

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

```
1  def check_loan_eligibility(credit_score, annual_income):  
2      if credit_score > 700 and annual_income >= 50000:  
3          return True  
4      else:  
5          return False  
6  usage:  
7  def main():  
8      credit_score = int(input("Enter your credit score: "))  
9      annual_income = float(input("Enter your annual income: $"))  
10     eligibility = check_loan_eligibility(credit_score, annual_income)  
11     if eligibility:  
12         print("Congratulations! You are eligible for a loan.")  
13     else:  
14         print("Sorry, you are not eligible for a loan at this time.")  
15  
16  if __name__ == "__main__":  
17      main()
```

```
↓ Enter your credit score: 725  
→ Enter your annual income: $100000  
← Congratulations! You are eligible for a loan.  
↑ Process finished with exit code 0
```

Activate Wi

Task 2: Nested Conditional Statement

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

```
8o 19 #TASK2
...
20     1 usage
21     def check_balance(balance):
22         print(f"Your current balance: ${balance}")
23
24     1 usage
25     def withdraw(balance, amount):
26         if amount > balance:
27             print("Insufficient funds. Withdrawal failed.")
28         elif amount % 100 != 0 or amount <= 0:
29             print("Invalid withdrawal amount. Please enter a multiple of 100 and greater than 0.")
30         else:
31             balance -= amount
32             print(f"Withdrawal successful. Remaining balance: ${balance}")
33             return balance
34
35     1 usage
36     def deposit(balance, amount):
37         if amount <= 0:
38             print("Invalid deposit amount. Please enter an amount greater than 0.")
39         else:
40             balance += amount
41             print(f"Deposit successful. New balance: ${balance}")
42             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

```
...
41     def main():
42
43         current_balance = float(input("Enter your current balance: $"))
44
45         print("\nATM Options:")
46         print("1. Check Balance")
47         print("2. Withdraw")
48         print("3. Deposit")
49
50         # Get user choice
51         choice = int(input("Enter your choice (1, 2, or 3): "))
52
53         if choice == 1:
54             check_balance(current_balance)
55         elif choice == 2:
56             withdrawal_amount = float(input("Enter the amount to withdraw: $"))
57             current_balance = withdraw(current_balance, withdrawal_amount)
58         elif choice == 3:
59             deposit_amount = float(input("Enter the amount to deposit: $"))
60             current_balance = deposit(current_balance, deposit_amount)
61         else:
62             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: $\text{future_balance} = \text{initial_balance} * (1 + \text{annual_interest_rate}/100)^{\text{years}}$.

```
... 63
64 #TASK_3
65 1 usage
66 def calculate_future_balance(initial_balance, annual_interest_rate, years):
67     future_balance = initial_balance * (1 + annual_interest_rate / 100) ** years
68     return future_balance
69
70 1 usage
71 def main2():
72     num_customers = int(input("Enter the number of customers: "))
73
74     for customer in range(1, num_customers + 1):
75         print(f"\nCustomer {customer}:")
76
77         initial_balance = float(input("Enter the initial balance: $"))
78         annual_interest_rate = float(input("Enter the annual interest rate (%): "))
79         years = int(input("Enter the number of years: "))
80
81         future_balance = calculate_future_balance(initial_balance, annual_interest_rate, years)
82
83         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

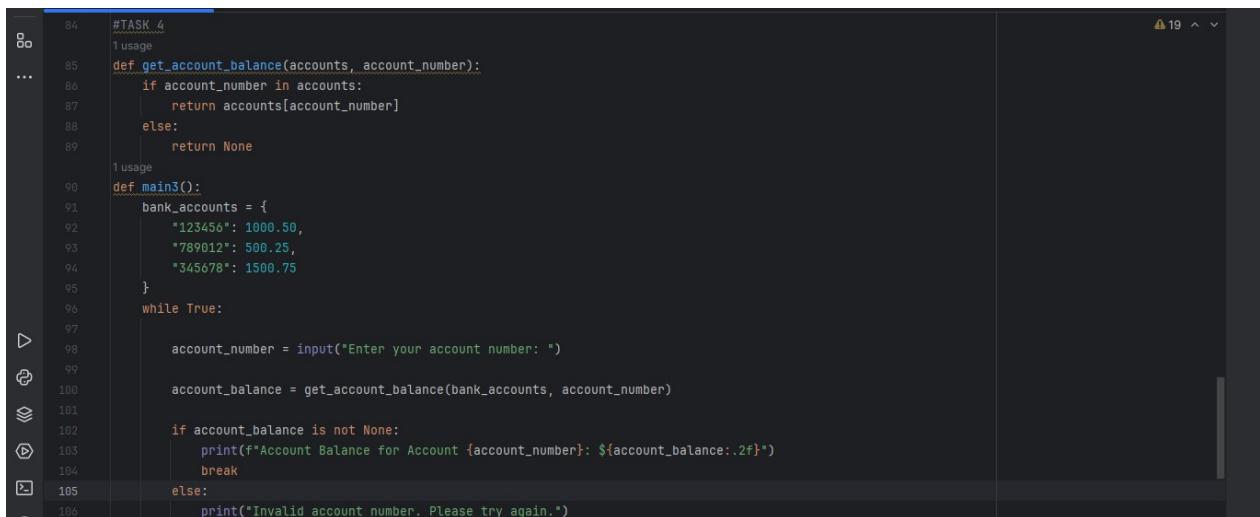
Process finished with exit code 0
```

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.

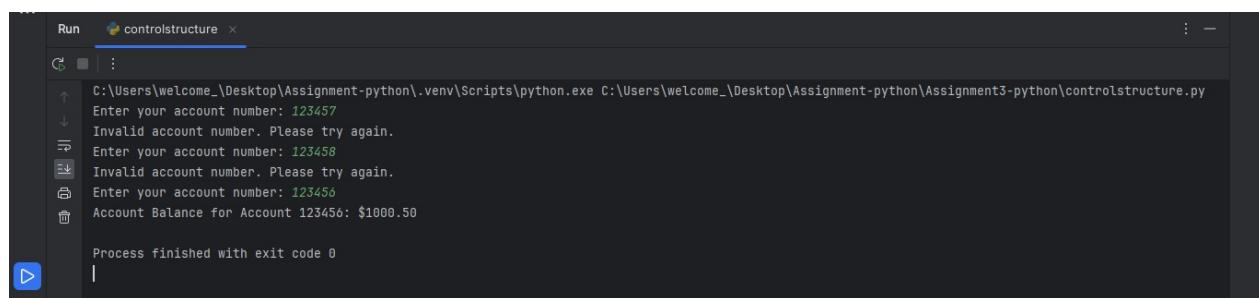
Tasks:

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.



```
#TASK_4
1 usage
...
85     def get_account_balance(accounts_, account_number):
86         if account_number in accounts_:
87             return accounts_[account_number]
88         else:
89             return None
1 usage
90     def main3():
91         bank_accounts = {
92             "123456": 1000.50,
93             "789012": 500.25,
94             "345678": 1500.75
95         }
96         while True:
97
98             account_number = input("Enter your account number: ")
99
100            account_balance = get_account_balance(bank_accounts, account_number)
101
102            if account_balance is not None:
103                print(f"Account Balance for Account {account_number}: ${account_balance:.2f}")
104                break
105            else:
106                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



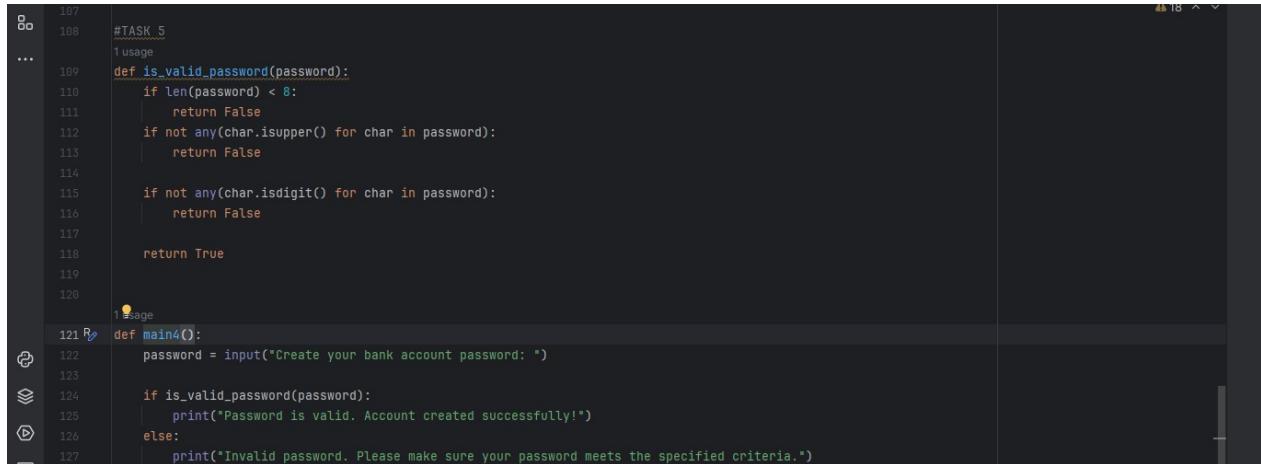
```
Run controlstructure x
C:\Users\welcome_\Desktop\Assignment-python\.venv\Scripts\python.exe C:\Users\welcome_\Desktop\Assignment-python\Assignment3-python\controlstructure.py
Enter your account number: 123457
Invalid account number. Please try again.
Enter your account number: 123458
Invalid account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

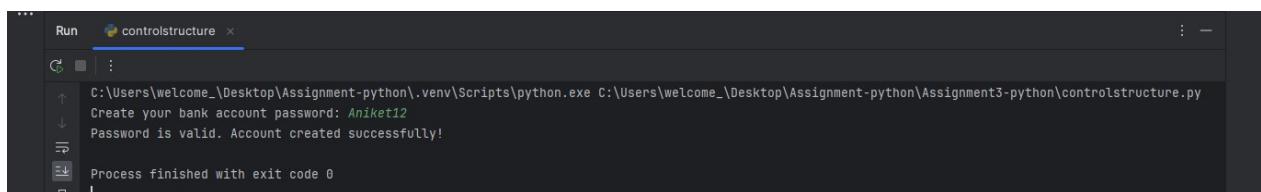
Write a program that prompts the user to create a password for their bank account. Implement if conditions to validate the password according to these rules:

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



```
107
108 #TASK_5
109 ...
110 def is_valid_password(password):
111     if len(password) < 8:
112         return False
113     if not any(char.isupper() for char in password):
114         return False
115     if not any(char.isdigit() for char in password):
116         return False
117
118     return True
119
120
121 R def main():
122     password = input("Create your bank account password: ")
123
124     if is_valid_password(password):
125         print("Password is valid. Account created successfully!")
126     else:
127         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 x
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.

```
130 ...
131     def display_transaction_history(transaction_history):
132         print("\nTransaction History:")
133         for transaction in transaction_history:
134             print(transaction)
135
136     usage
137     def main():
138         transaction_history = []
139         while True:
140             print("\nOptions:")
141             print("1. Add Deposit")
142             print("2. Add Withdrawal")
143             print("3. Exit")
144             choice = input("Enter your choice (1, 2, or 3): ")
145
146             if choice == '1':
147                 deposit_amount = float(input("Enter the deposit amount: $"))
148                 transaction_history.append(f"Deposit: +${deposit_amount:.2f}")
149             elif choice == '2':
150                 withdrawal_amount = float(input("Enter the withdrawal amount: $"))
151                 transaction_history.append(f"Withdrawal: -${withdrawal_amount:.2f}")
152             elif choice == '3':
153                 display_transaction_history(transaction_history)
154                 print("Exiting program. Thank you!")
155                 break
156             else:
157                 print("Invalid choice. Please enter 1, 2, or 3.")
```

Activate Windows

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!
```

Activate Windows
Go to Settings to activate Windows.

OOPS, Collections and Exception Handling

Task 7: Class & Object

1. Create a `Customer` class with the following confidential attributes:

- Attributes
 - Customer ID
 - First Name
 - Last Name
 - Email Address
 - Phone Number
 - Address

```
1  class Customer:
2      def __init__(self, CustomerID, FirstName, LastName, Email, PhoneNumber, Address):
3          self._CustomerID = CustomerID
4          self._FirstName = FirstName
5          self._LastName = LastName
6          self._Email = Email
7          self._PhoneNumber = PhoneNumber
8          self._Address = Address
9
10         1 usage
11         @property
12         def CustomerID(self):
13             return self._CustomerID
14
15         @CustomerID.setter
16         def CustomerID(self, new_CustomerID):
17             if isinstance(new_CustomerID, str) and new_CustomerID:
18                 self._CustomerID = new_CustomerID
19             else:
20                 raise ValueError("Customer ID must be a non-empty string.")
21
22         7 usages (6 dynamic)
23         @property
24         def FirstName(self):
25             return self._FirstName
26
27         6 usages (6 dynamic)
28         @FirstName.setter
29         def FirstName(self, new_FirstName):
30
31         # Getter and setter methods for Last Name
32         5 usages (4 dynamic)
33         @property
34         def LastName(self):
35             return self._LastName
36
37         4 usages (4 dynamic)
38         @LastName.setter
39         def LastName(self, new_LastName):
40             if isinstance(new_LastName, str) and new_LastName:
41                 self._LastName = new_LastName
42             else:
43                 raise ValueError("Last Name must be a non-empty string.")
44
45         1 usage
46         @property
47         def Email(self):
48             return self._Email
```

- Constructor and Methods

```
25         @FirstName.setter
26         def FirstName(self, new_FirstName):
27             if isinstance(new_FirstName, str) and new_FirstName:
28                 self._FirstName = new_FirstName
29             else:
30                 raise ValueError("First Name must be a non-empty string.")
31
32         # Getter and setter methods for Last Name
33         5 usages (4 dynamic)
34         @property
35         def LastName(self):
36             return self._LastName
37
38         4 usages (4 dynamic)
39         @LastName.setter
40         def LastName(self, new_LastName):
41             if isinstance(new_LastName, str) and new_LastName:
42                 self._LastName = new_LastName
43             else:
44                 raise ValueError("Last Name must be a non-empty string.")
45
46         1 usage
47         @property
48         def Email(self):
49             return self._Email
```

- 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
2 @property
3 def PhoneNumber(self):
4     return self._PhoneNumber
5
6
7 @PhoneNumber.setter
8 def PhoneNumber(self, new_PhoneNumber):
9     if isinstance(new_PhoneNumber, str) and new_PhoneNumber.isdigit():
10         self._PhoneNumber = new_PhoneNumber
11     else:
12         raise ValueError("Invalid phone number format.")
13
14 usage
15 @property
16 def Address(self):
17     return self._Address
18
19 @Address.setter
20 def Address(self, new_Address):
21     if isinstance(new_Address, str) and new_Address:
22         self._Address = new_Address
23     else:
24         raise ValueError("Address must be a non-empty string.")

```

Activate Windows

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

```

1 class Account:
2     def __init__(self, AccountNumber, AccountType, AccountBalance):
3         self._AccountNumber = AccountNumber
4         self._AccountType = AccountType
5         self._AccountBalance = AccountBalance
6
7     1 usage
8     @property
9     def AccountNumber(self):
10         return self._AccountNumber
11
12     @AccountNumber.setter
13     def AccountNumber(self, new_AccountNumber):
14         if isinstance(new_AccountNumber, str) and new_AccountNumber:
15             self._AccountNumber = new_AccountNumber
16         else:
17             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.

```

19     def AccountType(self):
20         return self._AccountType
21
22     @AccountType.setter
23     def AccountType(self, new_AccountType):
24         if isinstance(new_AccountType, str) and new_AccountType:
25             self._AccountType = new_AccountType
26         else:
27             raise ValueError("Account Type must be a non-empty string.")
28
29     1 usage
30     @property
31     def AccountBalance(self):
32         return self._AccountBalance
33
34     @AccountBalance.setter
35     def AccountBalance(self, new_AccountBalance):
36         if isinstance(new_AccountBalance, (int, float)) and new_AccountBalance >= 0:
37             self._AccountBalance = new_AccountBalance
38         else:
39             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%

```

41     def deposit(self, amount):
42         if amount > 0:
43             self._AccountBalance += amount
44             print(f"Deposited ${amount:.2f}. New balance: ${self._AccountBalance:.2f}")
45         else:
46             raise ValueError("Deposit amount must be greater than 0.")
47
48     1 usage (1 dynamic)
49     def withdraw(self, amount):
50         if amount > 0:
51             if amount <= self._AccountBalance:
52                 self._AccountBalance -= amount
53                 print(f"Withdrew ${amount:.2f}. New balance: ${self._AccountBalance:.2f}")
54             else:
55                 print("Insufficient balance. Withdrawal failed.")
56         else:
57             raise ValueError("Withdrawal amount must be greater than 0.")
58
59     1 usage (1 dynamic)
60     def calculate_interest(self):
61         interest_rate = 4.5
62         interest_amount = (interest_rate / 100) * self._AccountBalance
63         print(f"Interest calculated: ${interest_amount:.2f}")
64         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.

```
 1  from Account import Account
 2  usage
 3  class Bank:
 4      def __init__(self):
 5          self.accounts = []
 6
 7      1 usage
 8      def create_account(self, account_number, account_type, initial_balance):
 9          account = Account(account_number, account_type, initial_balance)
10          self.accounts.append(account)
11          print(f"Account created: {account.AccountType} Account ({account.AccountNumber}) with ini
12
13      1 usage
14      def main():
15          bank = Bank()
16
17          bank.create_account( account_number: "AC001", account_type: "Savings", initial_balance: 1000)
18
19          account1 = bank.accounts[0]
20          account1.deposit(500)
21
22          account1.withdraw(200)
23
24          account1.calculate_interest()
25
26  > if __name__ == "__main__":
27      main()
28
29 main()
```

Activate Windows
Go to Settings to activate Windows.

```
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)
41 def deposit(self, amount):
42     if isinstance(amount, (float, int)) and amount > 0:
43         self.account_balance += amount
44         print(f"Deposited ${amount:.2f}. New balance: ${self.account_balance:.2f}")
45     else:
46         raise ValueError("Deposit amount must be a positive number.")
47
48 usage (1 dynamic)
49 def withdraw(self, amount):
50     if isinstance(amount, (float, int)) and amount > 0:
51         if amount <= self.account_balance:
52             self.account_balance -= amount
53             print(f"Withdrew ${amount:.2f}. New balance: ${self.account_balance:.2f}")
54         else:
55             print("Insufficient balance. Withdrawal failed.")
56     else:
57         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).

```

64     class SavingsAccount(Account):
65         def __init__(self, account_number, account_balance, interest_rate):
66             super().__init__(account_number, account_balance)
67             self.interest_rate = interest_rate
68
69             1 usage (1 dynamic)
70             def calculate_interest(self):
71                 interest_amount = (self.interest_rate / 100) * self.account_balance
72                 self.account_balance += interest_amount
73                 print(f"Interest calculated and added: ${interest_amount:.2f}. New balance: ${self.account_balance:.2f}")
74             class CurrentAccount(Account):
75                 OVERDRAFT_LIMIT = 1000
76                 def __init__(self, account_number, account_balance):
77                     super().__init__(account_number, account_balance)
78                     1 usage (1 dynamic)
79                     def withdraw(self, amount):
80                         if amount > 0:
81                             available_balance = self.account_balance + self.OVERDRAFT_LIMIT
82                             if amount <= available_balance:
83                                 self.account_balance -= amount
84                                 print(f"Withdrew ${amount:.2f}. New balance: ${self.account_balance:.2f}")
85                             else:
86                                 print("Withdrawal limit exceeded. Withdrawal failed.")
87                         else:
88                             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.

```

1  from Account import Account,SavingsAccount,CurrentAccount
2  1 usage
3  class Bank:
4      1 usage
5      def create_account(self):
6          print("Select account type:")
7          print("1. Savings Account")
8          print("2. Current Account")
9
10         choice = int(input("Enter your choice (1 or 2): "))
11         account_number = input("Enter account number: ")
12         initial_balance = float(input("Enter initial balance: "))
13
14         if choice == 1:
15             interest_rate = float(input("Enter interest rate for savings account: "))
16             return SavingsAccount(account_number, initial_balance, interest_rate)
17         elif choice == 2:
18             return CurrentAccount(account_number, initial_balance)
19         else:
20             print("Invalid choice. Creating a generic account.")
21             return Account(account_number, AccountType="Generic", initial_balance)
22
23         1 usage
24         def main(self):
25             print("Customer Create Account first follow below steps")
26             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.

```

 25     print("\nSelect operation:")
 26     print("1. Deposit")
 27     print("2. Withdraw")
 28     print("3. Calculate Interest (for Savings Account)")
 29     print("4. Exit")
 30
 31     choice = int(input("Enter your choice (1-4): "))
 32
 33     if choice == 1:
 34         amount = float(input("Enter deposit amount: "))
 35         account.deposit(amount)
 36     elif choice == 2:
 37         amount = float(input("Enter withdrawal amount: "))
 38         account.withdraw(amount)
 39     elif choice == 3:
 40         if isinstance(account, SavingsAccount):
 41             account.calculate_interest()
 42         else:
 43             print("Interest calculation not applicable for the current account.")
 44     elif choice == 4:
 45         print("Exiting the program.")
 46         break
 47     else:
 48         print("Invalid choice. Please choose a valid option.")
 49
 50 bank = Bank()
 51 bank.main()

```

Activate Windows

```

...
 2. Withdraw
 3. Calculate Interest (for Savings Account)
 4. Exit
 Enter your choice (1-4): 2
 Enter withdrawal amount: 200
 Withdraw $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 Account 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

```

Activate Windows

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

```
1  from abc import ABC, abstractmethod
2
3  class BankAccount(ABC):
4      def __init__(self, AccountNumber, CustomerName, Balance):
5          self._Accountnumber = AccountNumber
6          self._Customername = CustomerName
7          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.

```
9     @property
10    def AccountNumber(self):
11        return self._AccountNumber
12
13    @AccountNumber.setter
14    def AccountNumber(self, new_AccountNumber):
15        if isinstance(new_AccountNumber, str) and new_AccountNumber:
16            self._AccountNumber = new_AccountNumber
17        else:
18            raise ValueError("Account number must be a non-empty string.")
19
20    @property
21    def CustomerName(self):
22        return self._CustomerName
23
24    @CustomerName.setter
25    def CustomerName(self, new_CustomerName):
26        if isinstance(new_CustomerName, str) and new_CustomerName:
27            self._CustomerName = new_CustomerName
28        else:
29            raise ValueError("Customer name must be a non-empty string.")
30
31    @property
32    def Balance(self):
33        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.

```
41    1 usage (1 dynamic)
42    @abstractmethod
43    def deposit(self, amount):
44        pass
45
46    1 usage (1 dynamic)
47    @abstractmethod
48    def withdraw(self, amount):
49        pass
50
51    1 usage (1 dynamic)
52    @abstractmethod
53    def calculate_interest(self):
54        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

```
52     class SavingsAccount(BankAccount):
53     ...
54         def __init__(self, account_number="", customer_name="", balance=0.0, interest_rate=0.0):
55             super().__init__(account_number, customer_name, balance)
56             self._interest_rate = interest_rate
57
58             1 usage
59             @property
60                 def interest_rate(self):
61                     return self._interest_rate
62
63             @interest_rate.setter
64                 def interest_rate(self, new_interest_rate):
65                     if isinstance(new_interest_rate, (int, float)) and 0 <= new_interest_rate <= 100:
66                         self._interest_rate = new_interest_rate
67                     else:
68                         raise ValueError("Interest rate must be a number between 0 and 100.")
69
70             1 usage (1 dynamic)
71             def deposit(self, amount):
72                 if isinstance(amount, (float, int)) and amount > 0:
73                     self._balance += amount
74                     print(f"Deposited ${amount:.2f}. New balance: ${self._balance:.2f}")
75                 else:
76                     raise ValueError("Deposit amount must be a positive number.")
77
78             68 ⚡
79             def deposit(self, amount):
80                 if isinstance(amount, (float, int)) and amount > 0:
81                     self._balance += amount
82                     print(f"Deposited ${amount:.2f}. New balance: ${self._balance:.2f}")
83                 else:
84                     raise ValueError("Deposit amount must be a positive number.")
85
86             1 usage (1 dynamic)
87             def withdraw(self, amount):
88                 if isinstance(amount, (float, int)) and amount > 0:
89                     if amount <= self._balance:
90                         self._balance -= amount
91                         print(f"Withdraw ${amount:.2f}. New balance: ${self._balance:.2f}")
92                     else:
93                         print("Insufficient balance. Withdrawal failed.")
94                 else:
95                     raise ValueError("Withdrawal amount must be a positive number.")
96
97             1 usage (1 dynamic)
98             def calculate_interest(self):
99                 interest_amount = (self._interest_rate / 100) * self._balance
100                self._balance += interest_amount
101                print(f"Interest calculated and added: ${interest_amount:.2f}. New balance: ${self._balance:.2f}")
102
103 ⚡
104             def withdraw(self, amount):
105                 if isinstance(amount, (float, int)) and amount > 0:
106                     available_balance = self._balance + self.OVERDRAFT_LIMIT
107                     if amount <= available_balance:
108                         self._balance -= amount
109                         print(f"Withdraw ${amount:.2f}. New balance: ${self._balance:.2f}")
110                     else:
111                         print("Withdrawal limit exceeded. Withdrawal failed.")
112                 else:
113                     raise ValueError("Withdrawal amount must be a positive number.")
```

- 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).

```
90     class CurrentAccount(BankAccount):
91     ...
92         OVERDRAFT_LIMIT = 1000
93
94         def __init__(self, account_number="", customer_name="", balance=0.0):
95             super().__init__(account_number, customer_name, balance)
96
97             1 usage (1 dynamic)
98             def deposit(self, amount):
99                 if isinstance(amount, (float, int)) and amount > 0:
100                     self._balance += amount
101                     print(f"Deposited ${amount:.2f}. New balance: ${self._balance:.2f}")
102                 else:
103                     raise ValueError("Deposit amount must be a positive number.")
104
105             1 usage (1 dynamic)
106             def withdraw(self, amount):
107                 if isinstance(amount, (float, int)) and amount > 0:
108                     available_balance = self._balance + self.OVERDRAFT_LIMIT
109                     if amount <= available_balance:
110                         self._balance -= amount
111                         print(f"Withdraw ${amount:.2f}. New balance: ${self._balance:.2f}")
112                     else:
113                         print("Withdrawal limit exceeded. Withdrawal failed.")
114                 else:
115                     raise ValueError("Withdrawal amount must be a positive number.")
```

Activate Windows

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();

```
1 from BankAccount import SavingsAccount, CurrentAccount
2
3 class Bank:
4     1 usage
5     def main(self):
6         while True:
7             print("1. Create Savings Account")
8             print("2. Create Current Account")
9             print("3. Exit")
10
11            choice = input("Enter your choice (1, 2, or 3): ")
12
13            if choice == '1':
14                account_type = "Savings"
15            elif choice == '2':
16                account_type = "Current"
17            elif choice == '3':
18                print("Exiting the program.")
19                break
20            else:
21                print("Invalid choice. Please enter 1, 2, or 3.")
22                continue
23
24            account = self.create_account(account_type)
25            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

```
1
2     def create_account(self, account_type):
3         account_number = input("Enter account number: ")
4         customer_name = input("Enter customer name: ")
5         initial_balance = float(input("Enter initial balance: "))
6
7         if account_type == "Savings":
8             interest_rate = float(input("Enter interest rate for savings account: "))
9             return SavingsAccount(account_number, customer_name, initial_balance, interest_rate)
10            elif account_type == "Current":
11                return CurrentAccount(account_number, customer_name, initial_balance)
12            else:
13                print("Invalid account type.")
14                return None
15
16
17     def perform_operations(self, account):
18         while True:
19             print("\n1. Deposit")
20             print("2. Withdraw")
21             print("3. Calculate Interest (for Savings Account)")
22             print("4. Exit")
23
24             choice = input("Enter your choice (1, 2, 3, or 4): ")
```

```

48     if choice == '1':
49         amount = float(input("Enter deposit amount: "))
50         account.deposit(amount)
51     elif choice == '2':
52         amount = float(input("Enter withdrawal amount: "))
53         account.withdraw(amount)
54     elif choice == '3' and isinstance(account, SavingsAccount):
55         account.calculate_interest()
56     elif choice == '4':
57         print("Exiting account operations.")
58         break
59     else:
60         print("Invalid choice. Please enter 1, 2, 3, or 4.")
61         continue
62
63 bank = Bank()
64 bank.main()

```

Activate Windows

OUTPUT FOR SAVING ACCOUNT

```

↓ 1. Create Savings Account
  ↵ 2. Create Current Account
  ↵ 3. Exit
  ↵ Enter your choice (1, 2, or 3): 1
  ↵ Enter account number: 101
  ↵ Enter customer name: Aniket
  ↵ Enter initial balance: 10000
  ↵ Enter interest rate for savings account: 4

  1. Deposit
  2. Withdraw
  3. Calculate Interest (for Savings Account)
  4. Exit
  ↵ Enter your choice (1, 2, 3, or 4): 2
  ↵ Enter withdrawal amount: 3000
  ↵ Withdraw $3000.00. New balance: $7000.00

  1. Deposit
  2. Withdraw
  3. Calculate Interest (for Savings Account)
  4. Exit
  ↵ Enter your choice (1, 2, 3, or 4): 4

```

OUTPUT FOR CURRENT ACCOUNT

```

...
↑ Enter your choice (1, 2, 3, or 4): 4
Exiting account operations.
  1. Create Savings Account
  2. Create Current Account
  3. Exit
  ↵ Enter your choice (1, 2, or 3): 2
  ↵ Enter account number: 102
  ↵ Enter customer name: Biyani
  ↵ Enter initial balance: 1000000

  1. Deposit
  2. Withdraw
  3. Calculate Interest (for Savings Account)
  4. Exit
  ↵ Enter your choice (1, 2, 3, or 4): 1
  ↵ Enter deposit amount: 20000
  ↵ Deposited $20000.00. New balance: $1020000.00

  1. Deposit
  2. Withdraw
  3. Calculate Interest (for Savings Account)
  4. Exit
  ↵ Enter your choice (1, 2, 3, or 4): 4
Exiting account operations.
  1. Create Savings Account

```

Activate Windows
Go to Settings to activate Windows.

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

```
 1 import re
 2 usages
 3 class Customer:
 4     def __init__(self, customer_id="", first_name="", last_name="", email="", phone_number="", address=""):
 5         self._CustomerID = customer_id
 6         self._FirstName = first_name
 7         self._LastName = last_name
 8         self._Email = email
 9         self._PhoneNumber = phone_number
10         self._Address = address
11
12     @usage
13     @property
14     def CustomerID(self):
15         return self._CustomerID
16
17     @CustomerID.setter
18     def CustomerID(self, new_customer_id):
19         if isinstance(new_customer_id, str) and new_customer_id:
20             self._CustomerID = new_customer_id
21         else:
22             raise ValueError("Customer ID must be a non-empty string.")
23
24     7 usages (6 dynamic)
25     @property
26     def FirstName(self):
```

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

```
40         self._LastName = new_last_name
41     else:
42         raise ValueError("Last Name must be a non-empty string.")
43
44     1 usage
45     @property
46     def Email(self):
47         return self._Email
48
49     @Email.setter
50     def Email(self, new_email):
51         if isinstance(new_email, str) and re.match(pattern: r"^[^@]+@[^@]+\.[^@]+", new_email):
52             self._Email = new_email
53         else:
54             raise ValueError("Invalid Email format.")
55
56     1 usage
57     @property
58     def PhoneNumber(self):
59         return self._PhoneNumber
60
61     @PhoneNumber.setter
62     def PhoneNumber(self, new_phone_number):
63         if isinstance(new_phone_number, str) and re.match(pattern: r"\d{10}", new_phone_number):
64             self._PhoneNumber = new_phone_number
65         else:
66             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)

```
1 from Customer_Task10 import Customer
2 class Account:
3     def __init__(self, account_number="", account_type="", account_balance=0.0, customer=None):
4         self._AccountNumber = account_number
5         self._AccountType = account_type
6         self._AccountBalance = account_balance
7         self._Customer = customer
8
9     1 usage
10    @property
11    def AccountNumber(self):
12        return self._AccountNumber
13
14    @AccountNumber.setter
15    def AccountNumber(self, new_account_number):
16        if isinstance(new_account_number, str) and new_account_number:
17            self._AccountNumber = new_account_number
18        else:
19            raise ValueError("Account Number must be a non-empty string.")
20
21    1 usage
22    @property
23    def AccountType(self):
24        return self._AccountType
25
26    @AccountType.setter
27    def AccountType(self, new_account_type):
28
29    1 usage
30    @property
31    def AccountBalance(self):
32        return self._AccountBalance
33
34    @AccountBalance.setter
35    def AccountBalance(self, new_account_balance):
36        if isinstance(new_account_balance, (int, float)) and new_account_balance >= 0:
37            self._AccountBalance = new_account_balance
38        else:
39            raise ValueError("Account Balance must be a non-negative number.")
40
41    1 usage
42    @property
43    def Customer(self):
44        return self._Customer
45
46    @Customer.setter
47    def Customer(self, new_customer):
48        if isinstance(new_customer, Customer):
49            self._Customer = new_customer
50        else:
51            raise ValueError("Invalid customer object.")
```

Activate Windows
Go to Settings to activate Windows.

- 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.

```
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
```

Activate Windows

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.

```

2   from Account_Task10 import Account
3   class Bank:
4       def __init__(self):
5           self.accounts = {}
6
7       def create_account(self, customer, acc_type, balance):
8           account = Account(customer, acc_type, balance)
9           self.accounts[account.AccountNumber] = account
10      return account
11
12     def get_account_balance(self, account_number):
13         if account_number in self.accounts:
14             return self.accounts[account_number].Balance
15         else:
16             print("Account not found.")
17             return None
18
19     2 usages (2 dynamic)
20     def deposit(self, account_number, amount):
21         if account_number in self.accounts:
22             return self.accounts[account_number].deposit(amount)
23         else:
24             print("Account not found.")
25             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.

```

26     def withdraw(self, account_number, amount):
27         if account_number in self.accounts:
28             return self.accounts[account_number].withdraw(amount)
29         else:
30             print("Account not found.")
31             return None
32
33     1 usage (1 dynamic)
34     def transfer(self, from_account_number, to_account_number, amount):
35         if from_account_number in self.accounts and to_account_number in self.accounts:
36             from_account = self.accounts[from_account_number]
37             to_account = self.accounts[to_account_number]
38             from_account.transfer(to_account, amount)
39         else:
40             print("One or both accounts not found.")
41
42     1 usage (1 dynamic)
43     def get_account_details(self, account_number):
44         if account_number in self.accounts:
45             return self.accounts[account_number].get_account_details()
46         else:
47             print("Account not found.")
48             return None

```

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

```
2 class Account:
3     account_number_counter=1001
4     def __init__(self, account_type="", account_balance=0.0, customer=None):
5         self._AccountNumber = Account.account_number_counter
6         Account.account_number_counter+=1
7         self._AccountType = account_type
8         self._AccountBalance = account_balance
9         self.Customer = customer
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47 bank = Bank()
48 customer = Customer(customer_id="C001", first_name="John", last_name="Doe", email="john@example.com", )
49 account1 = bank.create_account(customer, acc_type="Savings", balance=1000.0)
50 print(f"Account 1 Number: {account1.AccountNumber}")
51
52 account2 = bank.create_account(customer, acc_type="Current", balance=500.0)
53 print(f"Account 2 Number: {account2.AccountNumber}")
54
55
56
```

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.

```
3 usages
2
3 class Account:
4     lastAccNo = 1001
5
6     def __init__(self, account_type="", account_balance=0.0, customer=None):
7         self._AccountNumber = Account.generate_account_number()
8         self._AccountType = account_type
9         self._AccountBalance = account_balance
10        self._Customer = customer
11
12        1 usage (1 dynamic)
13
14 @property
15 def AccountNumber(self):
16     return self._AccountNumber
17
18 @property
19 def AccountType(self):
20     return self._AccountType
21
22 3 usages (2 dynamic)
23
24 @property
25 def AccountBalance(self):
26     return self._AccountBalance
27
28 @property
29 def Customer(self):
30     return self._Customer
```

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

```

27     2 usages (2 dynamic)
28     def deposit(self, amount):
29         if isinstance(amount, (float, int)) and amount > 0:
30             self._AccountBalance += amount
31             print(f"Deposited ${amount:.2f}. New balance: ${self._AccountBalance:.2f}")
32             return self._AccountBalance
33         else:
34             raise ValueError("Deposit amount must be a positive number.")
35
36     2 usages (2 dynamic)
37     def withdraw(self, amount):
38         if amount > 0 and amount <= self._AccountBalance:
39             self._AccountBalance -= amount
40             return self._AccountBalance
41         else:
42             print("Invalid withdrawal amount or insufficient funds.")
43             return None
44
45     1 usage
46     @staticmethod
47     def generate_account_number():
48         Account.lastAccNo += 1
49         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.

```

49     class SavingsAccount(Account):
50         def __init__(self, account_balance=500.0, interest_rate=0.02, customer=None):
51             super().__init__(account_type="Savings", account_balance, customer)
52             self._InterestRate = interest_rate
53
54         @property
55         def InterestRate(self):
56             return self._InterestRate
57
58     class CurrentAccount(Account):
59         def __init__(self, account_balance=0.0, overdraft_limit=1000.0, customer=None):
60             super().__init__(account_type="Current", account_balance, customer)
61             self._OverdraftLimit = overdraft_limit
62
63         @usage
64         @property
65         def OverdraftLimit(self):
66             return self._OverdraftLimit
67
68         2 usages (2 dynamic)
69         def withdraw(self, amount):
70             total_withdrawal = amount + abs(min(0, self._AccountBalance))
71             if total_withdrawal <= (self._AccountBalance + self._OverdraftLimit):
72                 self._AccountBalance -= amount
73                 print(f"Withdrawn ${amount:.2f}. New balance: ${self._AccountBalance:.2f}")
74                 return self._AccountBalance
75             else:
76                 print("Invalid withdrawal amount or exceeding overdraft limit.")
77                 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

```
 1  from abc import ABC, abstractmethod
 2
 3  class ICustomerServiceProvider(ABC):
 4      @abstractmethod
 5      def get_account_balance(self, account_number):
 6          pass
 7
 8      @abstractmethod
 9      def deposit(self, account_number, amount):
10          pass
11
12      @abstractmethod
13      def withdraw(self, account_number, amount):
14          pass
15
16      1 usage (1 dynamic)
17      @abstractmethod
18      def transfer(self, from_account_number, to_account_number, amount):
19          pass
20
21      1 usage (1 dynamic)
22      @abstractmethod
23      def get_account_details(self, account_number):
24          pass
25
```

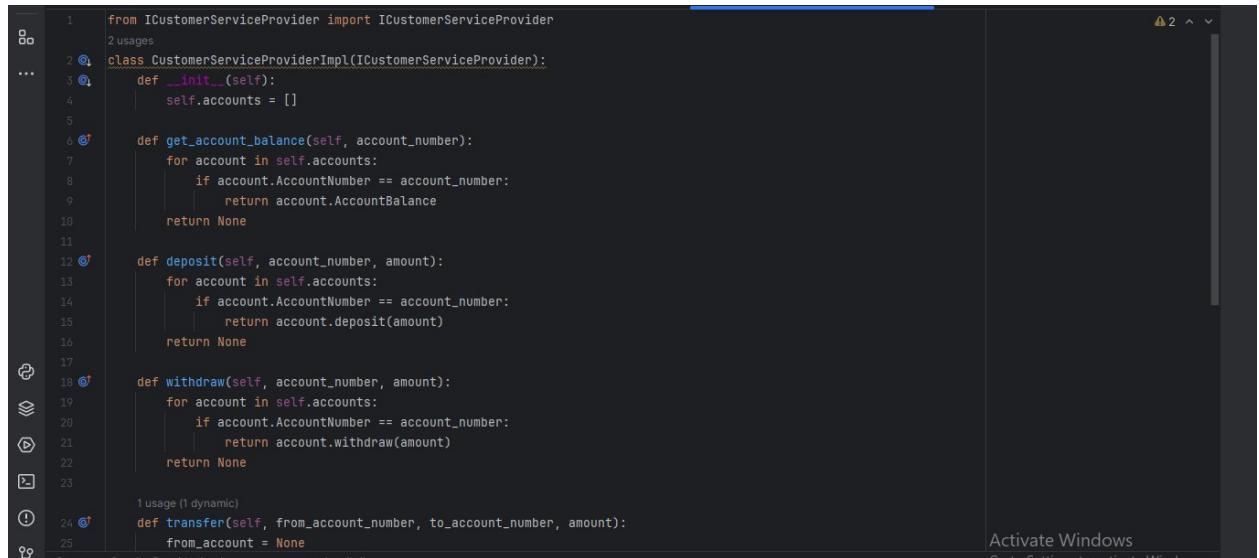
Create IBankServiceProvider interface/abstract class with following functions:

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

```
 1  from abc import ABC, abstractmethod
 2
 3  class IBankServiceProvider(ABC):
 4      @abstractmethod
 5      def create_account(self, customer, acc_type, balance):
 6          pass
 7
 8      @abstractmethod
 9      def list_accounts(self):
10          pass
11
12      1 usage (1 dynamic)
13      @abstractmethod
14      def calculate_interest(self):
15          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.

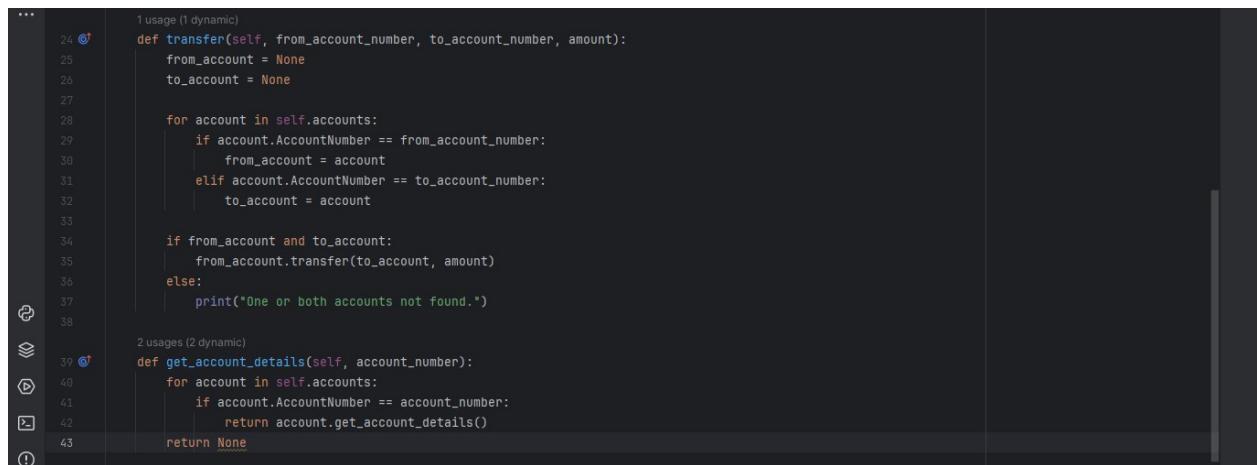


```

1  from ICustomerServiceProvider import ICustomerServiceProvider
2  ...
3  class CustomerServiceProviderImpl(ICustomerServiceProvider):
4      def __init__(self):
5          self.accounts = []
6
7      def get_account_balance(self, account_number):
8          for account in self.accounts:
9              if account.AccountNumber == account_number:
10                  return account.AccountBalance
11
12     def deposit(self, account_number, amount):
13         for account in self.accounts:
14             if account.AccountNumber == account_number:
15                 return account.deposit(amount)
16
17     def withdraw(self, account_number, amount):
18         for account in self.accounts:
19             if account.AccountNumber == account_number:
20                 return account.withdraw(amount)
21
22     def transfer(self, from_account_number, to_account_number, amount):
23         from_account = None
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43

```

Activate Windows
Go to Settings to activate Windows



```

...
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

```

1  from IBankServiceProvider import IBankServiceProvider
2  from CustomerServiceProviderImpl import CustomerServiceProviderImpl
...
4  class BankServiceProviderImpl(CustomerServiceProviderImpl, IBankServiceProvider):
5      def __init__(self, branch_name, branch_address):
6          super().__init__()
7          self.accountList = []
8          self.branchName = branch_name
9          self.branchAddress = branch_address
10
11     def create_account(self, customer, acc_type, balance):
12         account = super().create_account(customer, acc_type, balance)
13         self.accountList.append(account)
14         return account
15
16     def list_accounts(self):
17         return self.accountList
18
19     def calculate_interest(self):
20         for account in self.accountList:
21             pass
22

```

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

- 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.
- InvalidAccountException throw this exception when user entered the invalid account number when tries to transfer amount, get account details classes.
- OverDraftLimitExceededException thow this exception when current account customer try to with draw amount from the current account.

```

1  class InsufficientFundException(Exception):
2      def __init__(self, message="Insufficient funds."):
3          self.message = message
4          super().__init__(self.message)
5
6  class InvalidAccountException(Exception):
7      def __init__(self, message="Invalid account number."):
8          self.message = message
9          super().__init__(self.message)
10
11 class OverDraftLimitExceededException(Exception):
12     def __init__(self, message="Overdraft limit exceeded."):
13         self.message = message
14         super().__init__(self.message)
15

```

Task 14: Database Connectivity.

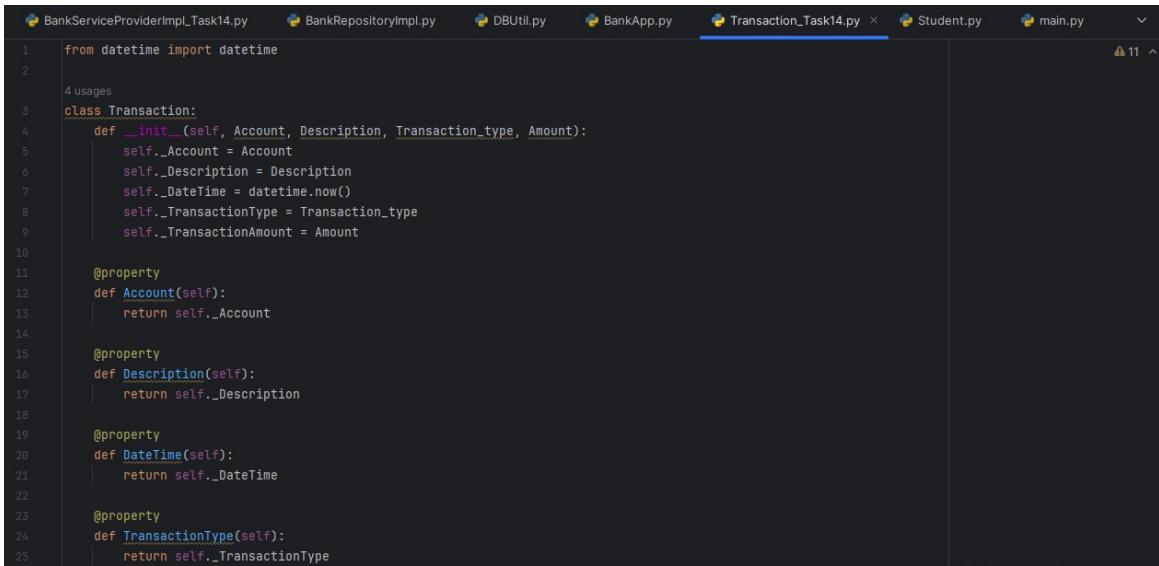
1. Create a 'Customer' class as mentioned above task.
2. Create an class 'Account' that includes the following attributes.

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



The screenshot shows a Python code editor with multiple tabs open. The active tab is 'Transaction_Task14.py'. The code defines a class 'Transaction' with the following properties and methods:

```
1 from datetime import datetime
2
3 class Transaction:
4     def __init__(self, Account, Description, Transaction_type, Amount):
5         self._Account = Account
6         self._Description = Description
7         self._DateTime = datetime.now()
8         self._TransactionType = Transaction_type
9         self._TransactionAmount = Amount
10
11     @property
12     def Account(self):
13         return self._Account
14
15     @property
16     def Description(self):
17         return self._Description
18
19     @property
20     def DateTime(self):
21         return self._DateTime
22
23     @property
24     def TransactionType(self):
25         return self._TransactionType
```

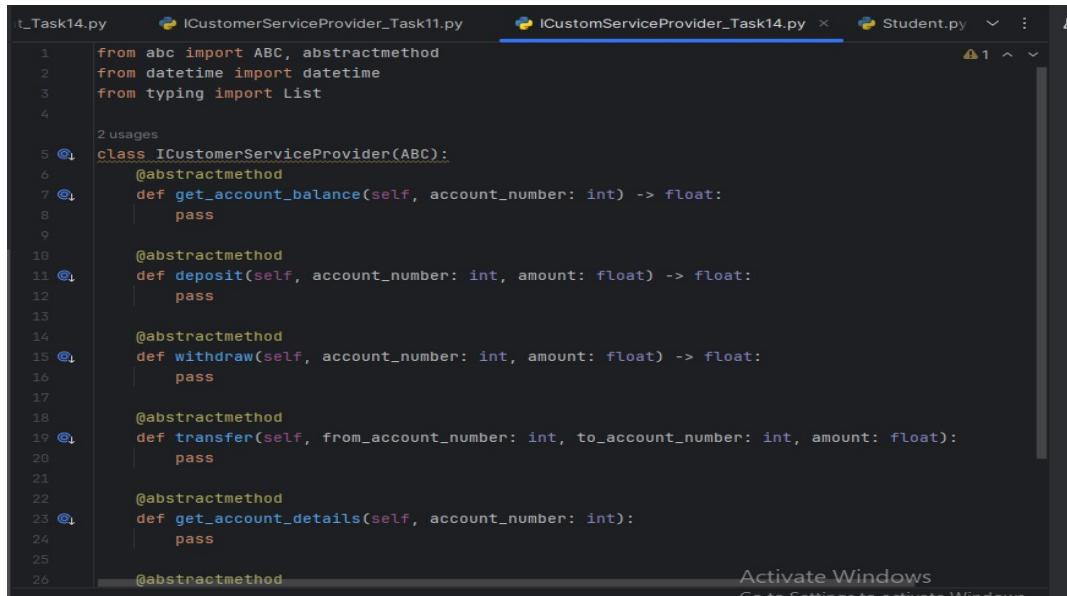
4. Create three child classes that inherit the Account class and each class must contain below mentioned attribute:

- 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.

```
14     return Account.lastAccNo
15
16 class SavingsAccount(Account):
17     def __init__(self, account_type="Savings", account_balance=500.0, customer=None, interest_rate=0.0):
18         super().__init__(account_type, account_balance, customer)
19         self._InterestRate = interest_rate
20
21     @usage
22     @property
23     def InterestRate(self):
24         return self._InterestRate
25
26     @InterestRate.setter
27     def InterestRate(self, interest_rate):
28         if isinstance(interest_rate, (float, int)) and interest_rate >= 0:
29             self._InterestRate = interest_rate
30         else:
31             raise ValueError("Interest rate must be a non-negative number.")
32
33 class CurrentAccount(Account):
34     def __init__(self, account_type="Current", account_balance=0.0, customer=None, overdraft_limit=0.0):
35         super().__init__(account_type, account_balance, customer)
36         self._OverdraftLimit = overdraft_limit
37
38     @usage
39     @property
40     def OverdraftLimit(self):
```

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

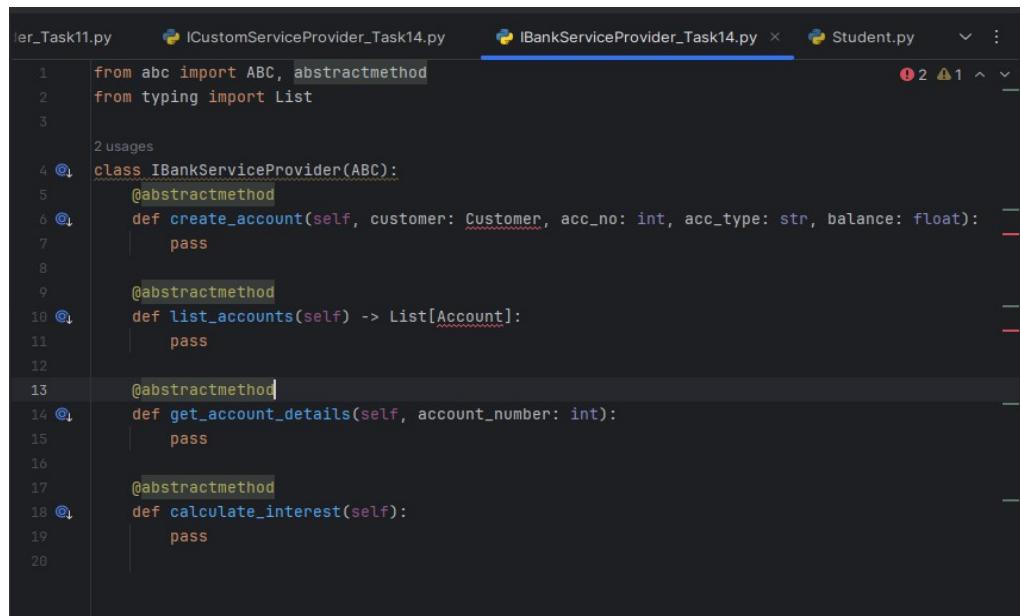
- `get_account_balance(account_number: long)`: Retrieve the balance of an account given its account number. should return the current balance of account.
- `deposit(account_number: long, amount: float)`: Deposit the specified amount into the account. Should return the current balance of account.
- `withdraw(account_number: 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.
- `getTransactions(account_number: long, FromDate: Date, ToDate: Date)`: Should return the list of transaction between two dates.



```
I_Task14.py ICustomerServiceProvider_Task11.py ICustomerServiceProvider_Task14.py Student.py
1 from abc import ABC, abstractmethod
2 from datetime import datetime
3 from typing import List
4
5 2 usages
6 @class ICustomerServiceProvider(ABC):
7     @abstractmethod
8         def get_account_balance(self, account_number: int) -> float:
9             pass
10
11     @abstractmethod
12         def deposit(self, account_number: int, amount: float) -> float:
13             pass
14
15     @abstractmethod
16         def withdraw(self, account_number: int, amount: float) -> float:
17             pass
18
19     @abstractmethod
20         def transfer(self, from_account_number: int, to_account_number: int, amount: float):
21             pass
22
23     @abstractmethod
24         def get_account_details(self, account_number: int):
25             pass
26
27     @abstractmethod
```

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.



```
ler_Task11.py ICustomerServiceProvider_Task11.py IBankServiceProvider_Task14.py Student.py
1 from abc import ABC, abstractmethod
2 from typing import List
3
4 2 usages
5 @class IBankServiceProvider(ABC):
6     @abstractmethod
7         def create_account(self, customer: Customer, acc_no: int, acc_type: str, balance: float):
8             pass
9
10    @abstractmethod
11        def list_accounts(self) -> List[Account]:
12            pass
13
14    @abstractmethod
15        def get_account_details(self, account_number: int):
16            pass
17
18    @abstractmethod
19        def calculate_interest(self):
20            pass
```

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

```
7      2 usages
8      class BankServiceProviderImpl(CustomerServiceProviderImpl, IBankServiceProvider):
9          def __init__(self, branch_name: str, branch_address: str):
10             super().__init__()
11             self.account_list = []
12             self.transaction_list = []
13             self.branch_name = branch_name
14             self.branch_address = branch_address
15
16             def create_account(self, customer: Customer, acc_type: str, balance: float) -> Account:
17                 account = Account(acc_type, balance, customer)
18                 self.account_list.append(account)
19                 self.accounts_list.append(account) # Add to the parent class list as well
20                 return account
21
22             1 usage
23             def list_accounts(self) -> List[Account]:
24                 return self.account_list
25
26             def get_account_details(self, account_number: int):
27                 for account in self.account_list:
28                     if account.AccountNumber == account_number:
29                         return account.get_account_details()
30                 return None
31
32             def calculate_interest(self):
33
BankServiceProviderImpl > __init__()
```

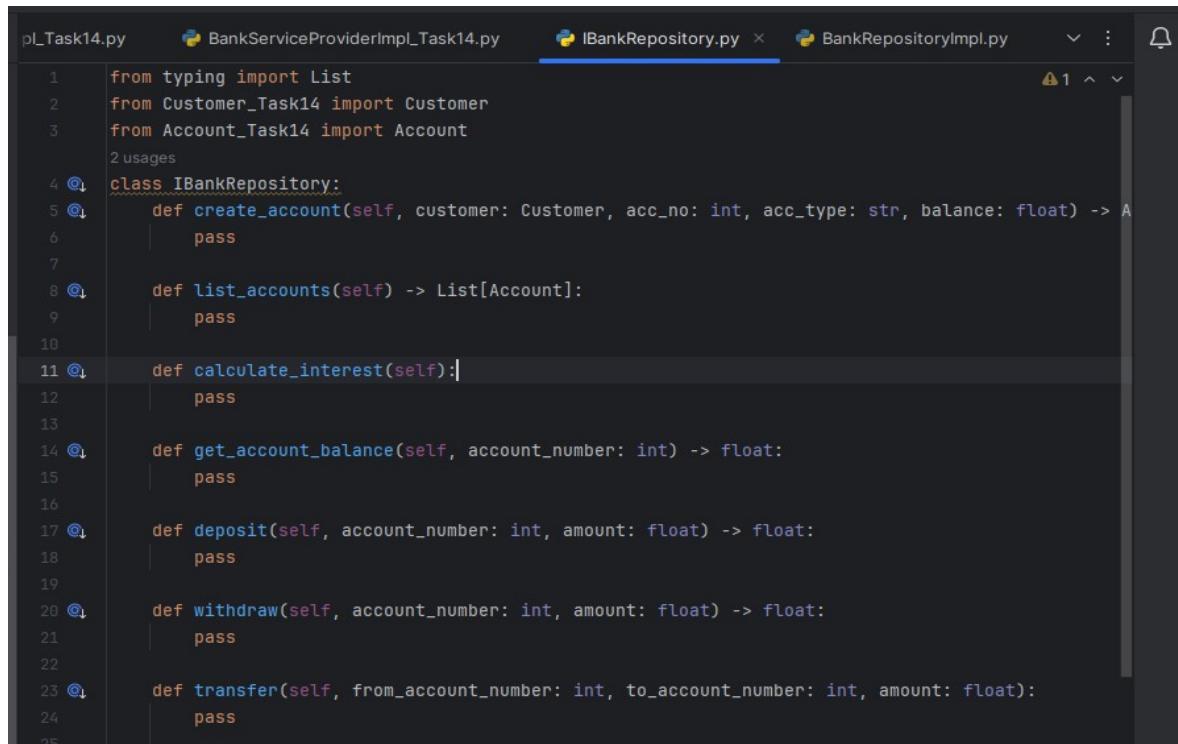
9. Create IBankRepository interface/abstract class which include following methods to interact with database.

- 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.

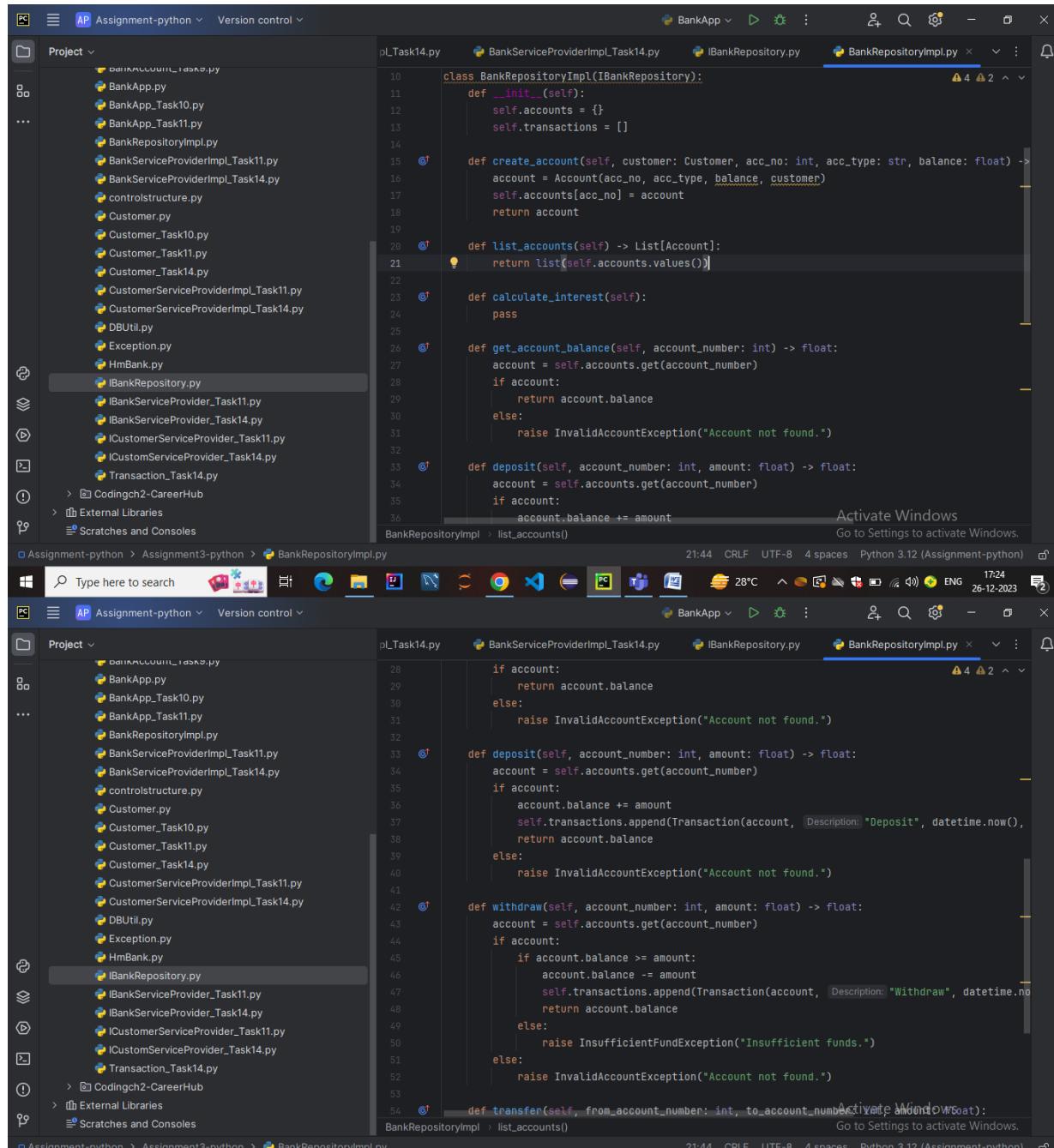
- transfer(from_account_number: long, to_account_number: int, amount: float): Transfer money from one account to another. check the balance from account in database and new balance should updated in Database.
- getAccountDetails(account_number: long): Should return the account and customer details from database.
- getTransactions(account_number: long, FromDate: Date, ToDate: Date): Should return the list of transaction between two dates from database



The screenshot shows a code editor with multiple tabs open. The active tab is 'IBankRepository.py'. The code defines an interface for a bank repository with methods for creating accounts, listing accounts, calculating interest, getting account balances, depositing, withdrawing, and transferring funds. All methods contain a 'pass' statement as a placeholder.

```
1  from typing import List
2  from Customer_Task14 import Customer
3  from Account_Task14 import Account
4  @dataclass
5  class IBankRepository:
6      def create_account(self, customer: Customer, acc_no: int, acc_type: str, balance: float) -> Account:
7          pass
8
9      def list_accounts(self) -> List[Account]:
10         pass
11
12     def calculate_interest(self):
13         pass
14
15     def get_account_balance(self, account_number: int) -> float:
16         pass
17
18     def deposit(self, account_number: int, amount: float) -> float:
19         pass
20
21     def withdraw(self, account_number: int, amount: float) -> float:
22         pass
23
24     def transfer(self, from_account_number: int, to_account_number: int, amount: float):
25         pass
```

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



The screenshot shows two instances of the Visual Studio Code (VS Code) interface. Both instances have the same project structure and code editor open, displaying the `BankRepositoryImpl.py` file.

Project Structure:

- Assignment-python
- Assignment3-python
- BankApp
- IBankAccount_Task5.py
- BankApp.py
- BankApp_Task10.py
- BankApp_Task11.py
- BankRepositoryImpl.py
- BankServiceProviderImpl_Task11.py
- BankServiceProviderImpl_Task14.py
- controlstructure.py
- Customer.py
- Customer_Task10.py
- Customer_Task11.py
- Customer_Task14.py
- CustomerServiceProviderImpl_Task11.py
- CustomerServiceProviderImpl_Task14.py
- DBUtil.py
- Exception.py
- HmBank.py
- IBankRepository.py
- IBankServiceProvider_Task1.py
- IBankServiceProvider_Task14.py
- ICustomerServiceProvider_Task1.py
- ICustomServiceProvider_Task14.py
- Transaction_Task14.py

Code Editor (Top Instance):

```
10  class BankRepositoryImpl(IBankRepository):
11      def __init__(self):
12          self.accounts = {}
13          self.transactions = []
14
15      @lru_cache(maxsize=1)
16      def create_account(self, customer: Customer, acc_no: int, acc_type: str, balance: float) -> Account:
17          account = Account(acc_no, acc_type, balance, customer)
18          self.accounts[acc_no] = account
19
20          return account
21
22      @lru_cache(maxsize=1)
23      def list_accounts(self) -> List[Account]:
24          return list(self.accounts.values())
25
26      def calculate_interest(self):
27          pass
28
29      def get_account_balance(self, account_number: int) -> float:
30          account = self.accounts.get(account_number)
31
32          if account:
33              return account.balance
34          else:
35              raise InvalidAccountException("Account not found.")
36
37      def deposit(self, account_number: int, amount: float) -> float:
38          account = self.accounts.get(account_number)
39
40          if account:
41              account.balance += amount
42
43          return account.balance
44
45      def withdraw(self, account_number: int, amount: float) -> float:
46          account = self.accounts.get(account_number)
47
48          if account:
49              if account.balance >= amount:
50                  account.balance -= amount
51                  self.transactions.append(Transaction(account, Description="Withdraw", datetime.now()))
52
53          return account.balance
54
55      def transfer(self, from_account_number: int, to_account_number: int, amount: float) -> float:
56          from_account = self.accounts.get(from_account_number)
57          to_account = self.accounts.get(to_account_number)
```

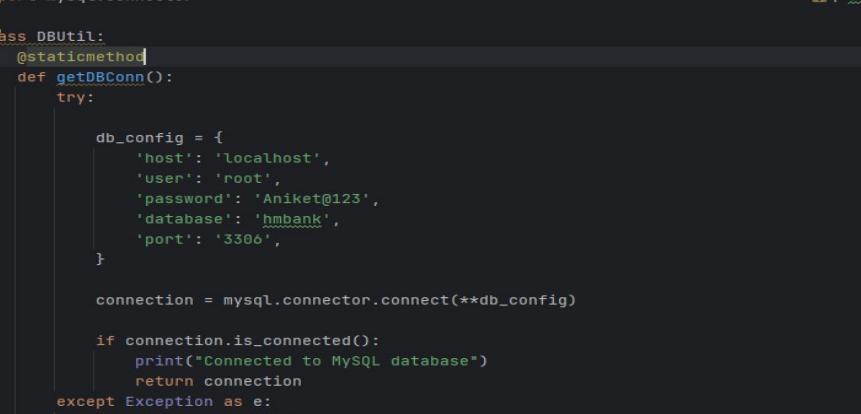
Code Editor (Bottom Instance):

```
28          if account:
29              return account.balance
30          else:
31              raise InvalidAccountException("Account not found.")
32
33      def deposit(self, account_number: int, amount: float) -> float:
34          account = self.accounts.get(account_number)
35
36          if account:
37              account.balance += amount
38              self.transactions.append(Transaction(account, Description="Deposit", datetime.now()),
39
40          return account.balance
41
42      def withdraw(self, account_number: int, amount: float) -> float:
43          account = self.accounts.get(account_number)
44
45          if account:
46              if account.balance >= amount:
47                  account.balance -= amount
48                  self.transactions.append(Transaction(account, Description="Withdraw", datetime.now(),
49
50          return account.balance
51
52      def transfer(self, from_account_number: int, to_account_number: int, amount: float) -> float:
53          from_account = self.accounts.get(from_account_number)
54          to_account = self.accounts.get(to_account_number)
```

11. Create DBUtil class and add the following method.

- static getDBConn():Connection

Establish a connection to the database and return Connection reference.



The screenshot shows the PyCharm IDE interface with multiple tabs open. The current tab is 'DBUtil.py'. The code in the editor is as follows:

```
1 import mysql.connector
2
3 class DBUtil:
4     @staticmethod
5     def getDBConn():
6         try:
7
8             db_config = {
9                 'host': 'localhost',
10                'user': 'root',
11                'password': 'Aniket@123',
12                'database': 'hmbank',
13                'port': '3306',
14            }
15
16            connection = mysql.connector.connect(**db_config)
17
18            if connection.is_connected():
19                print("Connected to MySQL database")
20                return connection
21            except Exception as e:
22                print(f"Error: {e}")
23                return None
```

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 ×  ▾  ⚡ 6 ⚡ 5 ▾ ▾
```

```
70     def create_savings_account(self):
71         customer = self.get_customer_details()
72         acc_no = self.generate_account_number()
73         acc_type = "Savings"
74         balance = float(input("Enter Initial Balance: "))
75         self.bank_service_provider.create_account(customer, acc_no, acc_type, balance)
76         print(f"Savings Account created successfully. Account Number: {acc_no}")
77
78     1 usage
79     def create_current_account(self):
80         customer = self.get_customer_details()
81         acc_no = self.generate_account_number()
82         acc_type = "Current"
83         balance = float(input("Enter Initial Balance: "))
84         self.bank_service_provider.create_account(customer, acc_no, acc_type, balance)
85         print(f"Current Account created successfully. Account Number: {acc_no}")
86
87     1 usage
88     def create_zero_balance_account(self):
89         customer = self.get_customer_details()
90         acc_no = self.generate_account_number()
91         acc_type = "Zero Balance"
92         balance = 0.0
93         self.bank_service_provider.create_account(customer, acc_no, acc_type, balance)
94         print(f"Zero Balance Account created successfully. Account Number: {acc_no}")
95
96     def deposit(self):
97         account_number = int(input("Enter Account Number: "))
98         amount = float(input("Enter Deposit Amount: "))
99         new_balance = self.bank_service_provider.deposit(account_number, amount)
100        print(f"Deposit successful. New Balance: {new_balance:.2f}")
101
102    except InsufficientFundException as e:
103        print(f"Error: {e}")
104
105    1 usage
106    def withdraw(self):
107        account_number = int(input("Enter Account Number: "))
108        amount = float(input("Enter Withdrawal Amount: "))
109        try:
110            new_balance = self.bank_service_provider.withdraw(account_number, amount)
111            print(f"Withdraw successful. New Balance: {new_balance:.2f}")
112        except InsufficientFundException as e:
113            print(f"Error: {e}")
114
115    def get_balance(self):
116        account_number = int(input("Enter Account Number: "))
117        try:
118            balance = self.bank_service_provider.get_account_balance(account_number)
119            print(f"Account Balance: {balance:.2f}")
120        except InvalidAccountException as e:
121            print(f"Error: {e}")
122
123    def transfer(self):
124        from_account_number = int(input("Enter From Account Number: "))
125        to_account_number = int(input("Enter To Account Number: "))
126        amount = float(input("Enter Transfer Amount: "))
127        try:
128            self.bank_service_provider.transfer(from_account_number, to_account_number, amount)
129            print("Transfer successful.")
130        except (InvalidAccountException, InsufficientFundException) as e:
131            print(f"Error: {e}")
132
133    def get_account_details(self):
134        account_number = int(input("Enter Account Number: "))
135        try:
```

Activate Windows
Go to Settings to activate Windows.

```
erImpl_Task14.py      IBankRepository.py      BankRepositoryImpl.py      DBUtil.py      BankApp.py ×  ▾  ⚡ 6 ⚡ 5 ▾ ▾
```

```
104     new_balance = self.bank_service_provider.withdraw(account_number, amount)
105     print(f"Withdraw successful. New Balance: {new_balance:.2f}")
106 except InsufficientFundException as e:
107     print(f"Error: {e}")
108
109 1 usage
110 def get_balance(self):
111     account_number = int(input("Enter account number: "))
112     try:
113         balance = self.bank_service_provider.get_account_balance(account_number)
114         print(f"Account Balance: {balance:.2f}")
115     except InvalidAccountException as e:
116         print(f"Error: {e}")
117
118 def transfer(self):
119     from_account_number = int(input("Enter From Account Number: "))
120     to_account_number = int(input("Enter To Account Number: "))
121     amount = float(input("Enter Transfer Amount: "))
122     try:
123         self.bank_service_provider.transfer(from_account_number, to_account_number, amount)
124         print("Transfer successful.")
125     except (InvalidAccountException, InsufficientFundException) as e:
126         print(f"Error: {e}")
127
128 def get_account_details(self):
129     account_number = int(input("Enter Account Number: "))
130     try:
```

Activate Windows
Go to Settings to activate Windows.

BankApp > main()

```
erImpl_Task14.py   IBankRepository.py   BankRepositoryImpl.py   DBUtil.py   BankApp.py   Activate Windows
135     def list_accounts(self):
136         accounts = self.bank_service_provider.list_accounts()
137         if accounts:
138             print("\nList of Accounts:")
139             for account in accounts:
140                 print(account)
141         else:
142             print("No accounts found.")
143
144     1 usage
145     def get_transactions(self):
146         account_number = int(input("Enter Account Number: "))
147         from_date = datetime.strptime(input("Enter From Date (YYYY-MM-DD): "), "%Y-%m-%d")
148         to_date = datetime.strptime(input("Enter To Date (YYYY-MM-DD): "), "%Y-%m-%d")
149         try:
150             transactions = self.bank_service_provider.get_transactions(account_number, from_date)
151             if transactions:
152                 print("\nList of Transactions:")
153                 for transaction in transactions:
154                     print(transaction)
155             else:
156                 print("No transactions found.")
157         except InvalidAccountException as e:
158             print(f"Error: {e}")
159
160     3 usages
161     def generate_account_number(self):
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
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.

```
erImpl_Task14.py   IBankRepository.py   BankRepositoryImpl.py   DBUtil.py   BankApp.py   Activate Windows
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
BankApp > main()
```

Activate Windows
Go to Settings to activate Windows.