

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

```
8o 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 1 usage  
7  def main():  
8  
9      credit_score = int(input("Enter your credit score: "))  
10     annual_income = float(input("Enter your annual income: $"))  
11  
12     eligibility = check_loan_eligibility(credit_score, annual_income)  
13  
14     if eligibility:  
15         print("Congratulations! You are eligible for a loan.")  
16     else:  
17         print("Sorry, you are not eligible for a loan at this time.")  
18 18 ▶ if __name__ == "__main__":  
19      main()
```

OUTPUT:-

```
⑧ ↴ Enter your credit score: 725  
⑨ ↵ Enter your annual income: $100000  
⑩ ↵ Congratulations! You are eligible for a loan.  
⑪ ↵ Process finished with exit code 0
```

Activate Windows

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.

```
... 41     def main1():
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.")
```

Display appropriate messages for success or failure

```
↓ 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     #TASK 3
64     1 usage
65     def calculate_future_balance(initial_balance, annual_interest_rate, years):
66         future_balance = initial_balance * (1 + annual_interest_rate / 100) ** years
67         return future_balance
68
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.

```
84 #TASK_4
85     1 usage
86     def get_account_balance(accounts, account_number):
87         if account_number in accounts:
88             return accounts[account_number]
89         else:
90             return None
91     1 usage
92     def main3():
93         bank_accounts = {
94             "123456": 1000.50,
95             "789012": 500.25,
96             "345678": 1500.75
97         }
98         while True:
99
100             account_number = input("Enter your account number: ")
101
102             account_balance = get_account_balance(bank_accounts, account_number)
103
104             if account_balance is not None:
105                 print(f"Account Balance for Account {account_number}: ${account_balance:.2f}")
106                 break
107             else:
108                 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

```
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 def is_valid_password(password):
110     if len(password) < 8:
111         return False
112     if not any(char.isupper() for char in password):
113         return False
114
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
138     def main():
139         transaction_history = []
140         while True:
141             print("\nOptions:")
142             print("1. Add Deposit")
143             print("2. Add Withdrawal")
144             print("3. Exit")
145             choice = input("Enter your choice (1, 2, or 3): ")
146
147             if choice == '1':
148                 deposit_amount = float(input("Enter the deposit amount: $"))
149                 transaction_history.append(f"Deposit: +${deposit_amount:.2f}")
150
151             elif choice == '2':
152                 withdrawal_amount = float(input("Enter the withdrawal amount: $"))
153                 transaction_history.append(f"Withdrawal: -${withdrawal_amount:.2f}")
154
155             elif choice == '3':
156                 display_transaction_history(transaction_history)
157                 print("Exiting program. Thank you!")
158                 break
159
160             else:
161                 print("Invalid choice. Please enter 1, 2, or 3.")
```

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
 - o Customer ID
 - o First Name
 - o Last Name
 - o Email Address
 - o Phone Number
 - o 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         if new_FirstName:
```

Activate Windows
Go to Settings to activate Windows

- Constructor and Methods

```
31     # Getter and setter methods for First 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
```

Activate Windows

- 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
53     @property
54     def PhoneNumber(self):
55         return self._PhoneNumber
56
57     @PhoneNumber.setter
58     def PhoneNumber(self, new_PhoneNumber):
59         if isinstance(new_PhoneNumber, str) and new_PhoneNumber.isdigit():
60             self._PhoneNumber = new_PhoneNumber
61         else:
62             raise ValueError("Invalid phone number format.")
63
64     1 usage
65     @property
66     def Address(self):
67         return self._Address
68
69     @Address.setter
70     def Address(self, new_Address):
71         if isinstance(new_Address, str) and new_Address:
72             self._Address = new_Address
73         else:
74             raise ValueError("Address must be a non-empty string.")

```

Activate Windows

2. Create an `Account` class with the following confidential attributes:

- Attributes

- Account Number

- Account Type (e.g., Savings, Current)

- 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
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

- Implement default constructors and overload the constructor with Account attributes,

- 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.")

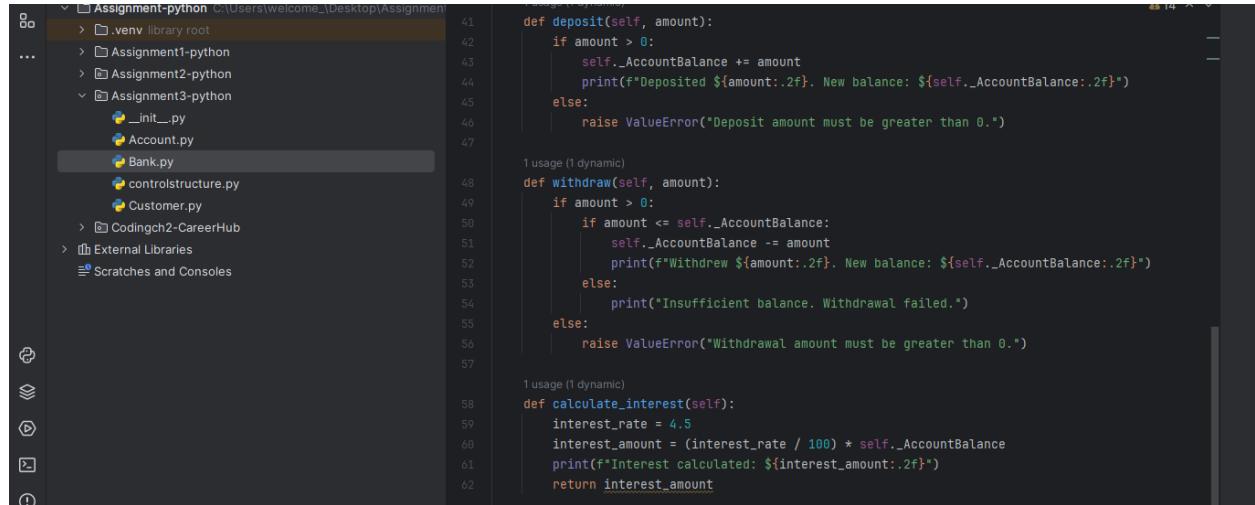
```

- 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%



```
Assignment-python C:\Users\welcome\Desktop\Assignment
> venv library root
...
> Assignment1-python
> Assignment2-python
> Assignment3-python
  __init__.py
  Account.py
  Bank.py
  controlstructure.py
  Customer.py
> Codingch2-CareerHub
> External Libraries
  Scratches and Consoles

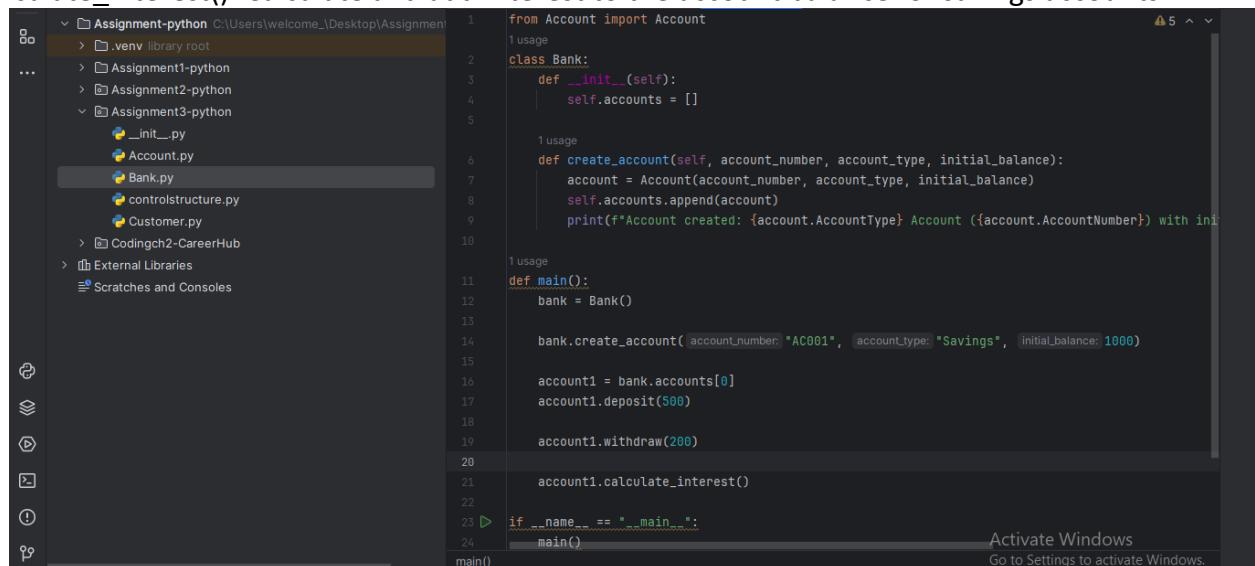
41 def deposit(self, amount):
42     if amount > 0:
43         self._AccountBalance += amount
44         print(f"Deposited ${amount:.2f}. New balance: ${self._AccountBalance:.2f}")
45     else:
46         raise ValueError("Deposit amount must be greater than 0.")
47
48 1 usage (1 dynamic)
49 def withdraw(self, amount):
50     if amount > 0:
51         if amount <= self._AccountBalance:
52             self._AccountBalance -= amount
53             print(f"Withdraw ${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.



```
Assignment-python C:\Users\welcome\Desktop\Assignment
> venv library root
...
> Assignment1-python
> Assignment2-python
> Assignment3-python
  __init__.py
  Account.py
  Bank.py
  controlstructure.py
  Customer.py
> Codingch2-CareerHub
> External Libraries
  Scratches and Consoles

1 from Account import Account
2
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 initial balance ${initial_balance:.2f}")
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.calculate_interest()
23
24    if __name__ == "__main__":
25        main()
```

Activate Windows
Go to Settings to activate Windows.

OUTPUT:-

```

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"Withdraw ${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).

```

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
75     class CurrentAccount(Account):
76         OVERDRAFT_LIMIT = 1000
77         def __init__(self, account_number, account_balance):
78             super().__init__(account_number, account_balance)
79             1 usage (1 dynamic)
80             def withdraw(self, amount):
81                 if amount > 0:
82                     available_balance = self.account_balance + self.OVERDRAFT_LIMIT
83                     if amount <= available_balance:
84                         self.account_balance -= amount
85                         print(f"Withdraw ${amount:.2f}. New balance: ${self.account_balance:.2f}")
86                     else:
87                         print("Withdrawal limit exceeded. Withdrawal failed.")
88                 else:
89                     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
3 class Bank:
4     Usage
5     def create_account(self):
6         print("Select account type:")
7         print("1. Savings Account")
8         print("2. Current Account")
9         choice = int(input("Enter your choice (1 or 2): "))
10        account_number = input("Enter account number: ")
11        initial_balance = float(input("Enter initial balance: "))
12
13        if choice == 1:
14            interest_rate = float(input("Enter interest rate for savings account: "))
15            return SavingsAccount(account_number, initial_balance, interest_rate)
16        elif choice == 2:
17            return CurrentAccount(account_number, initial_balance)
18        else:
19            print("Invalid choice. Creating a generic account.")
20            return Account(account_number, AccountType.Generic, initial_balance)
21
22    Usage
23    def main(self):
24        print("Customer Create Account first follow below steps")
25        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

OUTPUTS:-

```

*** ↑ 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:
 - o Account number.
 - o Customer name.
 - o Balance

```
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.

```
@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.

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

- o calculate_interest(): Abstract method for calculating interest.

```
1 usage (1 dynamic)
@abstractmethod
def deposit(self, amount):
    pass

1 usage (1 dynamic)
@abstractmethod
def withdraw(self, amount):
    pass

1 usage (1 dynamic)
@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

```
50 52     class SavingsAccount(BankAccount):  
... 53         def __init__(self, account_number="", customer_name="", balance=0.0, interest_rate=0.0):  
54             super().__init__(account_number, customer_name, balance)  
55             self._interest_rate = interest_rate  
56  
57             1 usage  
58             @property  
59             def interest_rate(self):  
60                 return self._interest_rate  
61  
62             @interest_rate.setter  
63             def interest_rate(self, new_interest_rate):  
64                 if isinstance(new_interest_rate, (int, float)) and 0 <= new_interest_rate <= 100:  
65                     self._interest_rate = new_interest_rate  
66                 else:  
67                     raise ValueError("Interest rate must be a number between 0 and 100.")  
68 68 1 usage (1 dynamic)  
69             def deposit(self, amount):  
70                 if isinstance(amount, (float, int)) and amount > 0:  
71                     self._balance += amount  
72                     print(f"Deposited ${amount:.2f}. New balance: ${self._balance:.2f}")  
73                 else:  
74                     raise ValueError("Deposit amount must be a positive number.")  
  
80 68 68 1 usage (1 dynamic)  
69             def deposit(self, amount):  
70                 if isinstance(amount, (float, int)) and amount > 0:  
71                     self._balance += amount  
72                     print(f"Deposited ${amount:.2f}. New balance: ${self._balance:.2f}")  
73                 else:  
74                     raise ValueError("Deposit amount must be a positive number.")  
75 75 1 usage (1 dynamic)  
76             def withdraw(self, amount):  
77                 if isinstance(amount, (float, int)) and amount > 0:  
78                     if amount <= self._balance:  
79                         self._balance -= amount  
80                         print(f"Withdraw ${amount:.2f}. New balance: ${self._balance:.2f}")  
81                     else:  
82                         print("Insufficient balance. Withdrawal failed.")  
83                 else:  
84                     raise ValueError("Withdrawal amount must be a positive number.")  
85 85 1 usage (1 dynamic)  
86             def calculate_interest(self):  
87                 interest_amount = (self._interest_rate / 100) * self._balance  
88                 self._balance += interest_amount  
89                 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).

```
90 90     class CurrentAccount(BankAccount):  
91         OVERDRAFT_LIMIT = 1000  
92  
93         def __init__(self, account_number="", customer_name="", balance=0.0):  
94             super().__init__(account_number, customer_name, balance)  
95  
96 96 1 usage (1 dynamic)  
97             def deposit(self, amount):  
98                 if isinstance(amount, (float, int)) and amount > 0:  
99                     self._balance += amount  
100                    print(f"Deposited ${amount:.2f}. New balance: ${self._balance:.2f}")  
101                else:  
102                    raise ValueError("Deposit amount must be a positive number.")  
103 103 1 usage (1 dynamic)  
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.")
```

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();

The screenshot shows a code editor with a file tree on the left and a code editor window on the right. The file tree shows a directory structure with several sub-directories and files. The code editor window displays a Python script named 'Bank_Task9.py'.

```
from BankAccount import SavingsAccount, CurrentAccount

class Bank:
    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.")

            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

The screenshot shows a code editor with a file tree on the left and a code editor window on the right. The file tree shows a directory structure with several sub-directories and files. The code editor window displays a Python script named 'Bank_Task9.py'.

```
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

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): ")
```

```
Bank_Task9.py
BankAccount.py
controlstructure.py
Customer.py
> Codingch2-CareerHub
> External Libraries
Scratches and Consoles

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()

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: Anklet
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
Withdrew $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: Bigani
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
```

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

The screenshot shows a code editor with a file tree on the left and a code editor window on the right. The file tree shows a project structure with several files: Assignment1-python, Assignment2-python, Assignment3-python, Account.py, Account_Task10.py, Bank_Task7.py, Bank_Task8.py, Bank_Task9.py, BankAccount.py, controlstructure.py, Customer.py, and Customer_Task10.py. The code editor window displays the following Python code:

```
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    1 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):
```

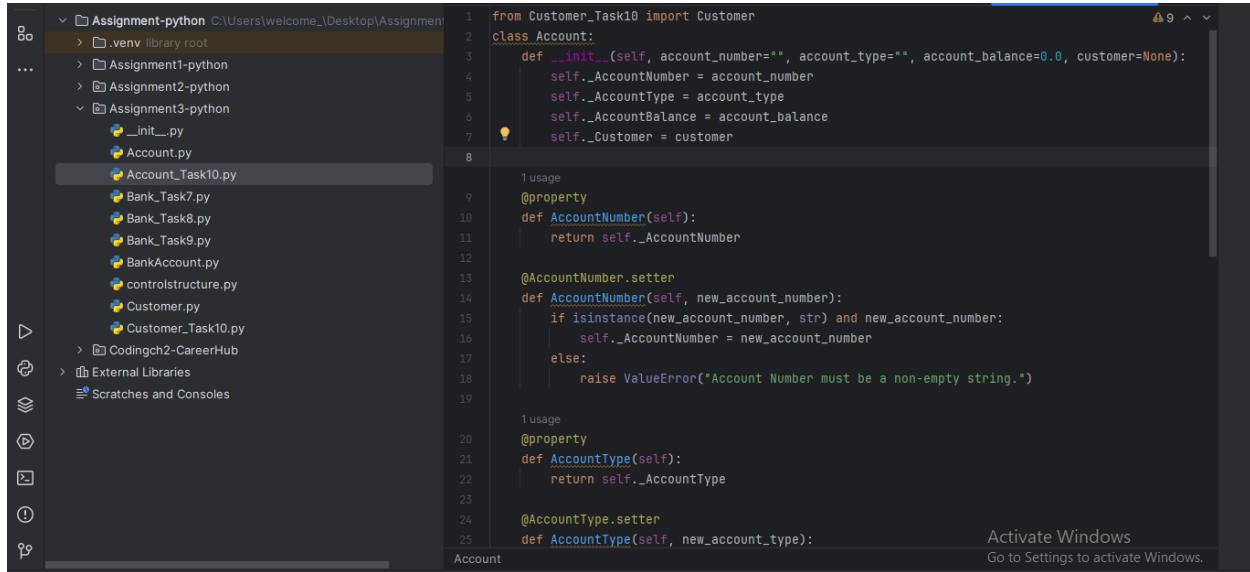
- Methods and Constructor:
 - Implement default constructors and overload the constructor with Account attributes, generate getter, setter, print all information of attribute) methods for the attributes

The screenshot shows a code editor with a file tree on the left and a code editor window on the right. The file tree is identical to the previous screenshot. The code editor window displays the following Python code, which includes validation logic for the email and phone number attributes:

```
40
41    self._LastName = new_last_name
42
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
50
51    @Email.setter
52    def Email(self, new_email):
53        if isinstance(new_email, str) and re.match(r"^\w+@\w+\.\w+", new_email):
54            self._Email = new_email
55        else:
56            raise ValueError("Invalid Email format.")
57
58    1 usage
59    @property
60    def PhoneNumber(self):
61        return self._PhoneNumber
62
63    @PhoneNumber.setter
64    def PhoneNumber(self, new_phone_number):
65        if isinstance(new_phone_number, str) and re.match(r"\d{10}", new_phone_number):
66            self._PhoneNumber = new_phone_number
67        else:
68            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)



The screenshot shows a code editor with a file tree on the left and a code editor window on the right. The file tree shows a folder structure with several Python files. The code editor window displays the following Python code for the `Account` class:

```
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

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

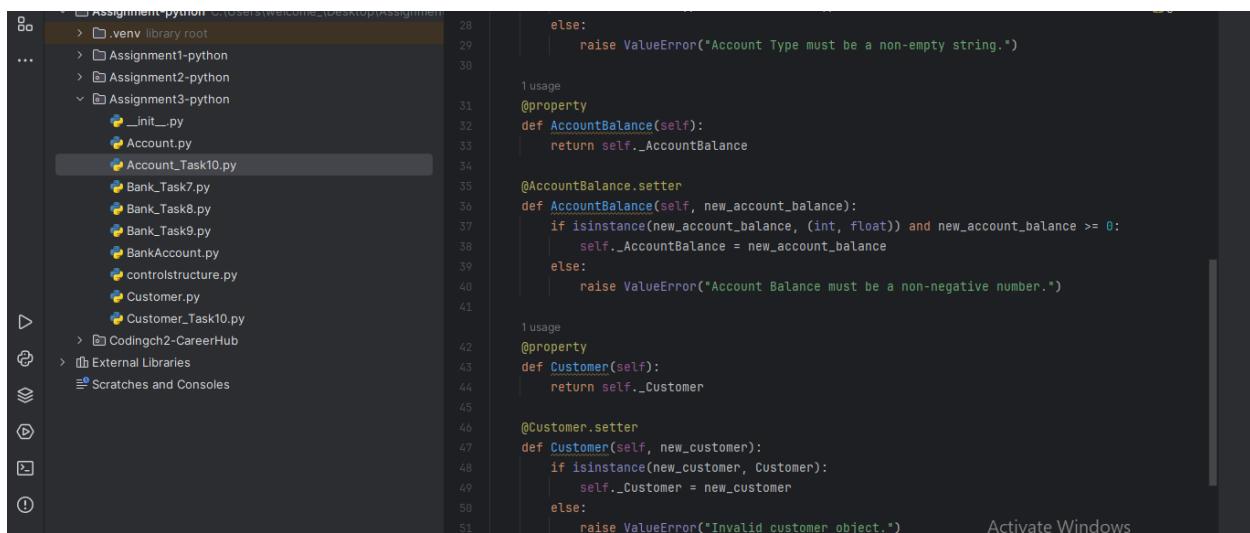
    @AccountNumber.setter
    def AccountNumber(self, new_account_number):
        if isinstance(new_account_number, str) and new_account_number:
            self._AccountNumber = new_account_number
        else:
            raise ValueError("Account Number must be a non-empty string.")

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

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

Activate Windows
Go to Settings to activate Windows.

- 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.



The screenshot shows a code editor with a file tree on the left and a code editor window on the right. The file tree shows a folder structure with several Python files. The code editor window displays the completed Python code for the `Account` class, including the `Customer` attribute and its methods:

```
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.")
```

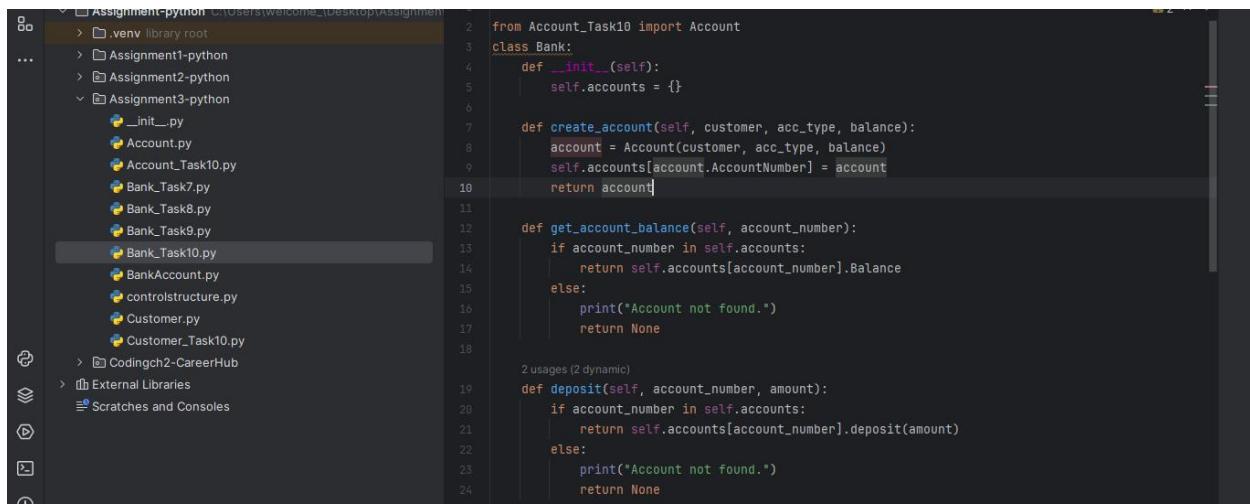
Activate Windows

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.



```

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

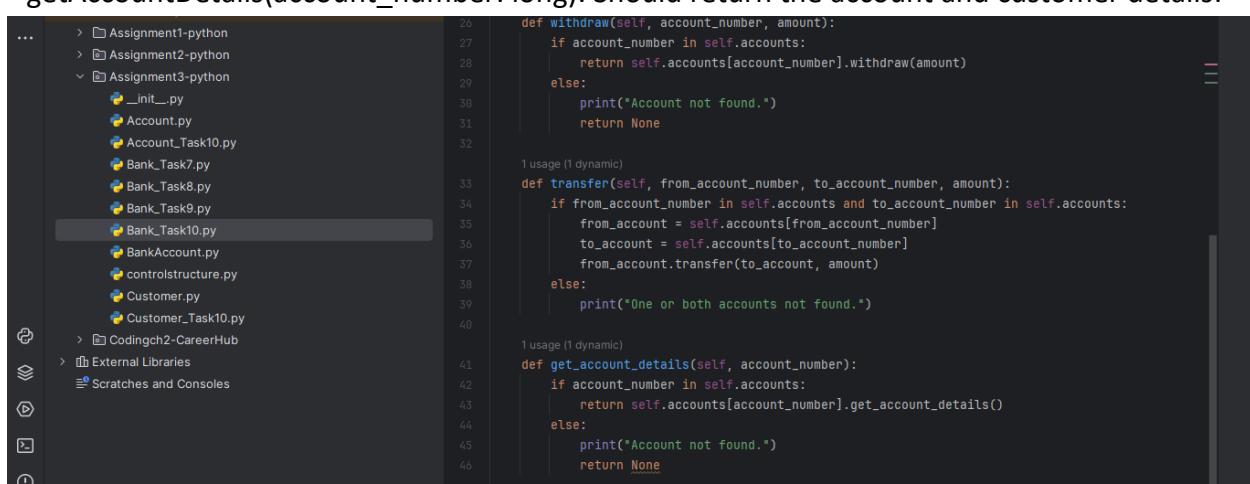
    def create_account(self, customer, acc_type, balance):
        account = Account(customer, acc_type, balance)
        self.accounts[account.AccountNumber] = account
        return account

    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.



```

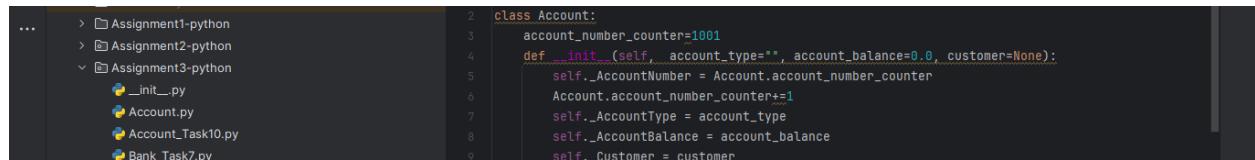
    def withdraw(self, account_number, amount):
        if account_number in self.accounts:
            return self.accounts[account_number].withdraw(amount)
        else:
            print("Account not found.")
            return None

    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.

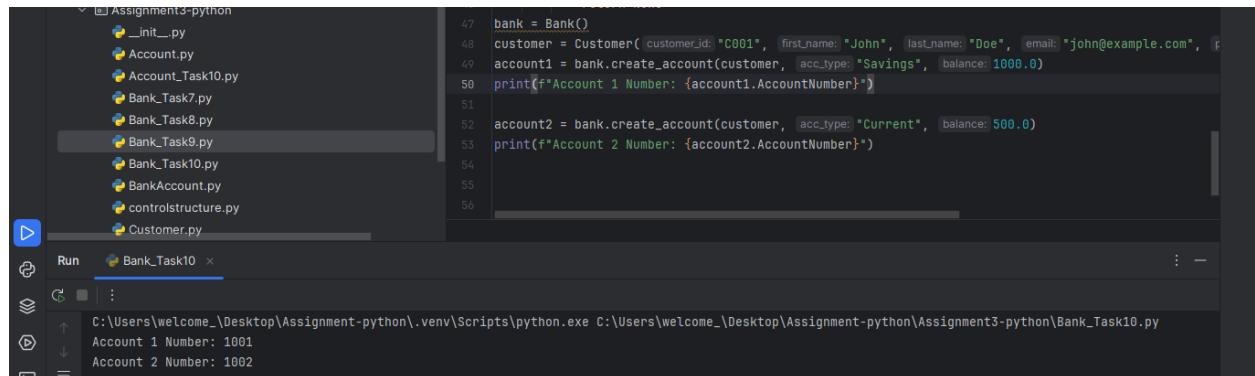


```

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

```

Output:-



```

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

```

Run Bank_Task10

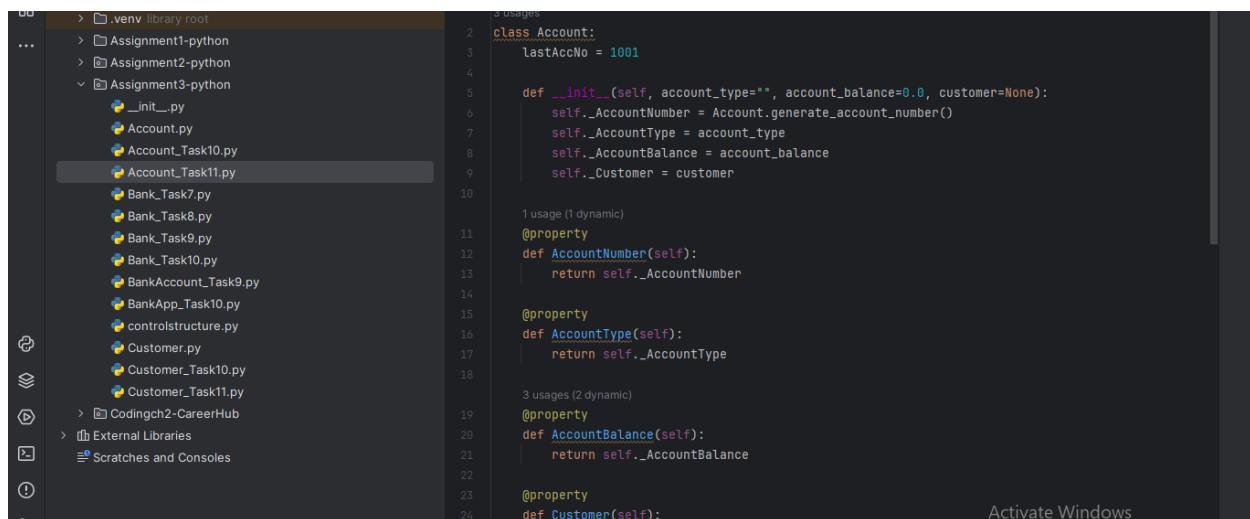
C:\Users\welcome\Desktop\Assignment-python\.venv\Scripts\python.exe C:\Users\welcome\Desktop\Assignment-python\Assignment3-python\Bank_Task10.py

Account 1 Number: 1001
Account 2 Number: 1002

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.

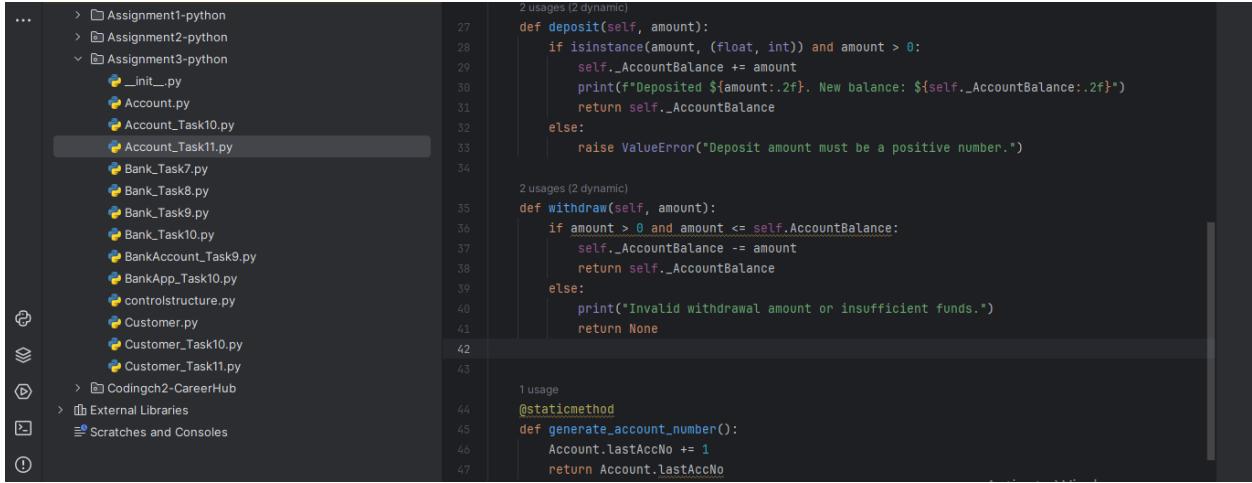


```

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

```

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



```

...
> Assignment1-python
> Assignment2-python
< Assignment3-python
  < __init__.py
  < Account.py
  < Account_Task10.py
  < Account_Task11.py
  < Bank_Task7.py
  < Bank_Task8.py
  < Bank_Task9.py
  < Bank_Task10.py
  < BankAccount_Task9.py
  < BankApp_Task10.py
  < controlstructure.py
  < Customer.py
  < Customer_Task10.py
  < Customer_Task11.py
> Codingch2-CareerHub
> External Libraries
> Scratches and Consoles
...
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.")

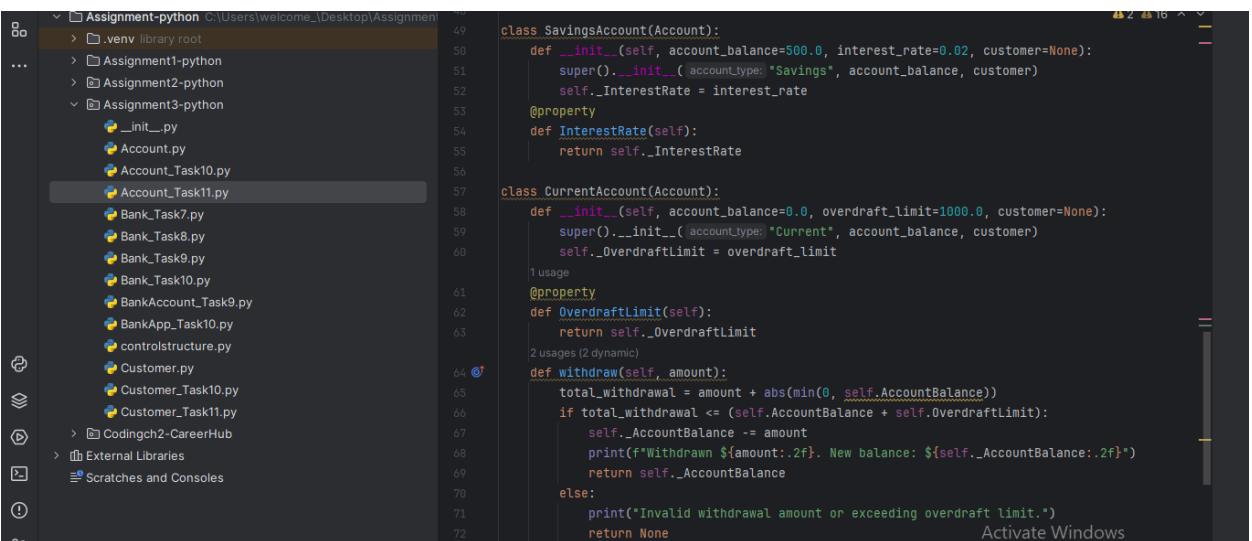
2 usages (2 dynamic)
def withdraw(self, amount):
    if amount > 0 and amount <= self._AccountBalance:
        self._AccountBalance -= amount
        return self._AccountBalance
    else:
        print("Invalid withdrawal amount or insufficient funds.")
        return None

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.



```

...
> Assignment-python C:\Users\welcome_1\Desktop\Assignment
  < .venv library root
  > Assignment1-python
  > Assignment2-python
  < Assignment3-python
    < __init__.py
    < Account.py
    < Account_Task10.py
    < Account_Task11.py
    < Bank_Task7.py
    < Bank_Task8.py
    < Bank_Task9.py
    < Bank_Task10.py
    < BankAccount_Task9.py
    < BankApp_Task10.py
    < controlstructure.py
    < Customer.py
    < Customer_Task10.py
    < Customer_Task11.py
  > Codingch2-CareerHub
  > External Libraries
  > Scratches and Consoles
...
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

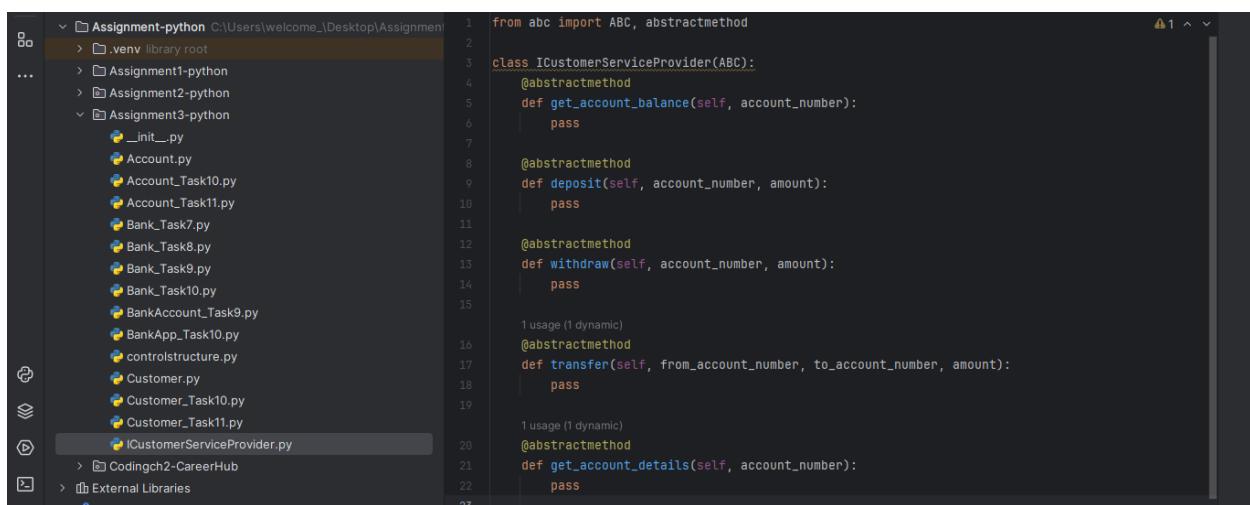
class CurrentAccount(Account):
    def __init__(self, account_balance=0.0, overdraft_limit=1000.0, customer=None):
        super().__init__(account_type="Current", account_balance, customer)
        self._OverdraftLimit = overdraft_limit
    @usage
    @property
    def OverdraftLimit(self):
        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"Withdraw ${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.

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



```
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

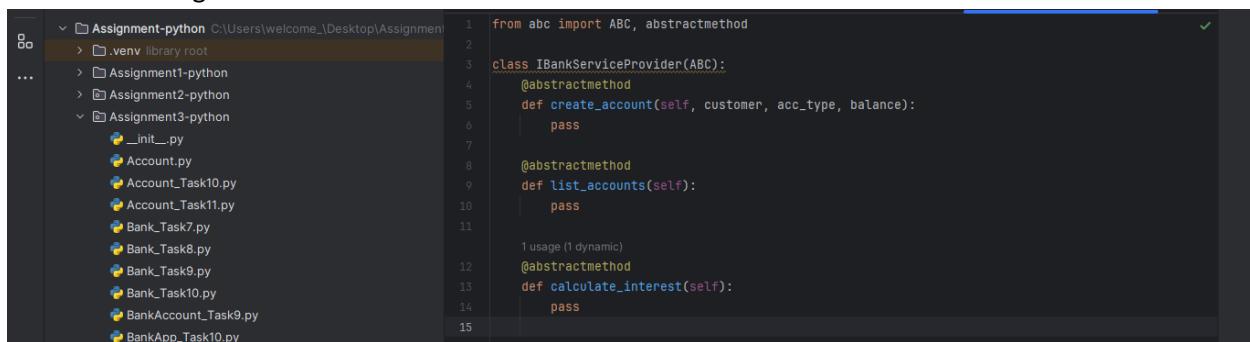
    @abstractmethod
    def withdraw(self, account_number, amount):
        pass

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

    @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.



```
from abc import ABC, abstractmethod

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

    @abstractmethod
    def list_accounts(self):
        pass

    @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.

6.

```
1 from ICustomerServiceProvider import ICustomerServiceProvider
2 usages
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         return None
12 
13     def deposit(self, account_number, amount):
14         for account in self.accounts:
15             if account.AccountNumber == account_number:
16                 return account.deposit(amount)
17         return None
18 
19     def withdraw(self, account_number, amount):
20         for account in self.accounts:
21             if account.AccountNumber == account_number:
22                 return account.withdraw(amount)
23         return None
24 
25 1 usage (1 dynamic)
26     def transfer(self, from_account_number, to_account_number, amount):
27         from_account = None
```

8.

```
...  
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

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. 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

```
from datetime import datetime

class Transaction:  
    def __init__(self, Account, Description, Transaction_type, Amount):  
        self._Account = Account  
        self._Description = Description  
        self._DateTime = datetime.now()  
        self._TransactionType = Transaction_type  
        self._TransactionAmount = Amount  
  
    @property  
    def Account(self):  
        return self._Account  
  
    @property  
    def Description(self):  
        return self._Description  
  
    @property  
    def DateTime(self):  
        return self._DateTime  
  
    @property  
    def TransactionType(self):  
        return self._TransactionType
```

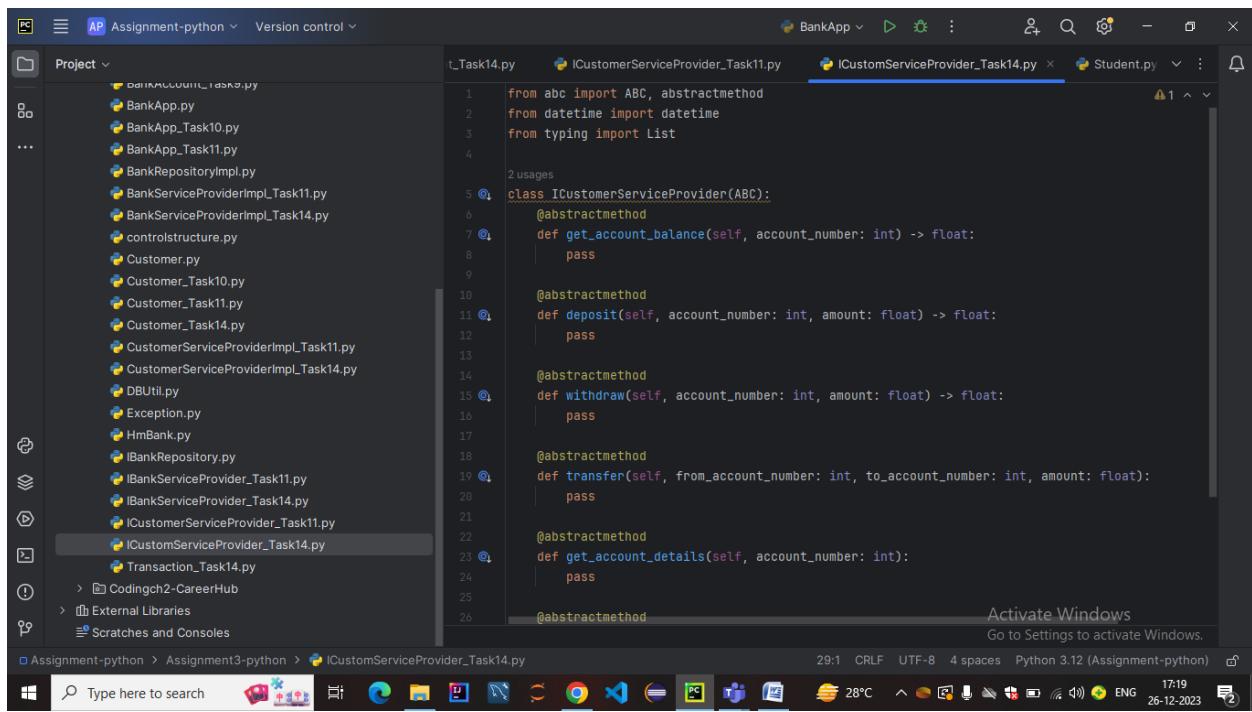
4. Create three child classes that inherit the Account class and each class must contain below mentioned attribute:

- 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):
41         return self._OverdraftLimit
```

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



```
from abc import ABC, abstractmethod
from datetime import datetime
from typing import List

class ICustomerServiceProvider(ABC):
    @abstractmethod
    def get_account_balance(self, account_number: int) -> float:
        pass

    @abstractmethod
    def deposit(self, account_number: int, amount: float) -> float:
        pass

    @abstractmethod
    def withdraw(self, account_number: int, amount: float) -> float:
        pass

    @abstractmethod
    def transfer(self, from_account_number: int, to_account_number: int, amount: float):
        pass

    @abstractmethod
    def get_account_details(self, account_number: int):
        pass
```

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.

```
from abc import ABC, abstractmethod
from typing import List

class IBankServiceProvider(ABC):
    @abstractmethod
    def create_account(self, customer: Customer, acc_no: int, acc_type: str, balance: float):
        pass

    @abstractmethod
    def list_accounts(self) -> List[Account]:
        pass

    @abstractmethod
    def get_account_details(self, account_number: int):
        pass

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

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

These methods do not interact with database directly.

```

from typing import List
from ICustomServiceProvider_Task14 import ICustomerServiceProvider
from datetime import datetime
from Exception import InvalidAccountException

class CustomerServiceImpl(ICustomerServiceProvider):
    def __init__(self):
        self.accounts_list = []

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

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

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

```



```

    return None

    def transfer(self, from_account_number: int, to_account_number: int, amount: float):
        from_account = None
        to_account = None

        for account in self.accounts_list:
            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:
            raise InvalidAccountException("One or both accounts not found.")

    def get_account_details(self, account_number: int):
        for account in self.accounts_list:
            if account.AccountNumber == account_number:
                return account.get_account_details()
        return None

```

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

```

BankApp v CustomerServiceProviderImpl_Task14.py BankServiceProviderImpl_Task14.py
Project v
Bank_Task9.py
Bank_Task10.py
BankAccount_Task9.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_Task11.py
IBankServiceProvider_Task14.py
ICustomerServiceProvider_Task14.py
ICustomerServiceProvider_Task14.py
Transaction_Task14.py
BankServiceProviderImpl > __init__()

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     @t 16 usages
17     def create_account(self, customer: Customer, acc_type: str, balance: float) -> Account:
18         account = Account(acc_type, balance, customer)
19         self.account_list.append(account)
20         self.accounts_list.append(account) # Add to the parent class list as well
21         return account
22     @t 1 usage
23     def list_accounts(self) -> List[Account]:
24         return self.account_list
25     @t 25 usages
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     @t 31 usages
32     def calculate_interest(self):
33         pass

```

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.

```

1  from typing import List
2  from Customer_Task14 import Customer
3  from Account_Task14 import Account
4  usages
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

```

The screenshot shows a code editor with several files open in tabs at the top: pL_Task14.py, BankServiceProviderImpl_Task14.py, IBankRepository.py (the current file), and BankRepositoryImpl.py. The left sidebar shows a project structure with various Python files. The code in IBankRepository.py implements the IBankRepository interface with methods for account creation, listing, interest calculation, balance retrieval, deposits, withdrawals, and transfers. The code editor interface includes a status bar at the bottom showing system information like date and time.

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

```

class BankRepositoryImpl(IBankRepository):
    def __init__(self):
        self.accounts = {}
        self.transactions = []

    def create_account(self, customer: Customer, acc_no: int, acc_type: str, balance: float) -> Account:
        account = Account(acc_no, acc_type, balance, customer)
        self.accounts[acc_no] = account
        return account

    def list_accounts(self) -> List[Account]:
        return list(self.accounts.values())

    def calculate_interest(self):
        pass

    def get_account_balance(self, account_number: int) -> float:
        account = self.accounts.get(account_number)
        if account:
            return account.balance
        else:
            raise InvalidAccountException("Account not found.")

    def deposit(self, account_number: int, amount: float) -> float:
        account = self.accounts.get(account_number)
        if account:
            account.balance += amount
            self.transactions.append(Transaction(account, Description="Deposit", datetime.now()))
            return account.balance
        else:
            raise InvalidAccountException("Account not found.")

    def withdraw(self, account_number: int, amount: float) -> float:
        account = self.accounts.get(account_number)
        if account:
            if account.balance >= amount:
                account.balance -= amount
                self.transactions.append(Transaction(account, Description="Withdraw", datetime.now()))
                return account.balance
            else:
                raise InsufficientFundException("Insufficient funds.")
        else:
            raise InvalidAccountException("Account not found.")

    def transfer(self, from_account_number: int, to_account_number: int, amount: float):
        from_account = self.accounts.get(from_account_number)
        to_account = self.accounts.get(to_account_number)
        if from_account and to_account:
            if from_account.balance >= amount:
                from_account.balance -= amount
                to_account.balance += amount
                self.transactions.append(Transaction(from_account, Description="Transfer", datetime.now()))
                self.transactions.append(Transaction(to_account, Description="Transfer", datetime.now()))
                return True
            else:
                raise InsufficientFundException("Insufficient funds for transfer.")
        else:
            raise InvalidAccountException("One or both accounts not found.")


BankRepositoryImpl > list_accounts()

```

. 11. Create DBUtil class and add the following method. • static getDBConn():Connection

Establish a connection to the database and return Connection reference

```

import mysql.connector
class DBUtil:
    @staticmethod
    def getDBConn():
        try:
            db_config = {
                'host': 'localhost',
                'user': 'root',
                'password': 'Aniket@123',
                'database': 'HmBank',
                'port': '3306',
            }
            connection = mysql.connector.connect(**db_config)
            if connection.is_connected():
                print("Connected to MySQL database")
                return connection
        except Exception as e:
            print(f"Error: {e}")
        return None

```

12. 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",

```

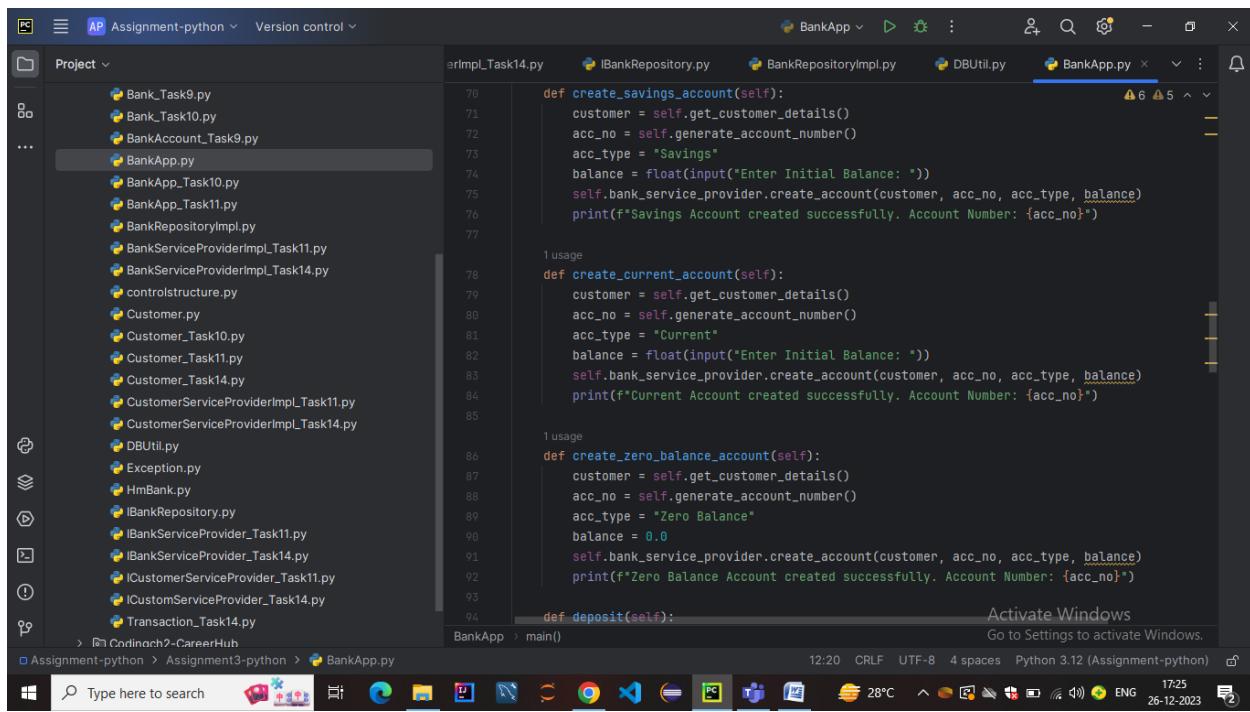
print("Invalid choice. Please enter a number between 1 and 9.*")
def create_account(self):
    while True:
        print("\nChoose Account Type:")
        print("1. Savings Account")
        print("2. Current Account")
        print("3. Zero Balance Account")
        print("4. Back to Main Menu")

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

        if acc_type_choice == "1":
            self.create_savings_account()
        elif acc_type_choice == "2":
            self.create_current_account()
        elif acc_type_choice == "3":
            self.create_zero_balance_account()
        elif acc_type_choice == "4":
            break
        else:
            print("Invalid choice. Please enter a number between 1 and 4.*")

```

"withdraw", "get_balance", "transfer"



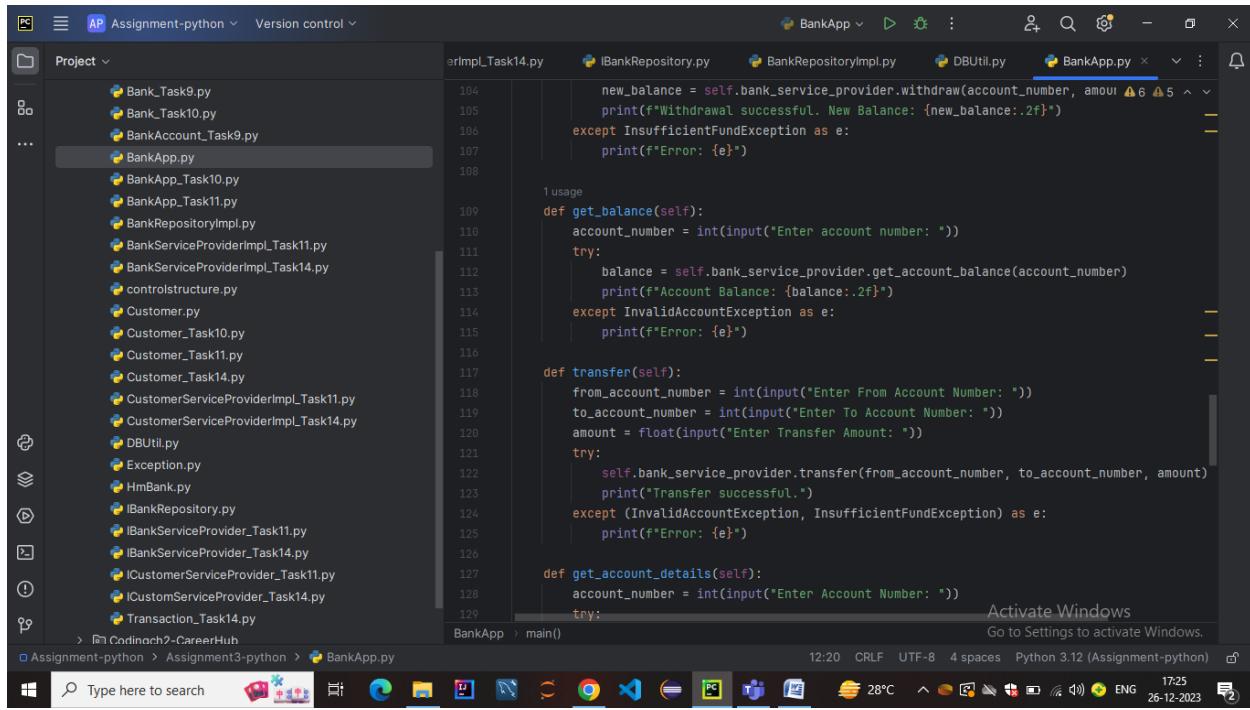
```
def create_savings_account(self):
    customer = self.get_customer_details()
    acc_no = self.generate_account_number()
    acc_type = "Savings"
    balance = float(input("Enter Initial Balance: "))
    self.bank_service_provider.create_account(customer, acc_no, acc_type, balance)
    print(f"Savings 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
    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):
    pass
```

, "getAccountDetails", "ListAccounts",



```
new_balance = self.bank_service_provider.withdraw(account_number, amount)
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:
        pass
    except:
        pass
```

"getTransactions" and "exit."

```
BankTask9.py 135 def list_accounts(self):
BankTask10.py 136     accounts = self.bank_service_provider.list_accounts()
BankAccount_Task9.py 137     if accounts:
BankApp.py 138         print("\nList of Accounts:")
BankApp_Task10.py 139         for account in accounts:
BankApp_Task11.py 140             print(account)
BankRepositoryImpl.py 141     else:
BankServiceProviderImpl_Task14.py 142         print("No accounts found.")
controlstructure.py 143
Customer.py 144     def get_transactions(self):
Customer_Task10.py 145         account_number = int(input("Enter Account Number: "))
Customer_Task11.py 146         from_date = datetime.strptime(input("Enter From Date (YYYY-MM-DD): "), "%Y-%m-%d")
Customer_Task14.py 147         to_date = datetime.strptime(input("Enter To Date (YYYY-MM-DD): "), "%Y-%m-%d")
CustomerServiceProviderImpl_Task11.py 148         try:
CustomerServiceProviderImpl_Task14.py 149             transactions = self.bank_service_provider.get_transactions(account_number, from_date)
DBUtil.py 150             if transactions:
Exception.py 151                 print("\nList of Transactions:")
HmBank.py 152                 for transaction in transactions:
IBankRepository.py 153                     print(transaction)
IBankServiceProvider_Task11.py 154                 else:
IBankServiceProvider_Task14.py 155                     print("No transactions found.")
ICustomerServiceProvider_Task11.py 156             except InvalidAccountException as e:
ICustomServiceProvider_Task14.py 157                 print(f"Error: {e}")
Transaction_Task14.py 158
BankApp 159     3 usages
        def generate_account_number(self):
BankApp > main()
```

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