, and the customer should be able to enter their account number, balance to check the balance.

Tasks:

1. Create a Python program that simulates a bank with multiple customer accounts.

2. Use a loop (e.g., while loop) to repeatedly ask the user for their account number and balance until they enter a valid account number.

3. Validate the account number entered by the user.



```python
#TASK 4
1 usage
def get_account_balance(accounts, account_number):
    if account_number in accounts:
        return accounts[account_number]
    else:
        return None
1 usage
def main3():
    bank_accounts = {
        "123456": 1000.50,
        "789012": 500.25,
        "345678": 1500.75
    }
    while True:

        account_number = input("Enter your account number: ")

        account_balance = get_account_balance(bank_accounts, account_number)

        if account_balance is not None:
            print(f"Account Balance for Account {account_number}: ${account_balance:.2f}")
            break
        else:
            print("Invalid account number. Please try again.")
```

4. If the account number is valid, display the account balance. If not, ask the user to try again

```
Run    controlstructure
C:\Users\welcome_\Desktop\Assignment-python\.venv\Scripts\python.exe C:\Users\welcome_\Desktop\Assignment-python\Assignment3-python\controlstructure.py
Enter your account number: 123457
Invalid account number. Please try again.
Enter your account number: 123458
Invalid account number. Please try again.
Enter your account number: 123456
Account Balance for Account 123456: $1000.50

Process finished with exit code 0
```

Task 5: Password Validation

Write a program that prompts the user to create a password for their bank account. Implement if conditions to validate the password according to these rules:

- The password must be at least 8 characters long.
- It must contain at least one uppercase letter.
- It must contain at least one digit.

```python
#TASK 5
1 usage
def is_valid_password(password):
    if len(password) < 8:
        return False
    if not any(char.isupper() for char in password):
        return False

    if not any(char.isdigit() for char in password):
        return False

    return True


1 usage
def main4():
    password = input("Create your bank account password: ")

    if is_valid_password(password):
        print("Password is valid. Account created successfully!")
    else:
        print("Invalid password. Please make sure your password meets the specified criteria.")
```

- Display appropriate messages to indicate whether their password is valid or not.

```
Run    controlstructure
C:\Users\welcome_\Desktop\Assignment-python\.venv\Scripts\python.exe C:\Users\welcome_\Desktop\Assignment-python\Assignment3-python\controlstructure.py
Create your bank account password: Aniket12
Password is valid. Account created successfully!

Process finished with exit code 0
```

Task 6: Password Validation

Create a program that maintains a list of bank transactions (deposits and withdrawals) for a customer. Use a while loop to allow the user to keep adding transactions until they choose to exit.

```python
def display_transaction_history(transaction_history):
    print("\nTransaction History:")
    for transaction in transaction_history:
        print(transaction)

1 usage
def main():
    transaction_history = []
    while True:
        print("\nOptions:")
        print("1. Add Deposit")
        print("2. Add Withdrawal")
        print("3. Exit")
        choice = input("Enter your choice (1, 2, or 3): ")

        if choice == '1':
            deposit_amount = float(input("Enter the deposit amount: $"))
            transaction_history.append(f"Deposit: +${deposit_amount:.2f}")
        elif choice == '2':
            withdrawal_amount = float(input("Enter the withdrawal amount: $"))
            transaction_history.append(f"Withdrawal: -${withdrawal_amount:.2f}")
        elif choice == '3':
            display_transaction_history(transaction_history)
            print("Exiting program. Thank you!")
            break
```

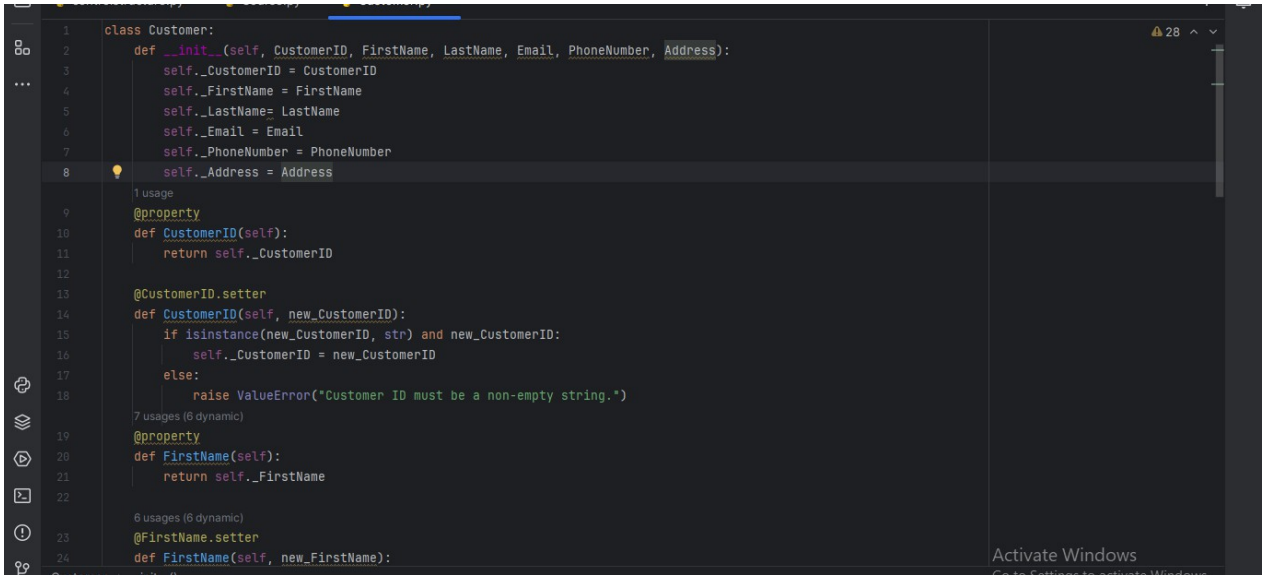
Display the transaction history upon exit using looping statements.

```
Options:
1. Add Deposit
2. Add Withdrawal
3. Exit
Enter your choice (1, 2, or 3): 1
Enter the deposit amount: $50000

Options:
1. Add Deposit
2. Add Withdrawal
3. Exit
Enter your choice (1, 2, or 3): 20000
Invalid choice. Please enter 1, 2, or 3.

Options:
1. Add Deposit
2. Add Withdrawal
3. Exit
Enter your choice (1, 2, or 3): 3

Transaction History:
Deposit: +$50000.00
Exiting program. Thank you!
```

OOPS, Collections and Exception Handling

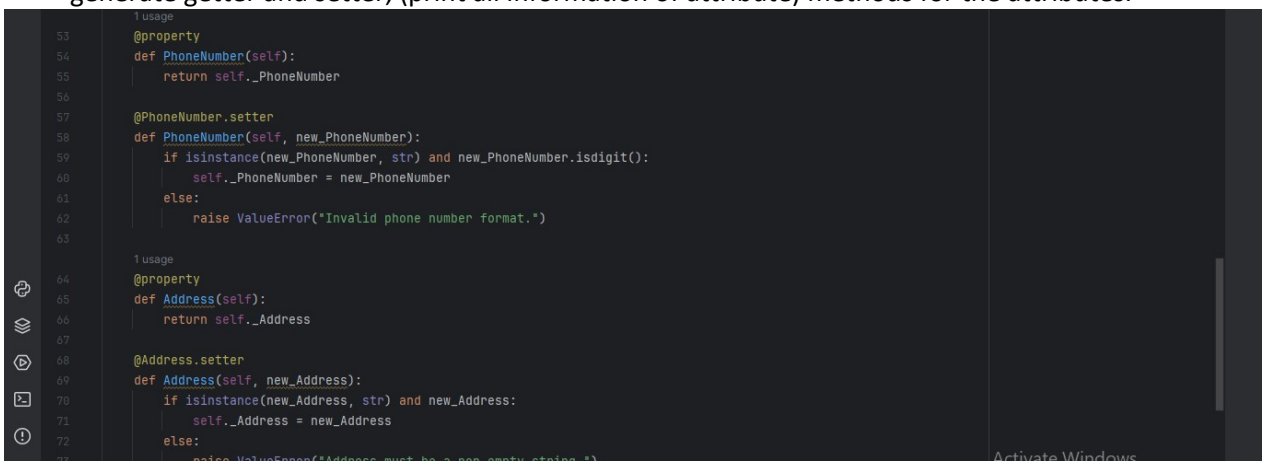
Task 7: Class & Object

1. Create a `Customer` class with the following confidential attributes:
   - Attributes
   - o Customer ID
   - o First Name
   - o Last Name
   - o Email Address
   - o Phone Number
   - o Address



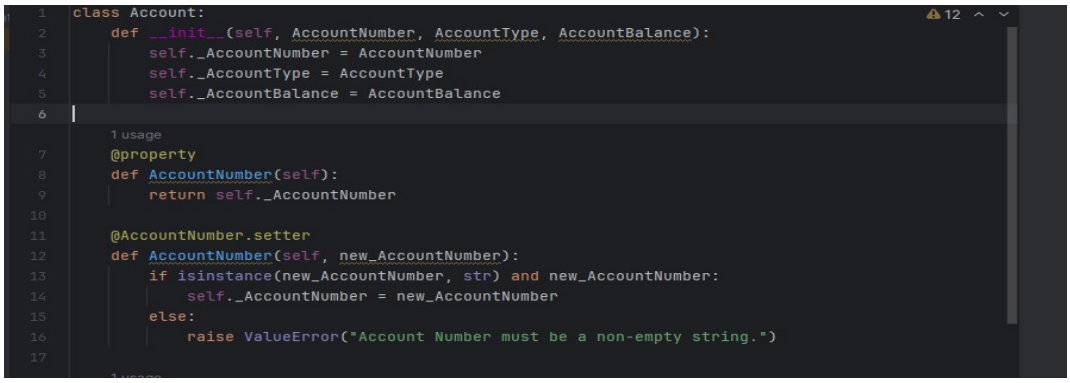
- Constructor and Methods

o Implement default constructors and overload the constructor with Customer attributes, generate getter and setter, (print all information of attribute) methods for the attributes.



```
1 usage
@property
def PhoneNumber(self):
    return self._PhoneNumber

@PhoneNumber.setter
def PhoneNumber(self, new_PhoneNumber):
    if isinstance(new_PhoneNumber, str) and new_PhoneNumber.isdigit():
        self._PhoneNumber = new_PhoneNumber
    else:
        raise ValueError("Invalid phone number format.")

1 usage
@property
def Address(self):
    return self._Address

@Address.setter
def Address(self, new_Address):
    if isinstance(new_Address, str) and new_Address:
        self._Address = new_Address
    else:
```

2. Create an `Account` class with the following confidential attributes:
   • Attributes
   o Account Number
   o Account Type (e.g., Savings, Current) o Account Balance



```
class Account:
    def __init__(self, AccountNumber, AccountType, AccountBalance):
        self._AccountNumber = AccountNumber
        self._AccountType = AccountType
        self._AccountBalance = AccountBalance

    1 usage
    @property
    def AccountNumber(self):
        return self._AccountNumber

    @AccountNumber.setter
    def AccountNumber(self, new_AccountNumber):
        if isinstance(new_AccountNumber, str) and new_AccountNumber:
            self._AccountNumber = new_AccountNumber
        else:
            raise ValueError("Account Number must be a non-empty string.")
```

• **Constructor and Methods**
o Implement default constructors and overload the constructor with Account attributes,
o Generate getter and setter, (print all information of attribute) methods for the attributes.

```python
    def AccountType(self):
        return self._AccountType

    @AccountType.setter
    def AccountType(self, new_AccountType):
        if isinstance(new_AccountType, str) and new_AccountType:
            self._AccountType = new_AccountType
        else:
            raise ValueError("Account Type must be a non-empty string.")

    1 usage
    @property
    def AccountBalance(self):
        return self._AccountBalance

    @AccountBalance.setter
    def AccountBalance(self, new_AccountBalance):
        if isinstance(new_AccountBalance, (int, float)) and new_AccountBalance >= 0:
            self._AccountBalance = new_AccountBalance
        else:
            raise ValueError("Account Balance must be a non-negative number.")
```

o **Add methods to the `Account` class to allow deposits and withdrawals.**

- deposit(amount: float): Deposit the specified amount into the account.

- withdraw(amount: float): Withdraw the specified amount from the account. withdraw amount only if there is sufficient fund else display insufficient balance

- calculate_interest(): method for calculating interest amount for the available balance. interest rate is fixed to 4.5%

```python
    def deposit(self, amount):
        if amount > 0:
            self._AccountBalance += amount
            print(f"Deposited ${amount:.2f}. New balance: ${self._AccountBalance:.2f}")
        else:
            raise ValueError("Deposit amount must be greater than 0.")

    1 usage (1 dynamic)
    def withdraw(self, amount):
        if amount > 0:
            if amount <= self._AccountBalance:
                self._AccountBalance -= amount
                print(f"Withdrew ${amount:.2f}. New balance: ${self._AccountBalance:.2f}")
            else:
                print("Insufficient balance. Withdrawal failed.")
        else:
            raise ValueError("Withdrawal amount must be greater than 0.")

    1 usage (1 dynamic)
    def calculate_interest(self):
        interest_rate = 4.5
        interest_amount = (interest_rate / 100) * self._AccountBalance
        print(f"Interest calculated: ${interest_amount:.2f}")
        return interest_amount
```

• Create a Bank class to represent the banking system. Perform the following operation in main method: o create object for account class by calling parameter constructor.

o deposit(amount: float): Deposit the specified amount into the account.
o withdraw(amount: float): Withdraw the specified amount from the account.
o calculate_interest(): Calculate and add interest to the account balance for savings accounts.

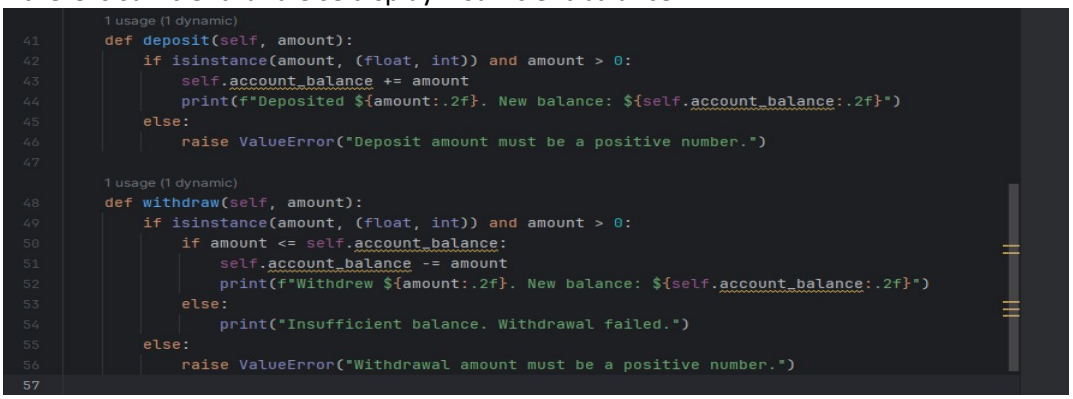
```python
from Account import Account
1 usage
class Bank:
    def __init__(self):
        self.accounts = []

    1 usage
    def create_account(self, account_number, account_type, initial_balance):
        account = Account(account_number, account_type, initial_balance)
        self.accounts.append(account)
        print(f"Account created: {account.AccountType} Account ({account.AccountNumber}) with ini

1 usage
def main():
    bank = Bank()

    bank.create_account( account_number: "AC001",  account_type: "Savings",  initial_balance: 1000)

    account1 = bank.accounts[0]
    account1.deposit(500)

    account1.withdraw(200)

    account1.calculate_interest()

if __name__ == "__main__":
    main()
```



```
C:\Users\welcome_\Desktop\Assignment-python\.venv\Scripts\python.exe C:\Users\welcome_\Desktop\Assignment-python\Assignment3-python\Bank.py
Account created: Savings Account (AC001) with initial balance $1000.00
Deposited $500.00. New balance: $1500.00
Withdrew $200.00. New balance: $1300.00
Interest calculated: $58.50

Process finished with exit code 0
```

Task 8: Inheritance and polymorphism

Overload the deposit and withdraw methods in Account class as mentioned below.
• deposit(amount: float): Deposit the specified amount into the account.
• withdraw(amount: float): Withdraw the specified amount from the account. withdraw amount only if there is sufficient fund else display insufficient balance.
• deposit(amount: int): Deposit the specified amount into the account
• withdraw(amount: int): Withdraw the specified amount from the account. withdraw amount only if there is sufficient fund else display insufficient balance
• deposit(amount: double): Deposit the specified amount into the account.
• withdraw(amount: double): Withdraw the specified amount from the account. withdraw amount only if there is sufficient fund else display insufficient balance



```
1 usage (1 dynamic)
def deposit(self, amount):
    if isinstance(amount, (float, int)) and amount > 0:
        self.account_balance += amount
        print(f"Deposited ${amount:.2f}. New balance: ${self.account_balance:.2f}")
    else:
        raise ValueError("Deposit amount must be a positive number.")

1 usage (1 dynamic)
def withdraw(self, amount):
    if isinstance(amount, (float, int)) and amount > 0:
        if amount <= self.account_balance:
            self.account_balance -= amount
            print(f"Withdrew ${amount:.2f}. New balance: ${self.account_balance:.2f}")
        else:
            print("Insufficient balance. Withdrawal failed.")
    else:
        raise ValueError("Withdrawal amount must be a positive number.")
```

Create Subclasses for Specific Account Types

• Create subclasses for specific account types (e.g., `SavingsAccount`, `CurrentAccount`) that inherit from the `Account` class.

o SavingsAccount: A savings account that includes an additional attribute for interest rate. override the calculate_interest() from Account class method to calculate interest based on the balance and interest rate.

o CurrentAccount: A current account that includes an additional attribute overdraftLimit. A current account with no interest. Implement the withdraw() method to allow overdraft up to a certain limit (configure a constant for the overdraft limit).

```python
class SavingsAccount(Account):
    def __init__(self, account_number, account_balance, interest_rate):
        super().__init__(account_number, account_balance)
        self.interest_rate = interest_rate

    1 usage (1 dynamic)
    def calculate_interest(self):
        interest_amount = (self.interest_rate / 100) * self.account_balance
        self.account_balance += interest_amount
        print(f"Interest calculated and added: ${interest_amount:.2f}. New balance: ${self.account_balance:.2f}")
class CurrentAccount(Account):
    OVERDRAFT_LIMIT = 1000
    def __init__(self, account_number, account_balance):
        super().__init__(account_number, account_balance)
    1 usage (1 dynamic)
    def withdraw(self, amount):
        if amount > 0:
            available_balance = self.account_balance + self.OVERDRAFT_LIMIT
            if amount <= available_balance:
                self.account_balance -= amount
                print(f"Withdrew ${amount:.2f}. New balance: ${self.account_balance:.2f}")
            else:
                print("Withdrawal limit exceeded. Withdrawal failed.")
        else:
            raise ValueError("Withdrawal amount must be greater than 0.")
```

.

Create a Bank class to represent the banking system.Perform the following operation in main method:

• Display menu for user to create object for account class by calling parameter constructor. Menu should display options `SavingsAccount` and `CurrentAccount`. user can choose any one option to create account. use switch case for implementation.

• deposit(amount: float): Deposit the specified amount into the account.

```python
from Account import Account,SavingsAccount,CurrentAccount
1 usage
class Bank:
    1 usage
    def create_account(self):
        print("Select account type:")
        print("1. Savings Account")
        print("2. Current Account")
        choice = int(input("Enter your choice (1 or 2): "))
        account_number = input("Enter account number: ")
        initial_balance = float(input("Enter initial balance: "))

        if choice == 1:
            interest_rate = float(input("Enter interest rate for savings account: "))
            return SavingsAccount(account_number, initial_balance, interest_rate)
        elif choice == 2:
            return CurrentAccount(account_number, initial_balance)
        else:
            print("Invalid choice. Creating a generic account.")
            return Account(account_number, AccountType: "Generic", initial_balance)

    1 usage
    def main(self):
        print("Customer Create Acccount first follow below steps")
        account = self.create_account()
```

• withdraw(amount: float): Withdraw the specified amount from the account. For saving account withdraw amount only if there is sufficient fund else display insufficient balance. For Current Account withdraw limit can exceed the available balance and should not exceed the overdraft limit.

- calculate_interest(): Calculate and add interest to the account balance for savings accounts.

```
            print("\nSelect operation:")
            print("1. Deposit")
            print("2. Withdraw")
            print("3. Calculate Interest (for Savings Account)")
            print("4. Exit")

            choice = int(input("Enter your choice (1-4): "))

            if choice == 1:
                amount = float(input("Enter deposit amount: "))
                account.deposit(amount)
            elif choice == 2:
                amount = float(input("Enter withdrawal amount: "))
                account.withdraw(amount)
            elif choice == 3:
                if isinstance(account, SavingsAccount):
                    account.calculate_interest()
                else:
                    print("Interest calculation not applicable for the current account.")
            elif choice == 4:
                print("Exiting the program.")
                break
            else:
                print("Invalid choice. Please choose a valid option.")

bank = Bank()
bank.main()
```



```
2. Withdraw
3. Calculate Interest (for Savings Account)
4. Exit
Enter your choice (1-4): 2
Enter withdrawal amount: 200
Withdrew $200.00. New balance: $1000.00

Select operation:
1. Deposit
2. Withdraw
3. Calculate Interest (for Savings Account)
4. Exit
Enter your choice (1-4): 3
Interest calculated and added: $50.00. New balance: $1050.00

Select operation:
1. Deposit
2. Withdraw
3. Calculate Interest (for Savings Account)
4. Exit
Enter your choice (1-4): 4
Exiting the program.
```

```
C:\Users\welcome_\Desktop\Assignment-python\.venv\Scripts\python.exe C:\Users\welcome_\Desktop\Assignment-python\Assignment3-python\Bank_Task8.py
Customer Create Acccount first follow below steps
Select account type:
1. Savings Account
2. Current Account
Enter your choice (1 or 2): 1
Enter account number: 101
Enter initial balance: 1000
Enter interest rate for savings account: 5

Select operation:
1. Deposit
2. Withdraw
3. Calculate Interest (for Savings Account)
4. Exit
Enter your choice (1-4): 1
Enter deposit amount: 200
Deposited $200.00. New balance: $1200.00

Select operation:
1. Deposit
2. Withdraw
3. Calculate Interest (for Savings Account)
4. Exit
```

Task 9: Abstraction

1. Create an abstract class BankAccount that represents a generic bank account. It should include the following attributes and methods:

• Attributes: Account number. o Customer name. o Balance

```python
from abc import ABC, abstractmethod

class BankAccount(ABC):
    def __init__(self,AccountNumber,CustomerName,Balance):
        self._Accountnumber = AccountNumber
        self._Customername = CustomerName
        self._Balance = Balance
```

. • Constructors:

o Implement default constructors and overload the constructor with Account attributes, generate getter and setter, print all information of attribute methods for the attributes.

```python
    @property
    def AccountNumber(self):
        return self._AccountNumber

    @AccountNumber.setter
    def AccountNumber(self, new_AccountNumber):
        if isinstance(new_AccountNumber, str) and new_AccountNumber:
            self._AccountNumber = new_AccountNumber
        else:
            raise ValueError("Account number must be a non-empty string.")
    @property
    def CustomerName(self):
        return self._CustomerName

    @CustomerName.setter
    def CustomerName(self, new_CustomerName):
        if isinstance(new_CustomerName, str) and new_CustomerName:
            self._CustomerName = new_CustomerName
        else:
            raise ValueError("Customer name must be a non-empty string.")
    @property
    def Balance(self):
        return self._Balance
```

• **Abstract methods:**

o deposit(amount: float): Deposit the specified amount into the account.

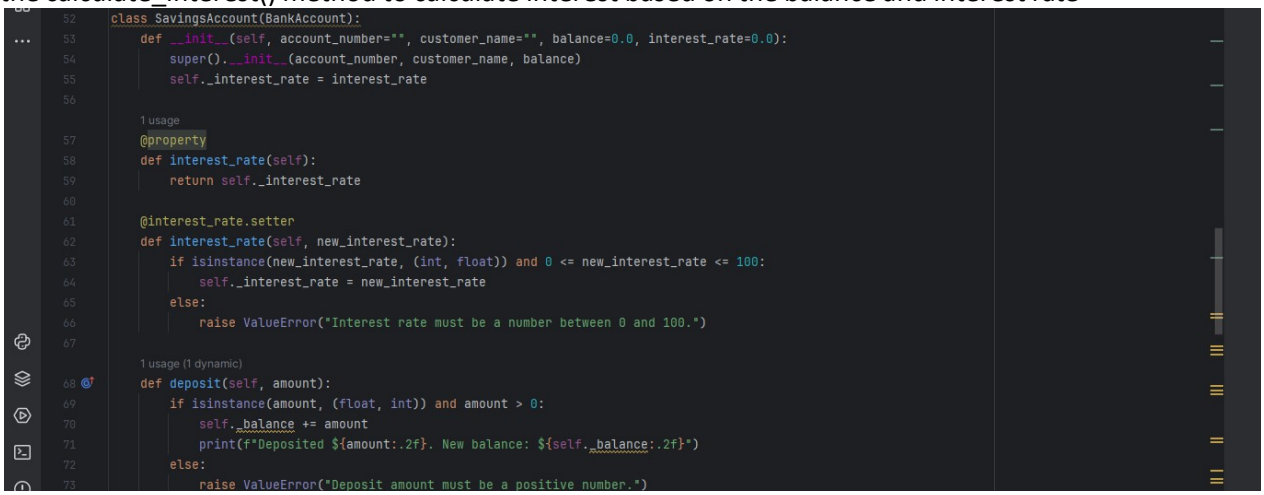
withdraw(amount: float): Withdraw the specified amount from the account (implement error handling for insufficient funds).

calculate_interest(): Abstract method for calculating interest.

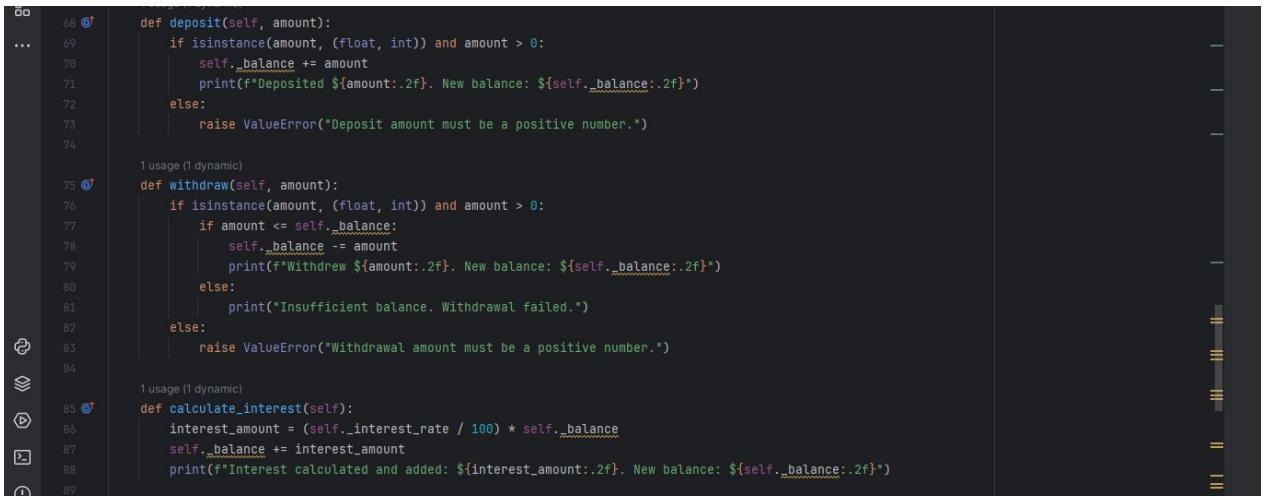
```python
    @abstractmethod
    def deposit(self, amount):
        pass

    @abstractmethod
    def withdraw(self, amount):
        pass

    @abstractmethod
    def calculate_interest(self):
        pass
```

Create two concrete classes that inherit from BankAccount:

• SavingsAccount: A savings account that includes an additional attribute for interest rate. Implement the calculate_interest() method to calculate interest based on the balance and interest rate

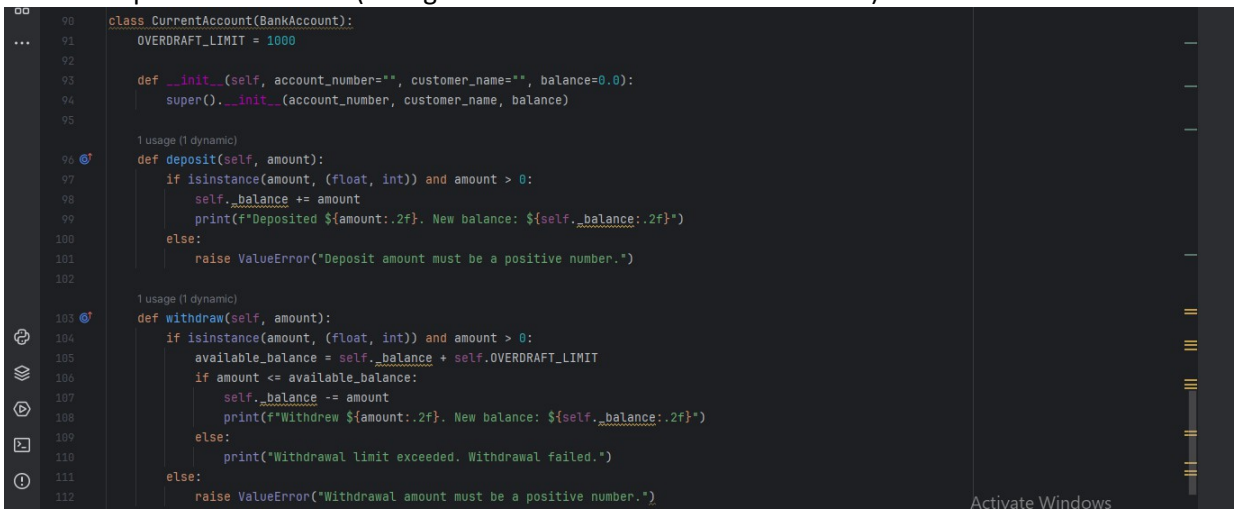


```python
class SavingsAccount(BankAccount):
    def __init__(self, account_number="", customer_name="", balance=0.0, interest_rate=0.0):
        super().__init__(account_number, customer_name, balance)
        self._interest_rate = interest_rate

    1 usage
    @property
    def interest_rate(self):
        return self._interest_rate

    @interest_rate.setter
    def interest_rate(self, new_interest_rate):
        if isinstance(new_interest_rate, (int, float)) and 0 <= new_interest_rate <= 100:
            self._interest_rate = new_interest_rate
        else:
            raise ValueError("Interest rate must be a number between 0 and 100.")

    1 usage (1 dynamic)
    def deposit(self, amount):
        if isinstance(amount, (float, int)) and amount > 0:
            self._balance += amount
            print(f"Deposited ${amount:.2f}. New balance: ${self._balance:.2f}")
        else:
            raise ValueError("Deposit amount must be a positive number.")
```



```python
    def deposit(self, amount):
        if isinstance(amount, (float, int)) and amount > 0:
            self._balance += amount
            print(f"Deposited ${amount:.2f}. New balance: ${self._balance:.2f}")
        else:
            raise ValueError("Deposit amount must be a positive number.")

    1 usage (1 dynamic)
    def withdraw(self, amount):
        if isinstance(amount, (float, int)) and amount > 0:
            if amount <= self._balance:
                self._balance -= amount
                print(f"Withdrew ${amount:.2f}. New balance: ${self._balance:.2f}")
            else:
                print("Insufficient balance. Withdrawal failed.")
        else:
            raise ValueError("Withdrawal amount must be a positive number.")

    1 usage (1 dynamic)
    def calculate_interest(self):
        interest_amount = (self._interest_rate / 100) * self._balance
        self._balance += interest_amount
        print(f"Interest calculated and added: ${interest_amount:.2f}. New balance: ${self._balance:.2f}")
```

. • CurrentAccount: A current account with no interest. Implement the withdraw() method to allow overdraft up to a certain limit (configure a constant for the overdraft limit).



```python
class CurrentAccount(BankAccount):
    OVERDRAFT_LIMIT = 1000

    def __init__(self, account_number="", customer_name="", balance=0.0):
        super().__init__(account_number, customer_name, balance)

    1 usage (1 dynamic)
    def deposit(self, amount):
        if isinstance(amount, (float, int)) and amount > 0:
            self._balance += amount
            print(f"Deposited ${amount:.2f}. New balance: ${self._balance:.2f}")
        else:
            raise ValueError("Deposit amount must be a positive number.")

    1 usage (1 dynamic)
    def withdraw(self, amount):
        if isinstance(amount, (float, int)) and amount > 0:
            available_balance = self._balance + self.OVERDRAFT_LIMIT
            if amount <= available_balance:
                self._balance -= amount
                print(f"Withdrew ${amount:.2f}. New balance: ${self._balance:.2f}")
            else:
                print("Withdrawal limit exceeded. Withdrawal failed.")
        else:
            raise ValueError("Withdrawal amount must be a positive number.")
```

Create a Bank class to represent the banking system.

Perform the following operation in main method

: • Display menu for user to create object for account class by calling parameter constructor. Menu should display options `SavingsAccount` and `CurrentAccount`. user can choose any one option to create account. use switch case for implementation. create_account should display sub menu to choose type of accounts. o Hint: Account acc = new SavingsAccount(); or Account acc = new CurrentAccount();

```
from BankAccount import SavingsAccount,CurrentAccount
1 usage
class Bank:
    1 usage
    def main(self):
        while True:
            print("1. Create Savings Account")
            print("2. Create Current Account")
            print("3. Exit")

            choice = input("Enter your choice (1, 2, or 3): ")

            if choice == '1':
                account_type = "Savings"
            elif choice == '2':
                account_type = "Current"
            elif choice == '3':
                print("Exiting the program.")
                break
            else:
                print("Invalid choice. Please enter 1, 2, or 3.")
                continue

            account = self.create_account(account_type)
            self.perform_operations(account)
```

• deposit(amount: float): Deposit the specified amount into the account.

• withdraw(amount: float): Withdraw the specified amount from the account. For saving account withdraw amount only if there is sufficient fund else display insufficient balance. For Current Account withdraw limit can exceed the available balance and should not exceed the overdraft limit. • calculate_interest(): Calculate and add interest to the account balance for savings accounts

```
    def create_account(self, account_type):
        account_number = input("Enter account number: ")
        customer_name = input("Enter customer name: ")
        initial_balance = float(input("Enter initial balance: "))

        if account_type == "Savings":
            interest_rate = float(input("Enter interest rate for savings account: "))
            return SavingsAccount(account_number, customer_name, initial_balance, interest_rate)
        elif account_type == "Current":
            return CurrentAccount(account_number, customer_name, initial_balance)
        else:
            print("Invalid account type.")
            return None

    1 usage
    def perform_operations(self, account):
        while True:
            print("\n1. Deposit")
            print("2. Withdraw")
            print("3. Calculate Interest (for Savings Account)")
            print("4. Exit")

            choice = input("Enter your choice (1, 2, 3, or 4): ")
```

```
            if choice == '1':
                amount = float(input("Enter deposit amount: "))
                account.deposit(amount)
            elif choice == '2':
                amount = float(input("Enter withdrawal amount: "))
                account.withdraw(amount)
            elif choice == '3' and isinstance(account, SavingsAccount):
                account.calculate_interest()
            elif choice == '4':
                print("Exiting account operations.")
                break
            else:
                print("Invalid choice. Please enter 1, 2, 3, or 4.")
                continue

bank = Bank()
bank.main()
```

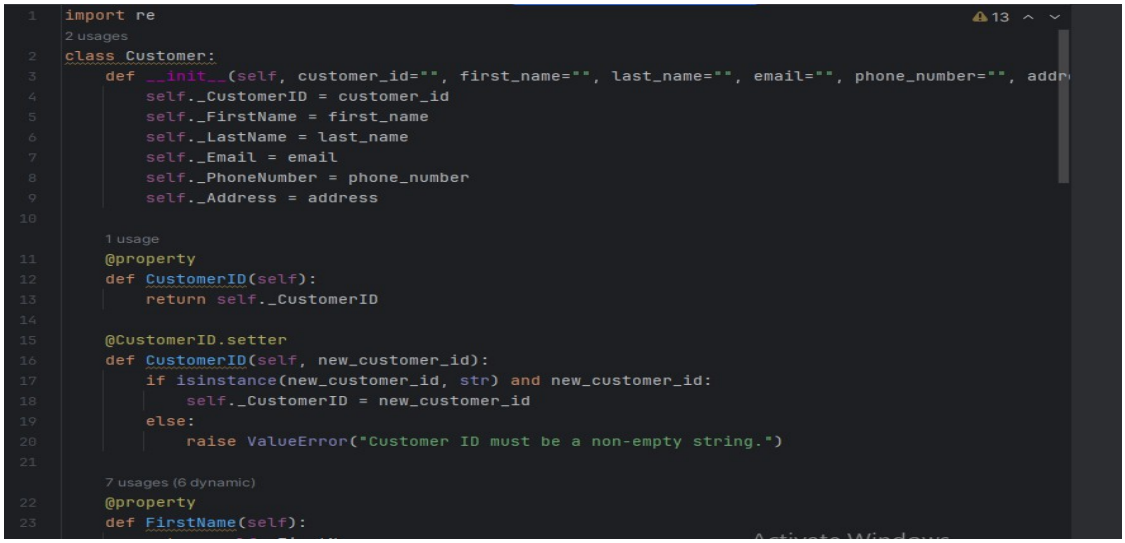

OUTPUT FOR SAVING ACCOUNT

OUTPUT FOR CURRENT ACCOUNT

Task 10: Has A Relation / Association

Create a `Customer` class with the following attributes:

• Customer ID • First Name • Last Name • Email Address (validate with valid email address) • Phone Number (Validate 10-digit phone number) • Address



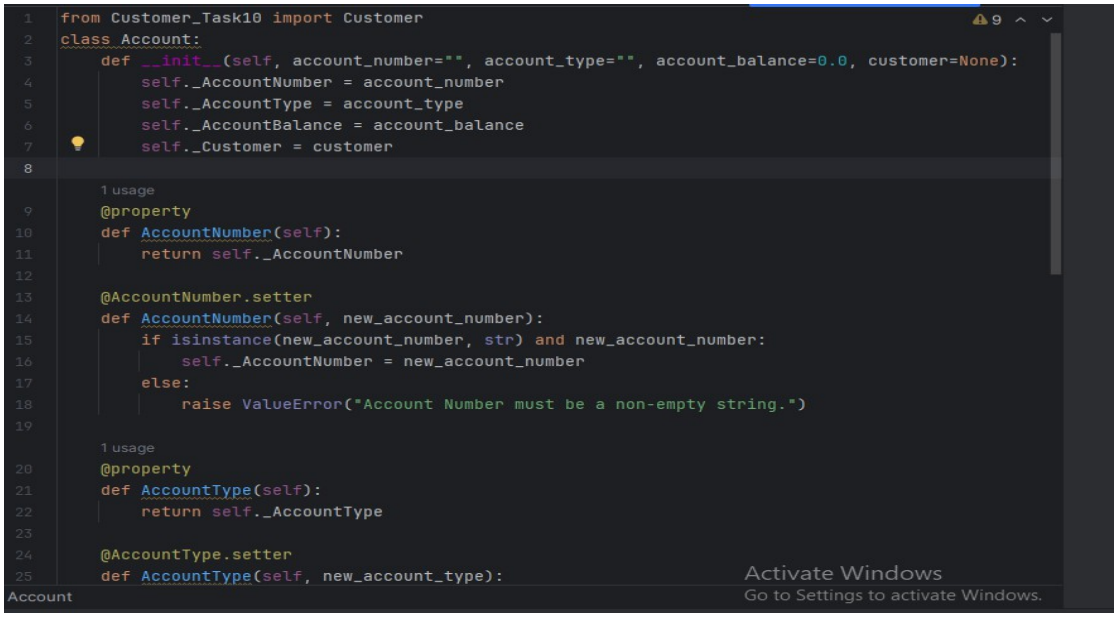
```python
import re
2 usages
class Customer:
    def __init__(self, customer_id="", first_name="", last_name="", email="", phone_number="", addr
        self._CustomerID = customer_id
        self._FirstName = first_name
        self._LastName = last_name
        self._Email = email
        self._PhoneNumber = phone_number
        self._Address = address

    1 usage
    @property
    def CustomerID(self):
        return self._CustomerID

    @CustomerID.setter
    def CustomerID(self, new_customer_id):
        if isinstance(new_customer_id, str) and new_customer_id:
            self._CustomerID = new_customer_id
        else:
            raise ValueError("Customer ID must be a non-empty string.")

    7 usages (6 dynamic)
    @property
    def FirstName(self):
```

• Methods and Constructor: o Implement default constructors and overload the constructor with Account attributes, generate getter, setter, print all information of attribute) methods for the attributes

```python
            self._LastName = new_last_name
        else:
            raise ValueError("Last Name must be a non-empty string.")

    1 usage
    @property
    def Email(self):
        return self._Email

    @Email.setter
    def Email(self, new_email):
        if isinstance(new_email, str) and re.match( pattern: r"[^@]+@[^@]+\.[^@]+", new_email):
            self._Email = new_email
        else:
            raise ValueError("Invalid Email format.")

    1 usage
    @property
    def PhoneNumber(self):
        return self._PhoneNumber

    @PhoneNumber.setter
    def PhoneNumber(self, new_phone_number):
        if isinstance(new_phone_number, str) and re.match( pattern: r"\d{10}", new_phone_number):
            self._PhoneNumber = new_phone_number
        else:
            raise ValueError("Invalid phone number format.")
```

Create an `Account` class with the following attributes:

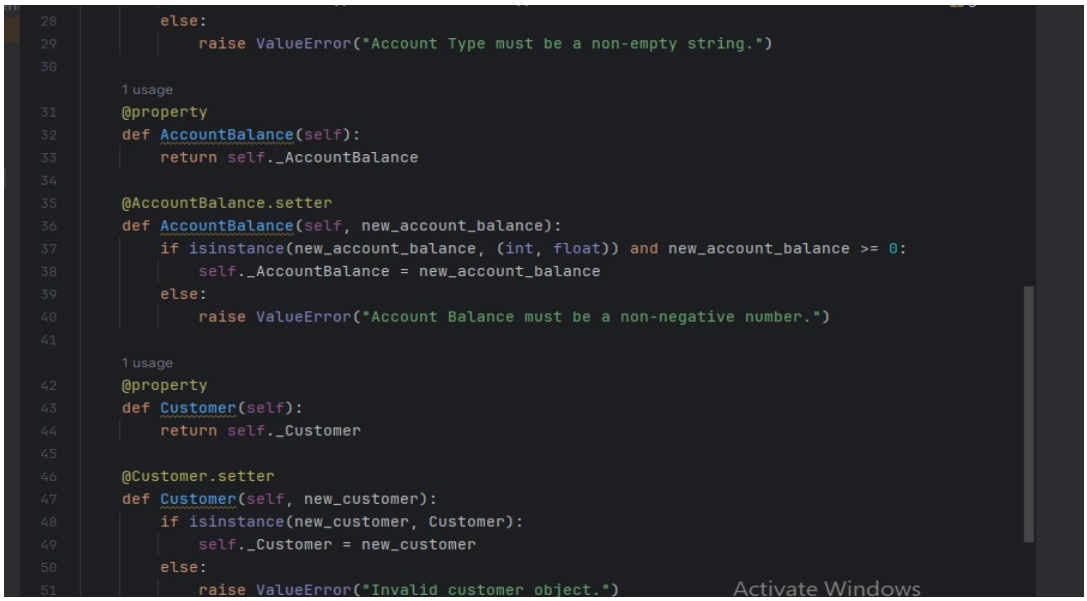
• Account Number (a unique identifier). • Account Type (e.g., Savings, Current) • Account Balance • Customer (the customer who owns the account)



```python
from Customer_Task10 import Customer
class Account:
    def __init__(self, account_number="", account_type="", account_balance=0.0, customer=None):
        self._AccountNumber = account_number
        self._AccountType = account_type
        self._AccountBalance = account_balance
        self._Customer = customer

    1 usage
    @property
    def AccountNumber(self):
        return self._AccountNumber

    @AccountNumber.setter
    def AccountNumber(self, new_account_number):
        if isinstance(new_account_number, str) and new_account_number:
            self._AccountNumber = new_account_number
        else:
            raise ValueError("Account Number must be a non-empty string.")

    1 usage
    @property
    def AccountType(self):
        return self._AccountType

    @AccountType.setter
    def AccountType(self, new_account_type):
```


Account

• Methods and Constructor: o Implement default constructors and overload the constructor with Account attributes, generate getter, setter, (print all information of attribute) methods for the attributes.



```python
        else:
            raise ValueError("Account Type must be a non-empty string.")

    1 usage
    @property
    def AccountBalance(self):
        return self._AccountBalance

    @AccountBalance.setter
    def AccountBalance(self, new_account_balance):
        if isinstance(new_account_balance, (int, float)) and new_account_balance >= 0:
            self._AccountBalance = new_account_balance
        else:
            raise ValueError("Account Balance must be a non-negative number.")

    1 usage
    @property
    def Customer(self):
        return self._Customer

    @Customer.setter
    def Customer(self, new_customer):
        if isinstance(new_customer, Customer):
            self._Customer = new_customer
        else:
            raise ValueError("Invalid customer object.")
```


Create a Bank Class and must have following requirements:

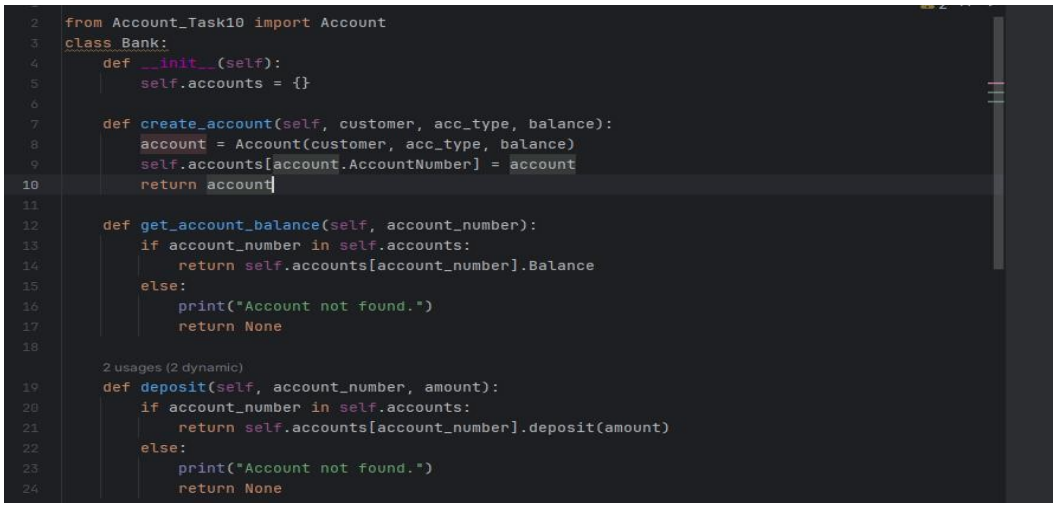
Create a Bank class to represent the banking system.

It should have the following methods:

• create_account(Customer customer, long accNo, String accType, float balance): Create a new bank account for the given customer with the initial balance.

• get_account_balance(account_number: long): Retrieve the balance of an account given its account number. should return the current balance of account.

• deposit(account_number: long, amount: float): Deposit the specified amount into the account. Should return the current balance of account.

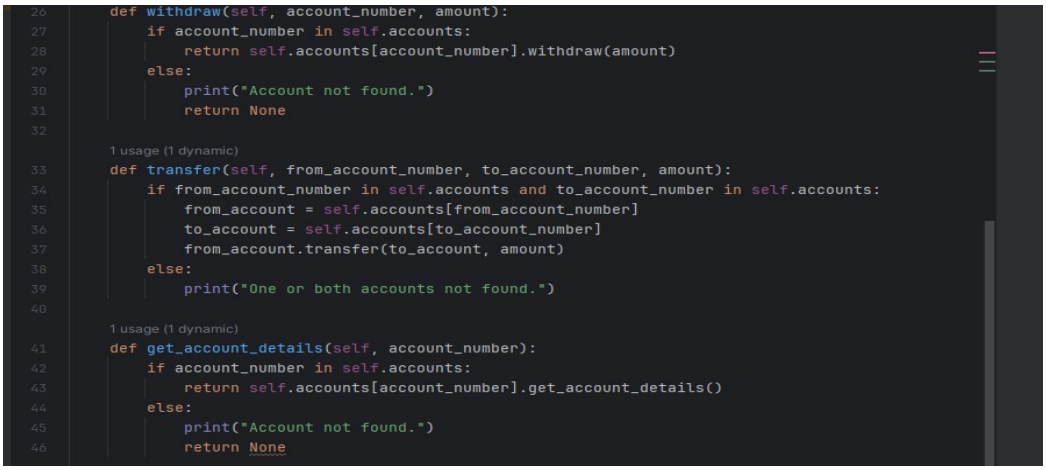


```python
from Account_Task10 import Account
class Bank:
    def __init__(self):
        self.accounts = {}

    def create_account(self, customer, acc_type, balance):
        account = Account(customer, acc_type, balance)
        self.accounts[account.AccountNumber] = account
        return account

    def get_account_balance(self, account_number):
        if account_number in self.accounts:
            return self.accounts[account_number].Balance
        else:
            print("Account not found.")
            return None

    2 usages (2 dynamic)
    def deposit(self, account_number, amount):
        if account_number in self.accounts:
            return self.accounts[account_number].deposit(amount)
        else:
            print("Account not found.")
            return None
```

• withdraw(account_number: long, amount: float): Withdraw the specified amount from the account. Should return the current balance of account.

• transfer(from_account_number: long, to_account_number: int, amount: float): Transfer money from one account to another.

• getAccountDetails(account_number: long): Should return the account and customer details.



```python
    def withdraw(self, account_number, amount):
        if account_number in self.accounts:
            return self.accounts[account_number].withdraw(amount)
        else:
            print("Account not found.")
            return None

    1 usage (1 dynamic)
    def transfer(self, from_account_number, to_account_number, amount):
        if from_account_number in self.accounts and to_account_number in self.accounts:
            from_account = self.accounts[from_account_number]
            to_account = self.accounts[to_account_number]
            from_account.transfer(to_account, amount)
        else:
            print("One or both accounts not found.")

    1 usage (1 dynamic)
    def get_account_details(self, account_number):
        if account_number in self.accounts:
            return self.accounts[account_number].get_account_details()
        else:
            print("Account not found.")
            return None
```

3. Ensure that account numbers are automatically generated when an account is created, starting from 1001 and incrementing for each new account.

```
class Account:
    account_number_counter=1001
    def __init__(self, account_type="", account_balance=0.0, customer=None):
        self._AccountNumber = Account.account_number_counter
        Account.account_number_counter+=1
        self._AccountType = account_type
        self._AccountBalance = account_balance
        self._Customer = customer
```

```
bank = Bank()
customer = Customer( customer_id: "C001", first_name: "John", last_name: "Doe", email: "john@example.com",
account1 = bank.create_account(customer, acc_type: "Savings", balance: 1000.0)
print(f"Account 1 Number: {account1.AccountNumber}")

account2 = bank.create_account(customer, acc_type: "Current", balance: 500.0)
print(f"Account 2 Number: {account2.AccountNumber}")
```

```
nv\Scripts\python.exe C:\Users\welcome_\Desktop\Assignment-python\Assignment3-python\Bank_Task10.py
```

4. Create a BankApp class with a main method to simulate the banking system. Allow the user to interact with the system by entering commands such as "create_account", "deposit", "withdraw", "get_balance", "transfer", "getAccountDetails" and "exit." create_account should display sub menu to choose type of accounts and repeat this operation until user exit.

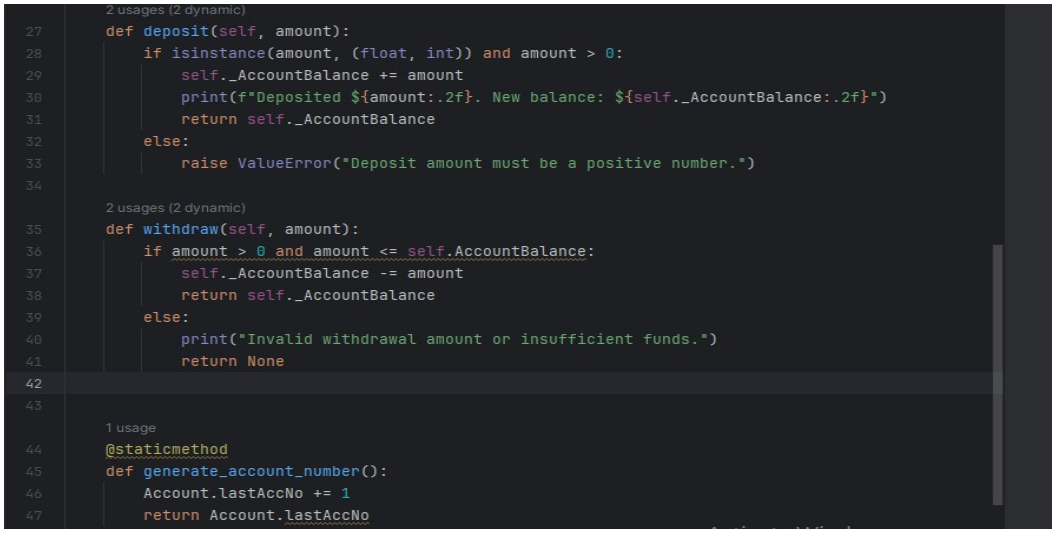
Task 11: Interface/abstract class, and Single Inheritance, static variable

1. Create a ‘Customer’ class as mentioned above task.

2. Create an class ‘Account’ that includes the following attributes. Generate account number using static variable.

```
class Account:
    lastAccNo = 1001

    def __init__(self, account_type="", account_balance=0.0, customer=None):
        self._AccountNumber = Account.generate_account_number()
        self._AccountType = account_type
        self._AccountBalance = account_balance
        self._Customer = customer

    1 usage (1 dynamic)
    @property
    def AccountNumber(self):
        return self._AccountNumber

    @property
    def AccountType(self):
        return self._AccountType

    3 usages (2 dynamic)
    @property
    def AccountBalance(self):
        return self._AccountBalance

    @property
    def Customer(self):
```

• Account Number (a unique identifier). • Account Type (e.g., Savings, Current) • Account Balance • Customer (the customer who owns the account) • lastAccNo

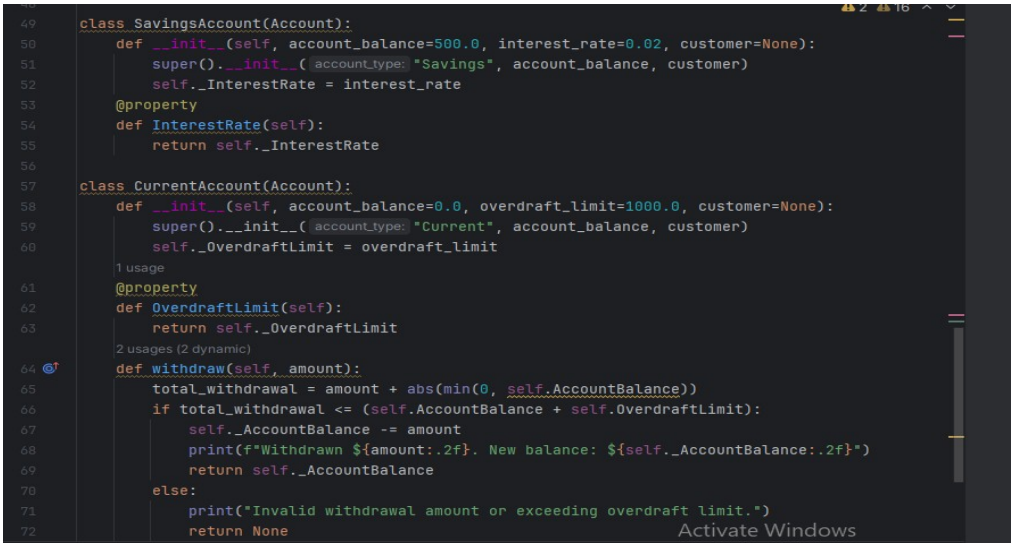


```
2 usages (2 dynamic)
def deposit(self, amount):
    if isinstance(amount, (float, int)) and amount > 0:
        self._AccountBalance += amount
        print(f"Deposited ${amount:.2f}. New balance: ${self._AccountBalance:.2f}")
        return self._AccountBalance
    else:
        raise ValueError("Deposit amount must be a positive number.")

2 usages (2 dynamic)
def withdraw(self, amount):
    if amount > 0 and amount <= self.AccountBalance:
        self._AccountBalance -= amount
        return self._AccountBalance
    else:
        print("Invalid withdrawal amount or insufficient funds.")
        return None


1 usage
@staticmethod
def generate_account_number():
    Account.lastAccNo += 1
    return Account.lastAccNo
```

3. Create three child classes that inherit the Account class and each class must contain below mentioned attribute:

• SavingsAccount: A savings account that includes an additional attribute for interest rate. Saving account should be created with minimum balance 500.

• CurrentAccount: A Current account that includes an additional attribute for overdraftLimit(credit limit). withdraw() method to allow overdraft up to a certain limit. withdraw limit can exceed the available balance and should not exceed the overdraft limit.



```
class SavingsAccount(Account):
    def __init__(self, account_balance=500.0, interest_rate=0.02, customer=None):
        super().__init__( account_type: "Savings", account_balance, customer)
        self._InterestRate = interest_rate
    @property
    def InterestRate(self):
        return self._InterestRate

class CurrentAccount(Account):
    def __init__(self, account_balance=0.0, overdraft_limit=1000.0, customer=None):
        super().__init__( account_type: "Current", account_balance, customer)
        self._OverdraftLimit = overdraft_limit
    1 usage
    @property
    def OverdraftLimit(self):
        return self._OverdraftLimit
    2 usages (2 dynamic)
    def withdraw(self, amount):
        total_withdrawal = amount + abs(min(0, self.AccountBalance))
        if total_withdrawal <= (self.AccountBalance + self.OverdraftLimit):
            self._AccountBalance -= amount
            print(f"Withdrawn ${amount:.2f}. New balance: ${self._AccountBalance:.2f}")
            return self._AccountBalance
        else:
            print("Invalid withdrawal amount or exceeding overdraft limit.")
            return None
```


• ZeroBalanceAccount: ZeroBalanceAccount can be created with Zero balance.

4.Create ICustomerServiceProvider interface/abstract class with following functions:

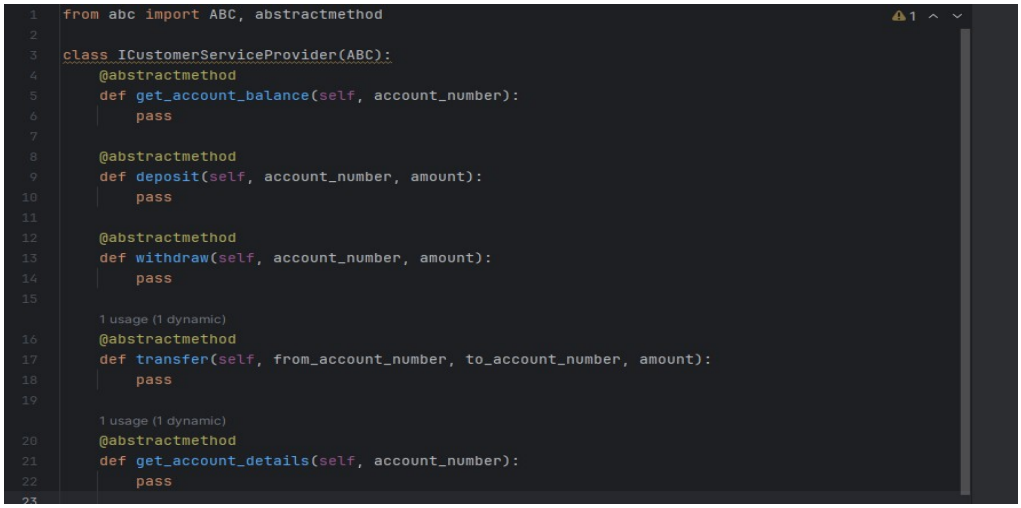
• get_account_balance(account_number: long): Retrieve the balance of an account given its account number. should return the current balance of account.

• deposit(account_number: long, amount: float): Deposit the specified amount into the account. Should return the current balance of account.

• withdraw(account_number: long, amount: float): Withdraw the specified amount from the account. Should return the current balance of account. A savings account should maintain a minimum balance and checking if the withdrawal violates the minimum balance rule.

• transfer(from_account_number: long, to_account_number: int, amount: float): Transfer money from one account to another.

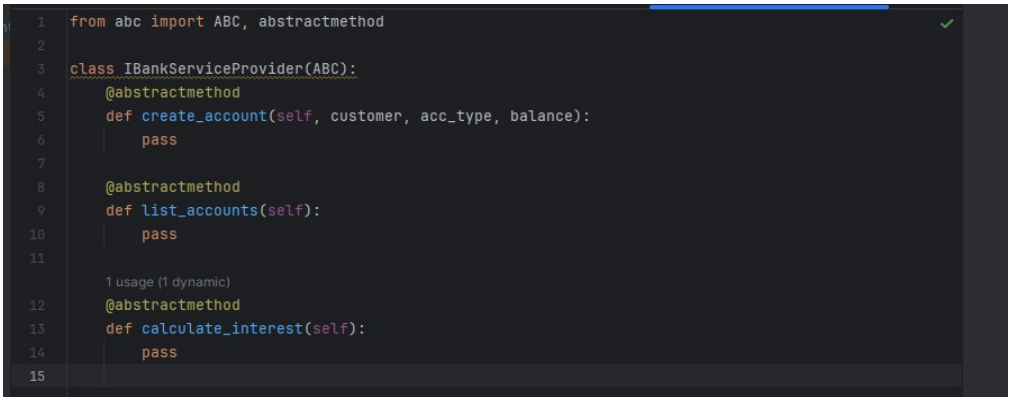
• getAccountDetails(account_number: long): Should return the account and customer details



```python
from abc import ABC, abstractmethod

class ICustomerServiceProvider(ABC):
    @abstractmethod
    def get_account_balance(self, account_number):
        pass

    @abstractmethod
    def deposit(self, account_number, amount):
        pass

    @abstractmethod
    def withdraw(self, account_number, amount):
        pass

    1 usage (1 dynamic)
    @abstractmethod
    def transfer(self, from_account_number, to_account_number, amount):
        pass

    1 usage (1 dynamic)
    @abstractmethod
    def get_account_details(self, account_number):
        pass
```

Create IBankServiceProvider interface/abstract class with following functions:

• create_account(Customer customer, long accNo, String accType, float balance): Create a new bank account for the given customer with the initial balance.

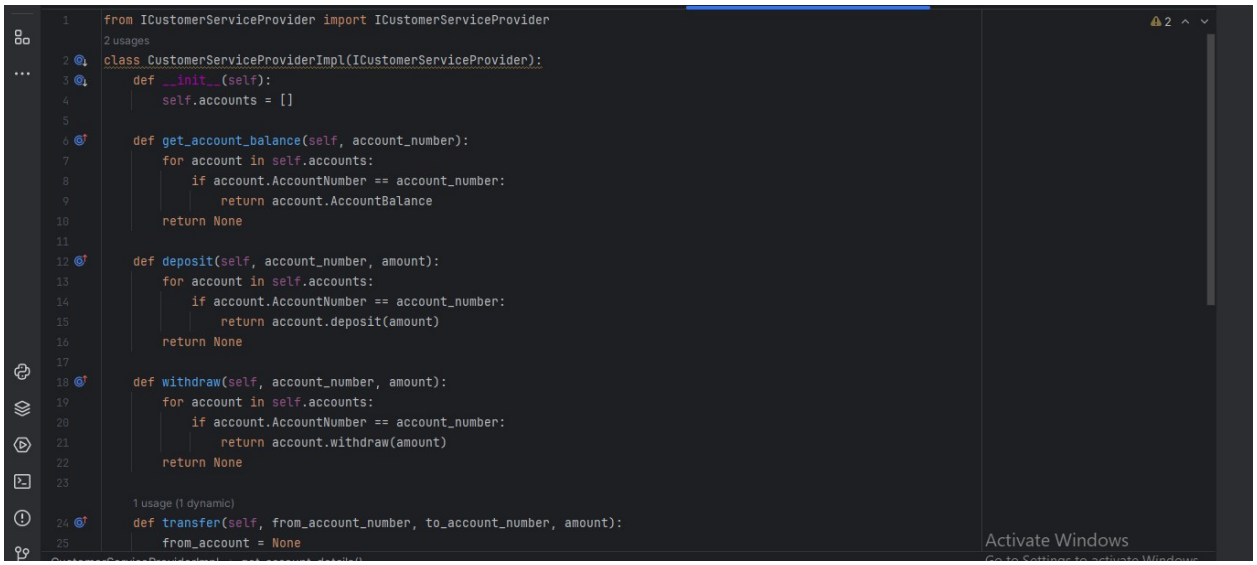


```python
from abc import ABC, abstractmethod

class IBankServiceProvider(ABC):
    @abstractmethod
    def create_account(self, customer, acc_type, balance):
        pass

    @abstractmethod
    def list_accounts(self):
        pass

    1 usage (1 dynamic)
    @abstractmethod
    def calculate_interest(self):
        pass
```

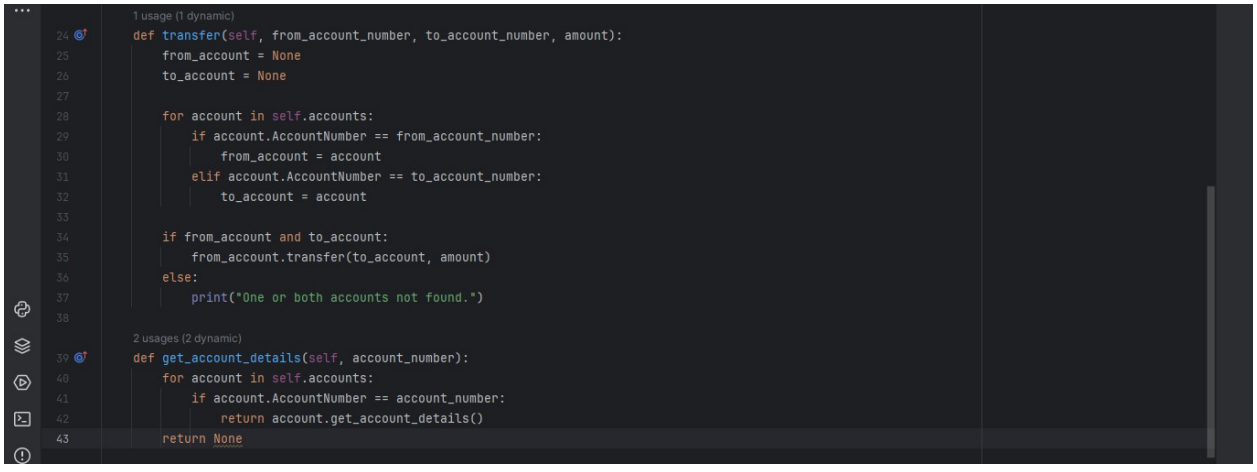
• listAccounts():Account[] accounts: List all accounts in the bank.

• calculateInterest(): the calculate_interest() method to calculate interest based on the balance and interest rate.

5. **Create CustomerServiceProviderImpl class which implements** ICustomerServiceProvider provide all implementation methods.



```python
from ICustomerServiceProvider import ICustomerServiceProvider
2 usages
class CustomerServiceProviderImpl(ICustomerServiceProvider):
    def __init__(self):
        self.accounts = []

    def get_account_balance(self, account_number):
        for account in self.accounts:
            if account.AccountNumber == account_number:
                return account.AccountBalance
        return None

    def deposit(self, account_number, amount):
        for account in self.accounts:
            if account.AccountNumber == account_number:
                return account.deposit(amount)
        return None

    def withdraw(self, account_number, amount):
        for account in self.accounts:
            if account.AccountNumber == account_number:
                return account.withdraw(amount)
        return None

    1 usage (1 dynamic)
    def transfer(self, from_account_number, to_account_number, amount):
        from_account = None
```

```python
    1 usage (1 dynamic)
    def transfer(self, from_account_number, to_account_number, amount):
        from_account = None
        to_account = None

        for account in self.accounts:
            if account.AccountNumber == from_account_number:
                from_account = account
            elif account.AccountNumber == to_account_number:
                to_account = account

        if from_account and to_account:
            from_account.transfer(to_account, amount)
        else:
            print("One or both accounts not found.")

    2 usages (2 dynamic)
    def get_account_details(self, account_number):
        for account in self.accounts:
            if account.AccountNumber == account_number:
                return account.get_account_details()
        return None
```

7. Create BankServiceProviderImpl class which inherits from CustomerServiceProviderImpl and implements IBankServiceProvider

• Attributes o accountList: Array of Accounts to store any account objects. o branchName and branchAddress as String objects

```python
from IBankServiceProvider import IBankServiceProvider
from CustomerServiceProviderImpl import CustomerServiceProviderImpl

class BankServiceProviderImpl(CustomerServiceProviderImpl, IBankServiceProvider):
    def __init__(self, branch_name, branch_address):
        super().__init__()
        self.accountList = []
        self.branchName = branch_name
        self.branchAddress = branch_address

    def create_account(self, customer, acc_type, balance):
        account = super().create_account(customer, acc_type, balance)
        self.accountList.append(account)
        return account

    def list_accounts(self):
        return self.accountList

    1 usage (1 dynamic)
    def calculate_interest(self):
        for account in self.accountList:
            pass
```

8. Create BankApp class and perform following operation: • main method to simulate the banking system. Allow the user to interact with the system by entering choice from menu such as "create_account", "deposit", "withdraw", "get_balance", "transfer", "getAccountDetails", "ListAccounts" and "exit." • create_account should display sub menu to choose type of accounts and repeat this operation until user exit

Task 12: Exception Handling

throw the exception whenever needed and Handle in main method

, 1. InsufficientFundException throw this exception when user try to withdraw amount or transfer amount to another account and the account runs out of money in the account.

2. InvalidAccountException throw this exception when user entered the invalid account number when tries to transfer amount, get account details classes.

3. OverDraftLimitExcededException thow this exception when current account customer try to with draw amount from the current account.

```python
class InsufficientFundException(Exception):
    def __init__(self, message="Insufficient funds."):
        self.message = message
        super().__init__(self.message)

class InvalidAccountException(Exception):
    def __init__(self, message="Invalid account number."):
        self.message = message
        super().__init__(self.message)

class OverDraftLimitExceededException(Exception):
    def __init__(self, message="Overdraft limit exceeded."):
        self.message = message
        super().__init__(self.message)
```

Task 14: Database Connectivity.

1. Create a ‘Customer’ class as mentioned above task.

2. Create an class ‘Account’ that includes the following attributes.

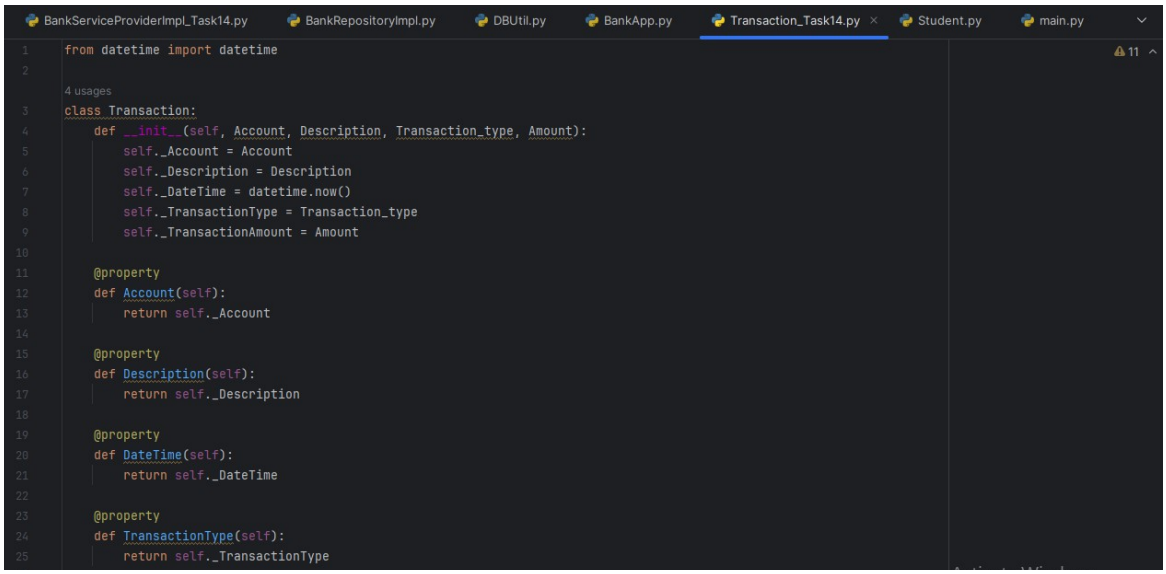
Generate account number using static variable.

• Account Number (a unique identifier). • Account Type (e.g., Savings, Current) • Account Balance • Customer (the customer who owns the account) • lastAccNo

3. Create a class ‘TRANSACTION’ that include following attributes

• Account • Description • Date and Time

• TransactionType(Withdraw, Deposit, Transfer) • TransactionAmount

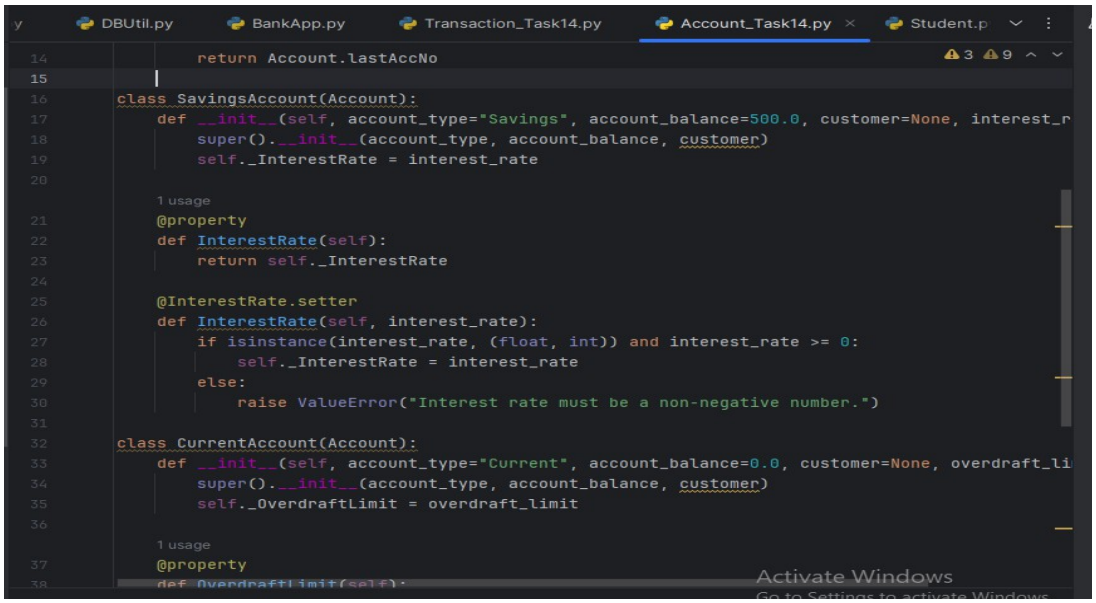


4. Create three child classes that inherit the Account class and each class must contain below mentioned attribute:

• SavingsAccount: A savings account that includes an additional attribute for interest rate. Saving account should be created with minimum balance 500.

• CurrentAccount: A Current account that includes an additional attribute for overdraftLimit(credit limit).

• ZeroBalanceAccount: ZeroBalanceAccount can be created with Zero balance.



5. Create ICustomerServiceProvider interface/abstract class with following functions:

• get_account_balance(account_number: long): Retrieve the balance of an account given its account number. should return the current balance of account.

• deposit(account_number: long, amount: float): Deposit the specified amount into the account. Should return the current balance of account.

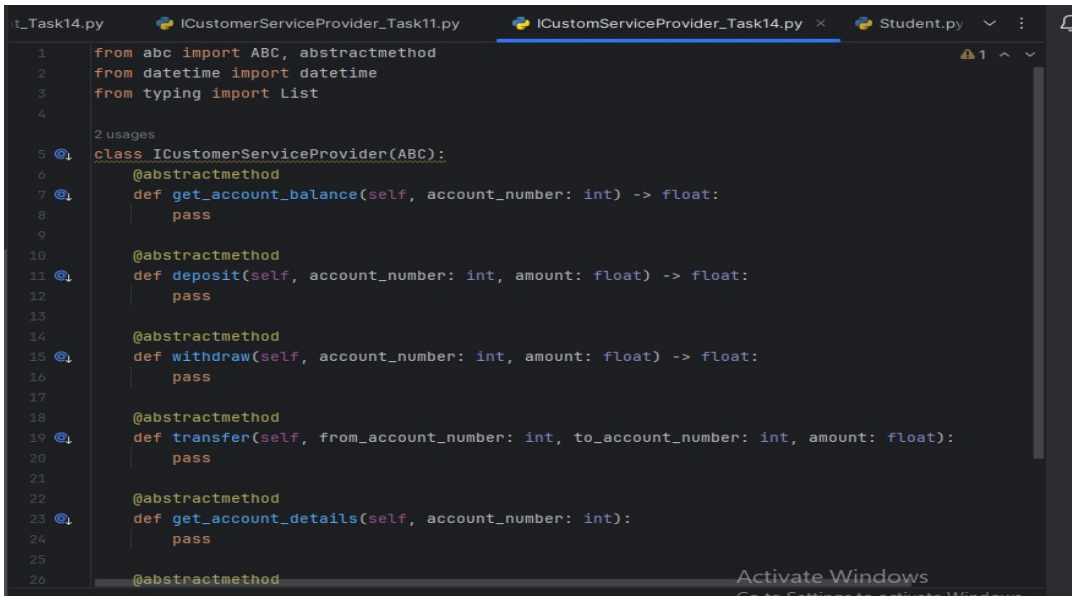
• withdraw(account_number: long, amount: float): Withdraw the specified amount from the account. Should return the current balance of account.

o A savings account should maintain a minimum balance and checking if the withdrawal violates the minimum balance rule.

o Current account customers are allowed withdraw overdraftLimit and available account balance. withdraw limit can exceed the available balance and should not exceed the overdraft limit.

• transfer(from_account_number: long, to_account_number: int, amount: float): Transfer money from one account to another. both account number should be validate from the database use getAccountDetails method.

• getAccountDetails(account_number: long): Should return the account and customer details. getTransations(account_number: long, FromDate:Date, ToDate: Date): Should return the list of transaction between two dates.

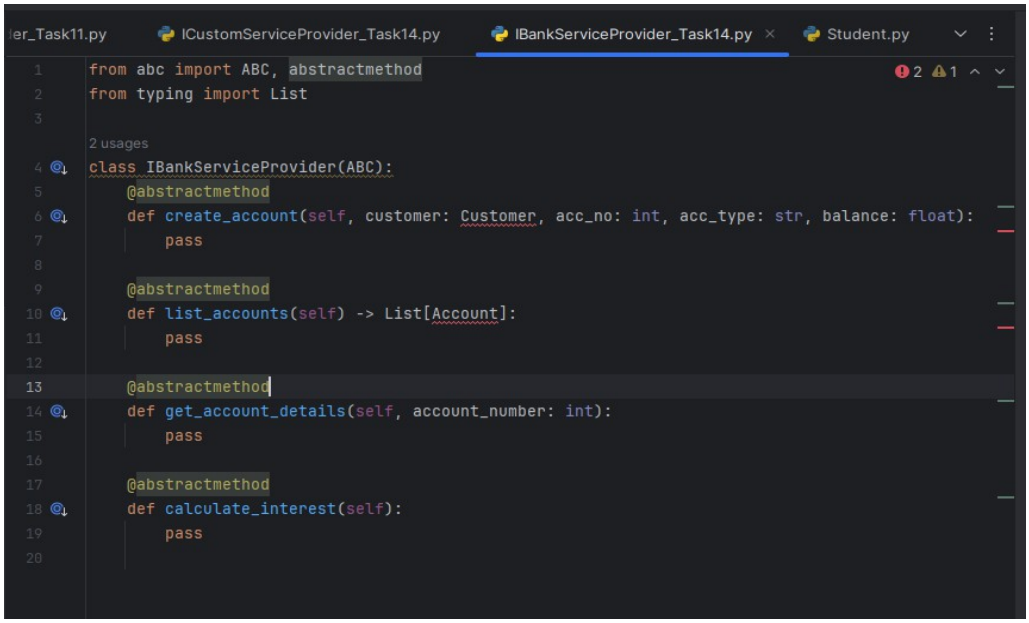


6. Create IBankServiceProvider interface/abstract class with following functions:

• create_account(Customer customer, long accNo, String accType, float balance): Create a new bank account for the given customer with the initial balance.

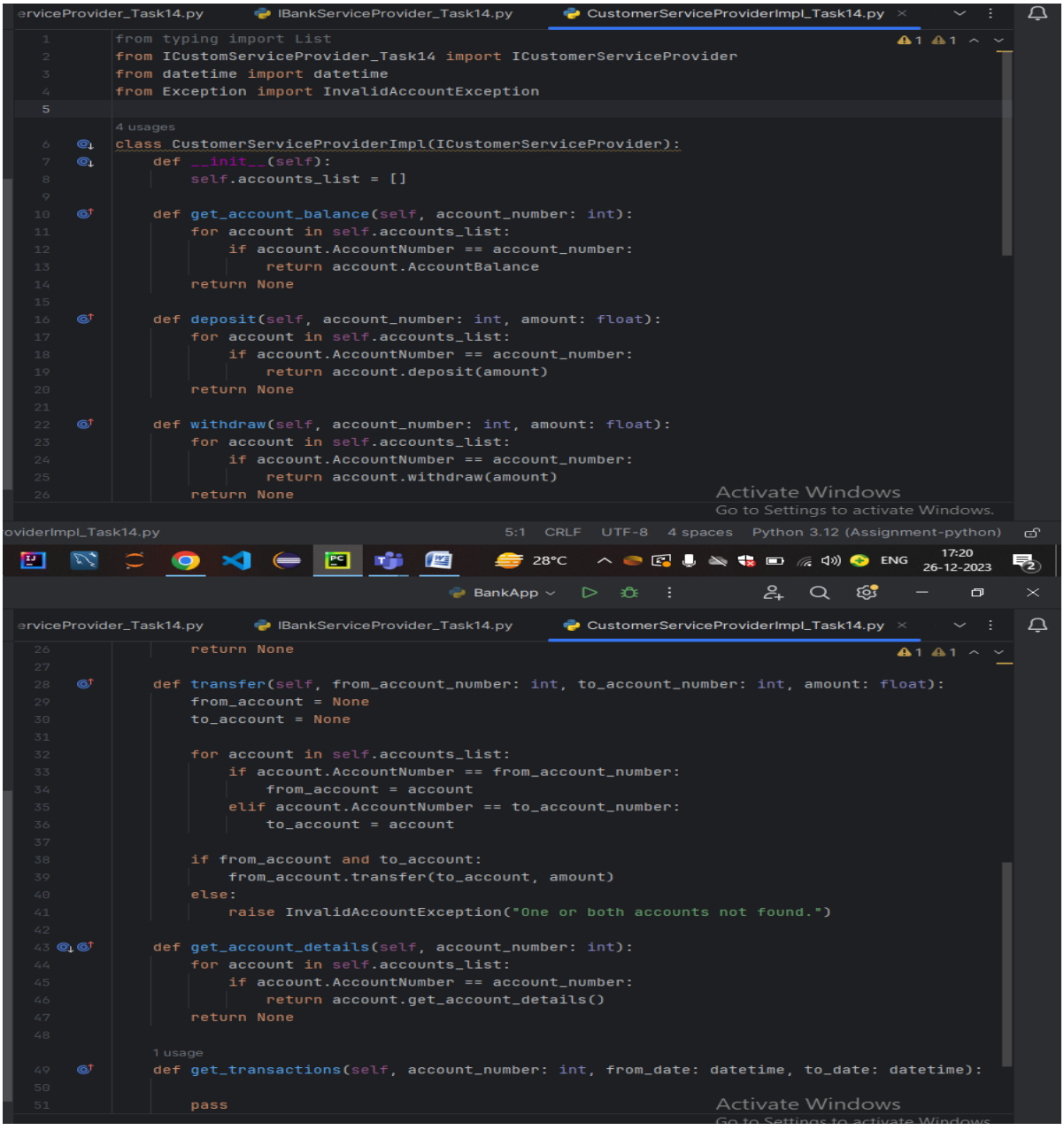
• listAccounts(): Array of BankAccount: List all accounts in the bank.(List[Account] accountsList)
• getAccountDetails(account_number: long): Should return the account and customer details.

• calculateInterest(): the calculate_interest() method to calculate interest based on the balance and interest rate.



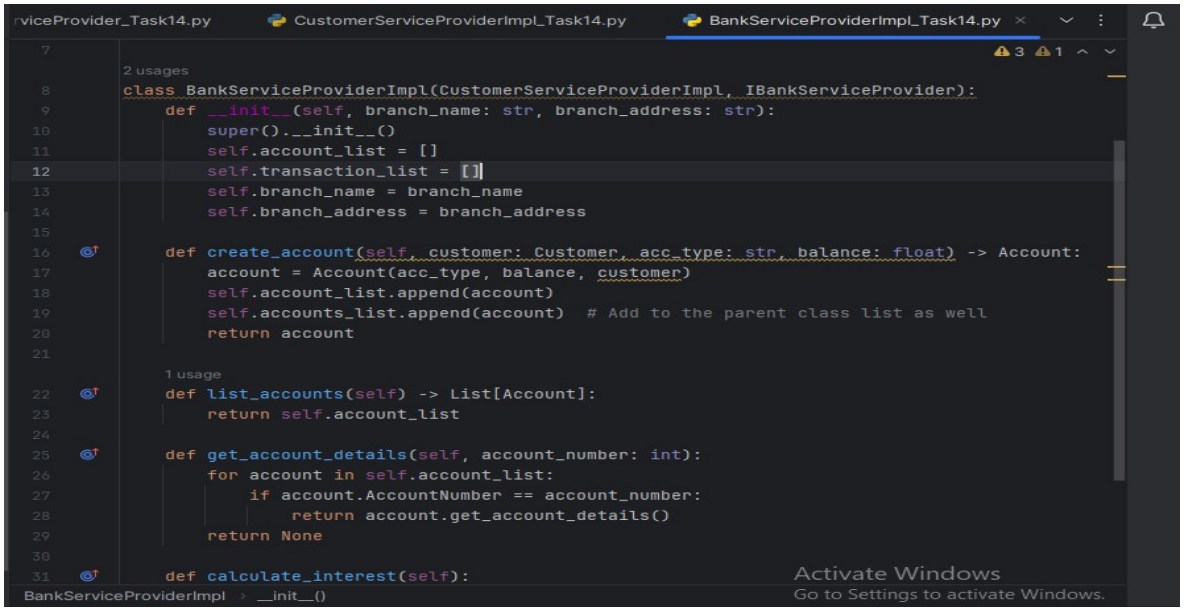
7. Create CustomerServiceProviderImpl class which implements ICustomerServiceProvider provide all implementation methods.

These methods do not interact with database directly.



8. Create BankServiceProviderImpl class which inherits from CustomerServiceProviderImpl and implements IBankServiceProvider.

• Attributes o accountList: List of Accounts to store any account objects. o transactionList: List of Transaction to store transaction objects. o branchName and branchAddress as String objects



9. Create IBankRepository interface/abstract class which include following methods to interact with database.

• createAccount(customer: Customer, accNo: long, accType: String, balance: float): Create a new bank account for the given customer with the initial balance and store in database.

• listAccounts(): List accountsList: List all accounts in the bank from database. • calculateInterest(): the calculate_interest() method to calculate interest based on the balance and interest rate.

• getAccountBalance(account_number: long): Retrieve the balance of an account given its account number. should return the current balance of account from database.

• deposit(account_number: long, amount: float): Deposit the specified amount into the account. Should update new balance in database and return the new balance.

• withdraw(account_number: long, amount: float): Withdraw amount should check the balance from account in database and new balance should updated in Database. o A savings account should maintain a minimum balance and checking if the withdrawal violates the minimum

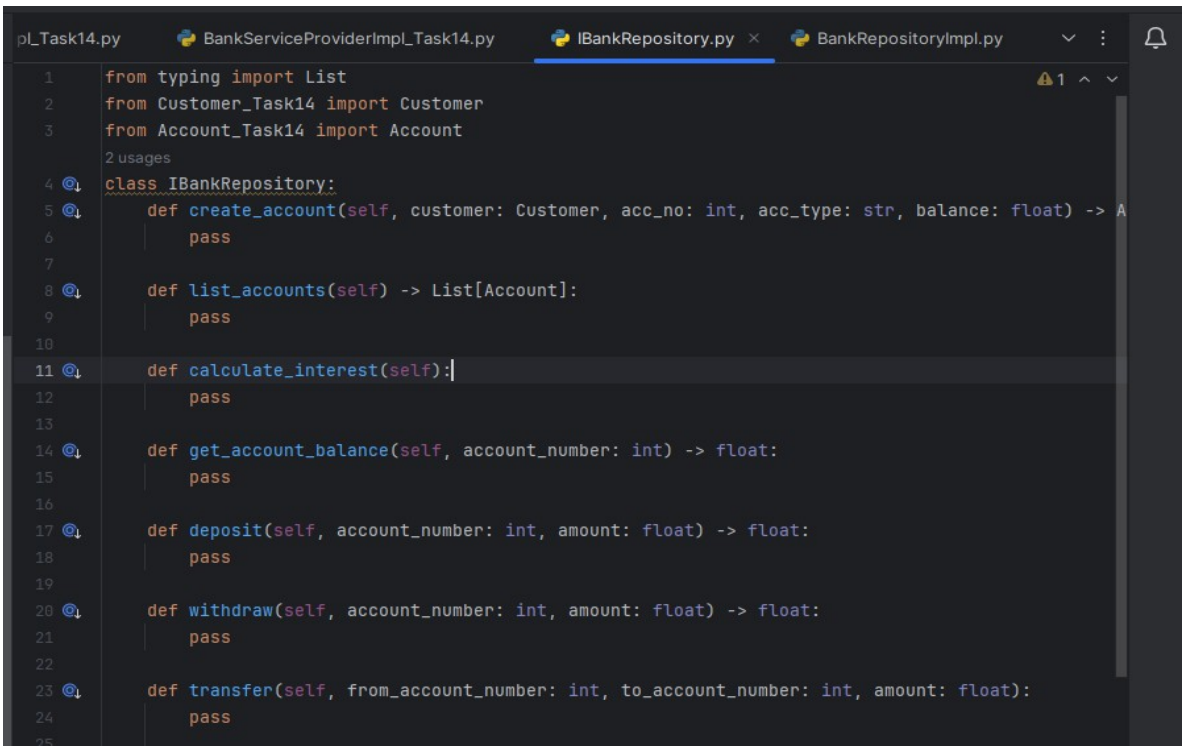
balance rule.

o Current account customers are allowed withdraw overdraftLimit and available account balance. withdraw limit can exceed the available balance and should not exceed the overdraft limit.

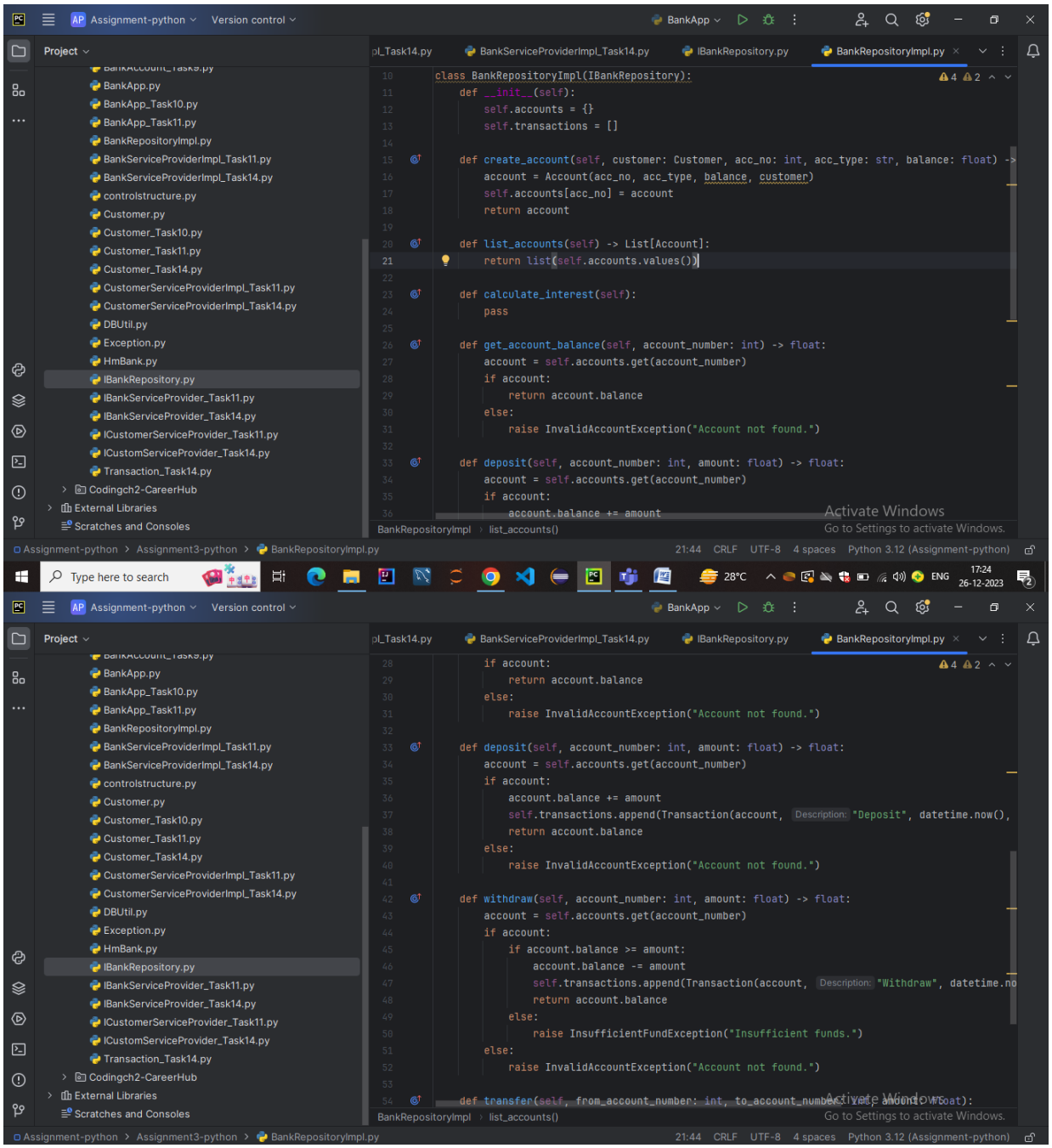
• transfer(from_account_number: long, to_account_number: int, amount: float): Transfer money from one account to another. check the balance from account in database and new balance should updated in Database.

• getAccountDetails(account_number: long): Should return the account and customer details from databse.

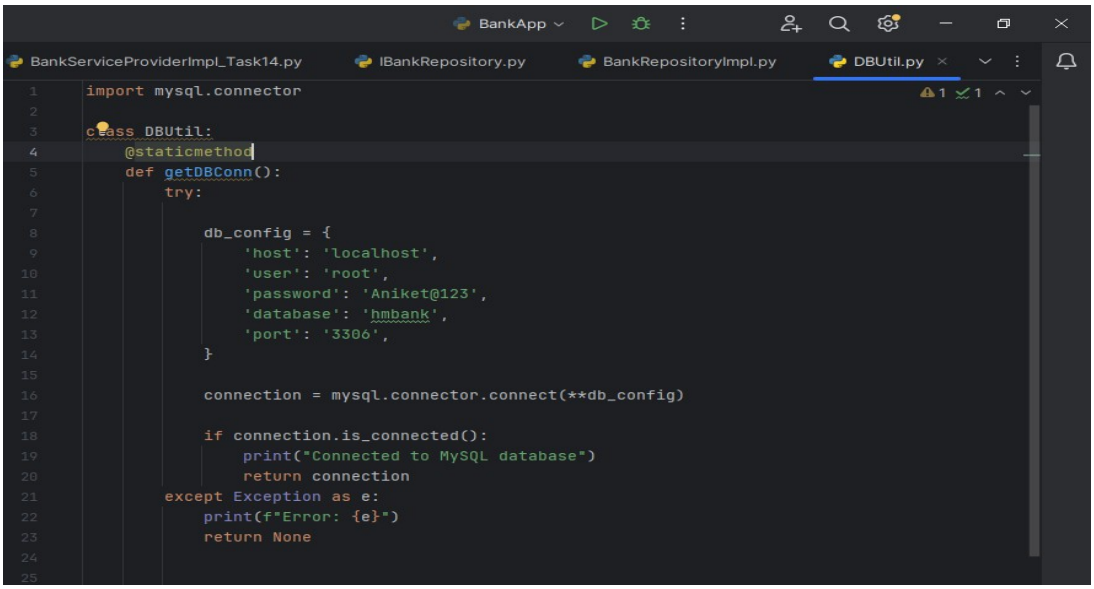
• getTransations(account_number: long, FromDate:Date, ToDate: Date): Should return the list of transaction between two dates from database



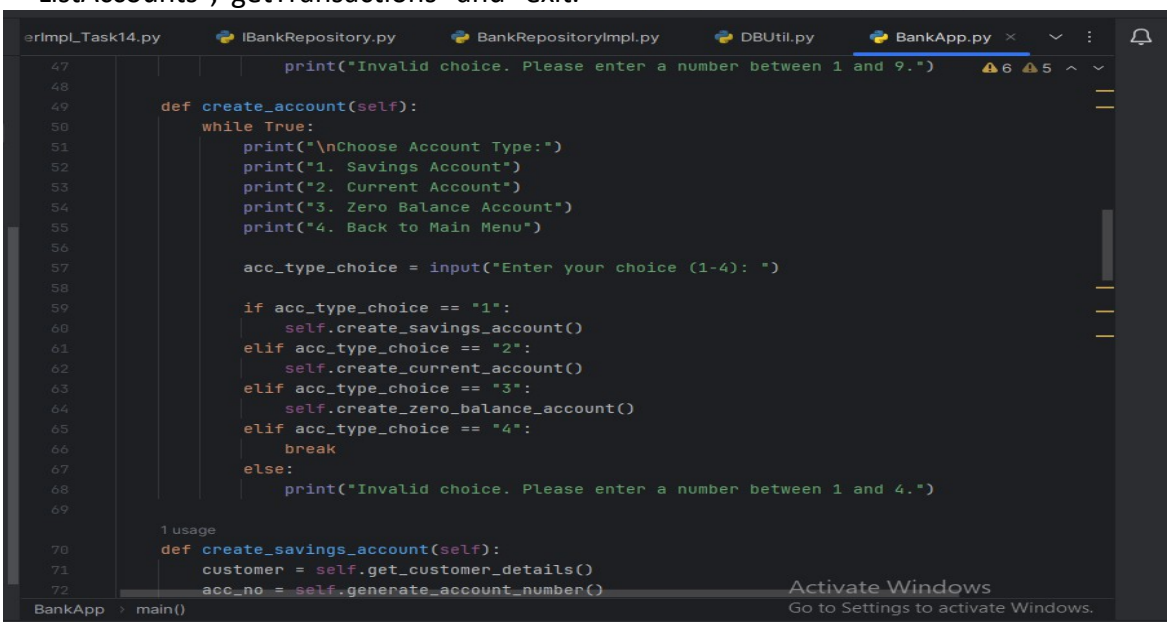
10. Create BankRepositoryImpl class which implement the IBankRepository interface/abstract class and provide implementation of all methods and perform the database operations

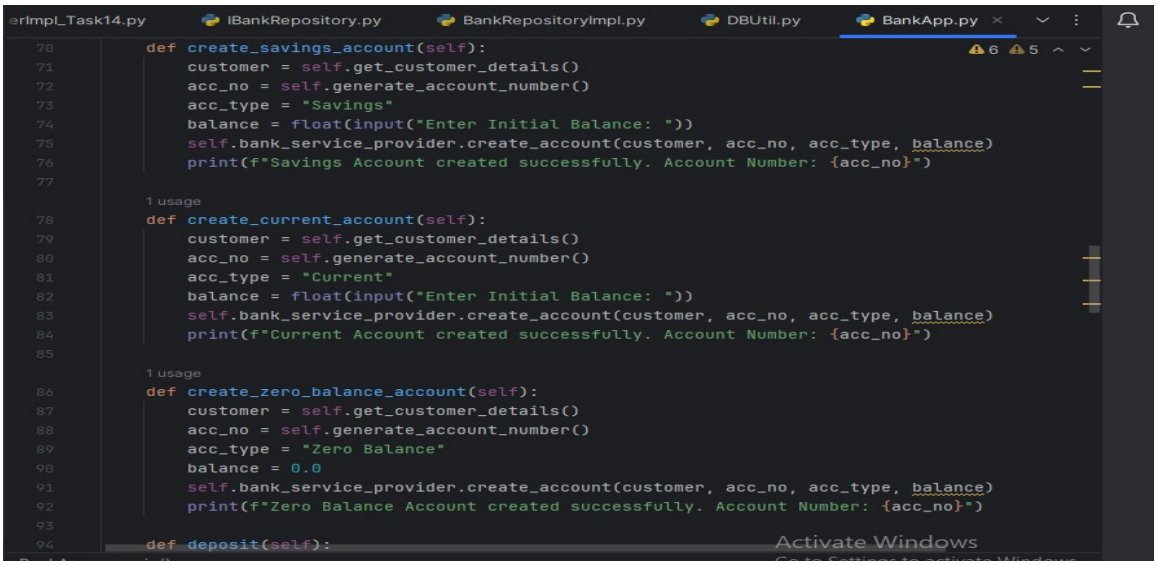


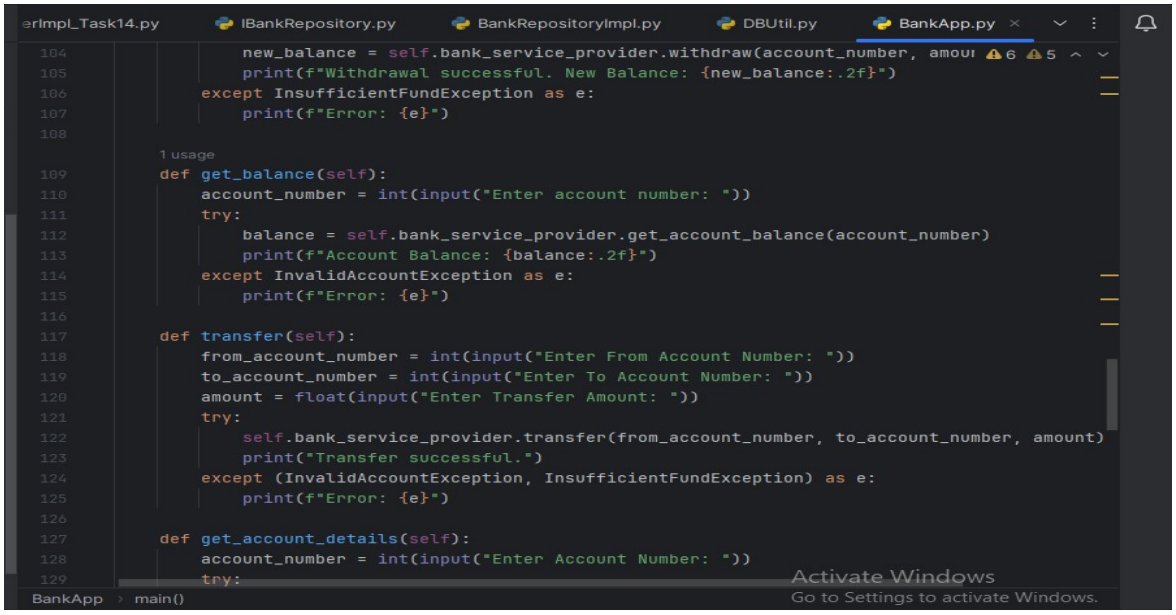
11. Create DBUtil class and add the following method. • static getDBConn():Connection Establish a connection to the database and return Connection reference.

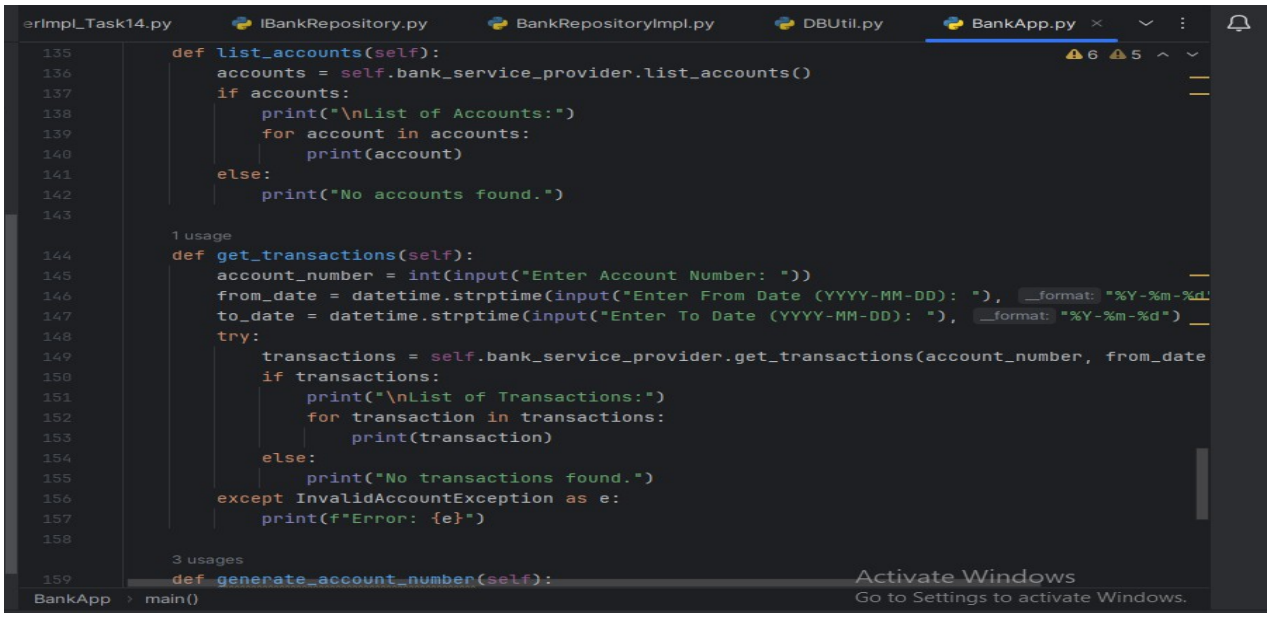


Create BankApp class and perform following operation: • main method to simulate the banking system. Allow the user to interact with the system by entering choice from menu such as "create_account", "deposit","withdraw", "get_balance", "transfer", "getAccountDetails", "ListAccounts","getTransactions" and "exit."




erImpl_Task14.py
IBankRepository.py
BankRepositoryImpl.py
DBUtil.py
BankApp.py
def create_savings_account(self):
customer = self.get_customer_details()
acc_no = self.generate_account_number()
acc_type = "Savings"
balance = float(input("Enter Initial Balance: "))
self.bank_service_provider.create_account(customer, acc_no, acc_type, balance)
print(f"Savings Account created successfully. Account Number: {acc_no}")
1 usage
def create_current_account(self):
customer = self.get_customer_details()
acc_no = self.generate_account_number()
acc_type = "Current"
balance = float(input("Enter Initial Balance: "))
self.bank_service_provider.create_account(customer, acc_no, acc_type, balance)
print(f"Current Account created successfully. Account Number: {acc_no}")
1 usage
def create_zero_balance_account(self):
customer = self.get_customer_details()
acc_no = self.generate_account_number()
acc_type = "Zero Balance"
balance = 0.0
self.bank_service_provider.create_account(customer, acc_no, acc_type, balance)
print(f"Zero Balance Account created successfully. Account Number: {acc_no}")
def deposit(self):
Activate Windows


erImpl_Task14.py
IBankRepository.py
BankRepositoryImpl.py
DBUtil.py
BankApp.py
new_balance = self.bank_service_provider.withdraw(account_number, amou
print(f"Withdrawal successful. New Balance: {new_balance:.2f}")
except InsufficientFundException as e:
print(f"Error: {e}")
1 usage
def get_balance(self):
account_number = int(input("Enter account number: "))
try:
balance = self.bank_service_provider.get_account_balance(account_number)
print(f"Account Balance: {balance:.2f}")
except InvalidAccountException as e:
print(f"Error: {e}")
def transfer(self):
from_account_number = int(input("Enter From Account Number: "))
to_account_number = int(input("Enter To Account Number: "))
amount = float(input("Enter Transfer Amount: "))
try:
self.bank_service_provider.transfer(from_account_number, to_account_number, amount)
print("Transfer successful.")
except (InvalidAccountException, InsufficientFundException) as e:
print(f"Error: {e}")
def get_account_details(self):
account_number = int(input("Enter Account Number: "))
try:
BankApp › main()
Activate Windows
Go to Settings to activate Windows.



• create_account should display sub menu to choose type of accounts and repeat this operation until user exit.