ASSIGNMENT

BANKING SYSTEM

LOGESH.D

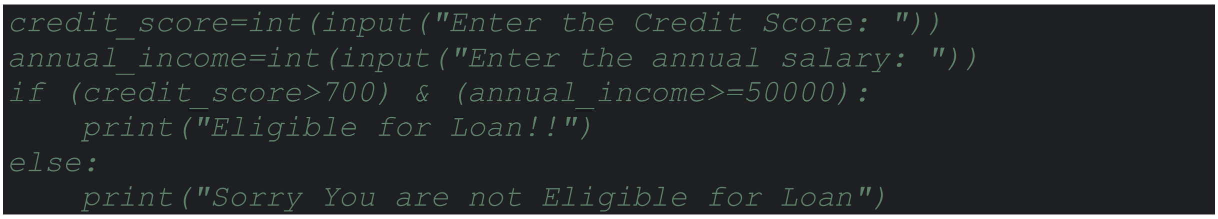
Task 1: Conditional Statements

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:

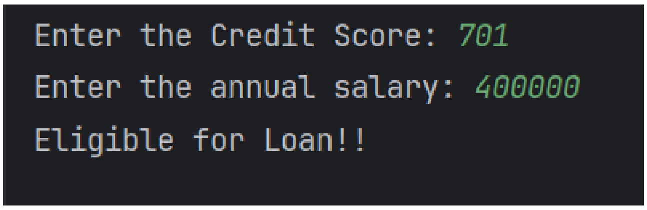
- Credit Score must be above 700.
- Annual Income must be at least $50,000.

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.

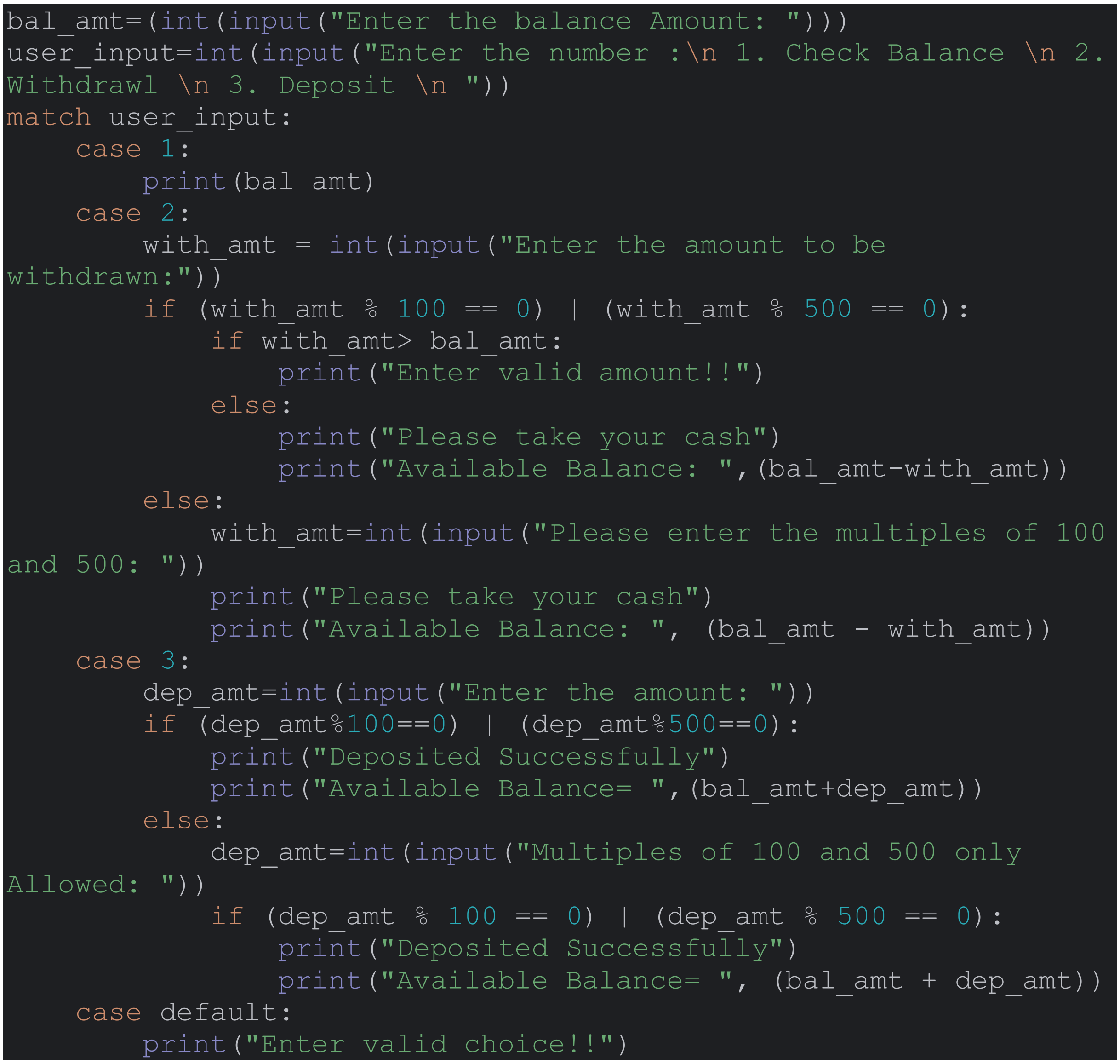


```
credit_score=int(input("Enter the Credit Score: "))
annual_income=int(input("Enter the annual salary: "))
if (credit_score>700) & (annual_income>=50000):
    print("Eligible for Loan!!")
else:
    print("Sorry You are not Eligible for Loan")
```

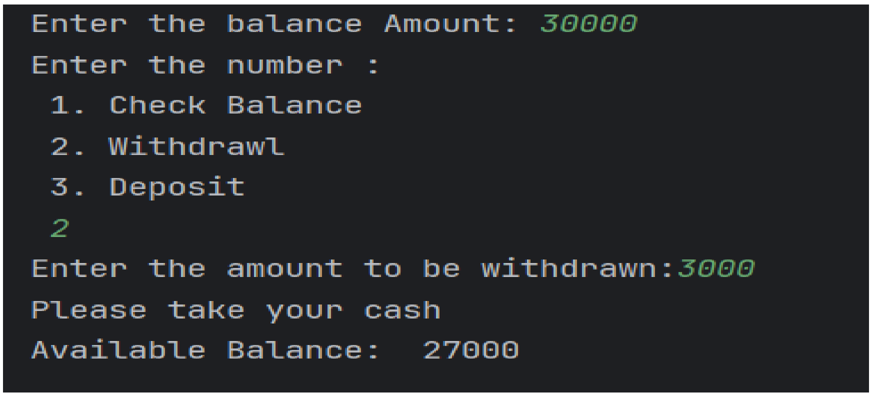


```
Enter the Credit Score: 701
Enter the annual salary: 400000
Eligible for Loan!!
```

Task 2: Nested Conditional Statements
Create a program that simulates an ATM transaction. Display options such as "Check Balance," "Withdraw," "Deposit,". 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.



```
bal_amt=(int(input("Enter the balance Amount: ")))
user_input=int(input("Enter the number :\n 1. Check Balance \n 2.
Withdrawl \n 3. Deposit \n "))
match user_input:
    case 1:
        print(bal_amt)
    case 2:
        with_amt = int(input("Enter the amount to be
withdrawn:"))
        if (with_amt % 100 == 0) | (with_amt % 500 == 0):
            if with_amt> bal_amt:
                print("Enter valid amount!!")
            else:
                print("Please take your cash")
                print("Available Balance: ",(bal_amt-with_amt))
        else:
            with_amt=int(input("Please enter the multiples of 100
and 500: "))
            print("Please take your cash")
            print("Available Balance: ", (bal_amt - with_amt))
    case 3:
        dep_amt=int(input("Enter the amount: "))
        if (dep_amt%100==0) | (dep_amt%500==0):
            print("Deposited Successfully")
            print("Available Balance= ",(bal_amt+dep_amt))
        else:
            dep_amt=int(input("Multiples of 100 and 500 only
Allowed: "))
            if (dep_amt % 100 == 0) | (dep_amt % 500 == 0):
                print("Deposited Successfully")
                print("Available Balance= ", (bal_amt + dep_amt))
    case default:
        print("Enter valid choice!!")
```



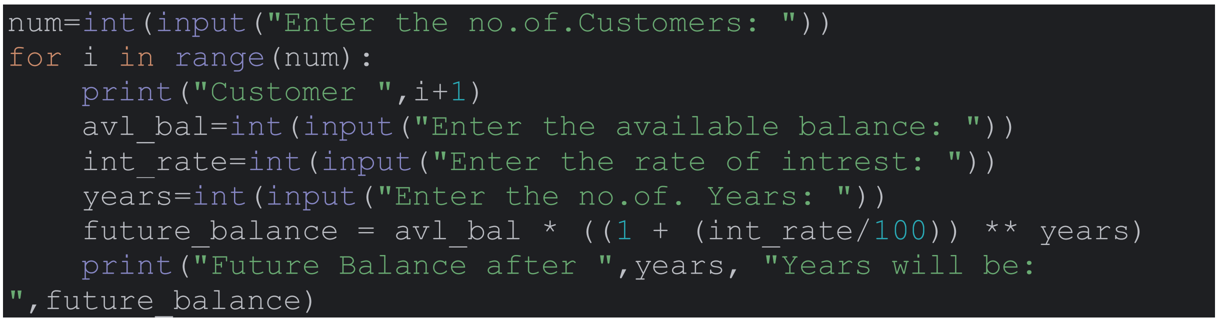
```
Enter the balance Amount: 30000
Enter the number :
 1. Check Balance
 2. Withdrawl
 3. Deposit
 2
Enter the amount to be withdrawn:3000
Please take your cash
Available Balance:  27000
```

Task 3: Loop Structures

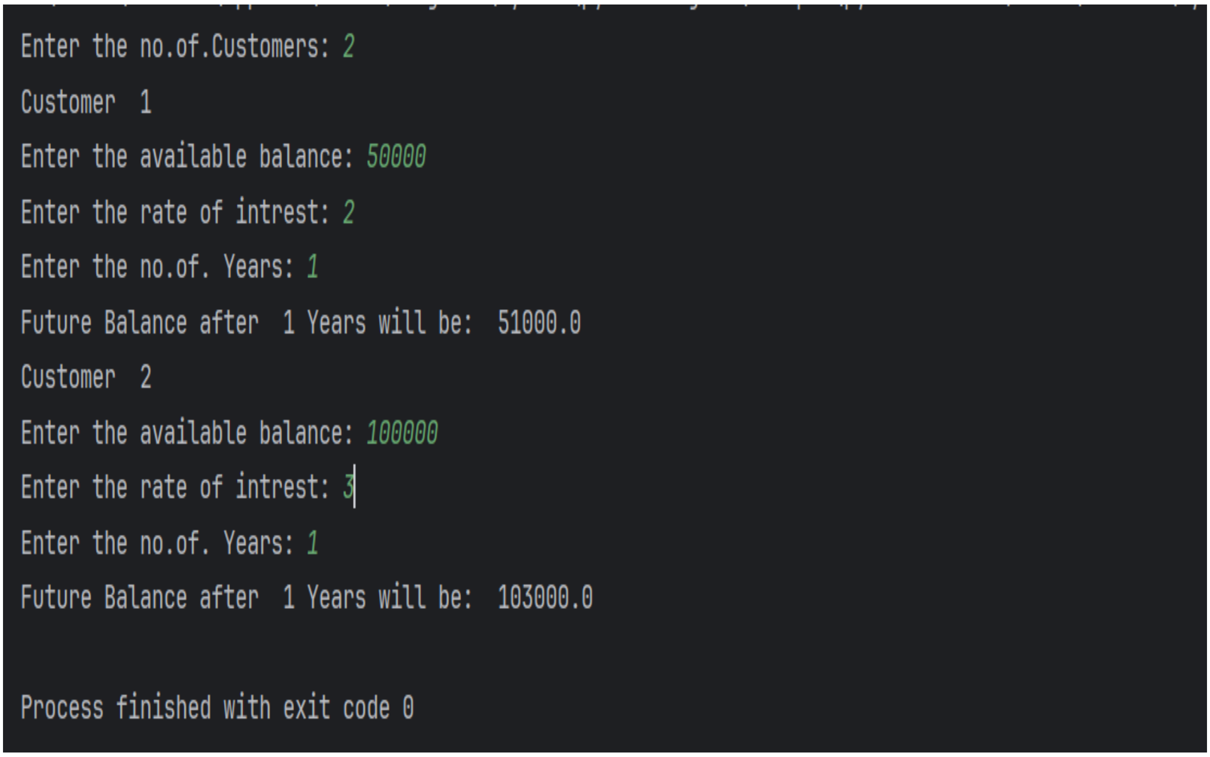
You are responsible for calculating compound interest on savings accounts for bank customers. You need to calculate the future balance for each customer's savings account after a certain number of years.

Tasks:

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



```
num=int(input("Enter the no.of.Customers: "))
for i in range(num):
    print("Customer ",i+1)
    avl_bal=int(input("Enter the available balance: "))
    int_rate=int(input("Enter the rate of intrest: "))
    years=int(input("Enter the no.of. Years: "))
    future_balance = avl_bal * ((1 + (int_rate/100)) ** years)
    print("Future Balance after ",years, "Years will be:
",future_balance)
```



```
Enter the no.of.Customers: 2

Customer  1

Enter the available balance: 50000

Enter the rate of intrest: 2

Enter the no.of. Years: 1

Future Balance after  1 Years will be:  51000.0

Customer  2

Enter the available balance: 100000

Enter the rate of intrest: 3

Enter the no.of. Years: 1

Future Balance after  1 Years will be:  103000.0


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.

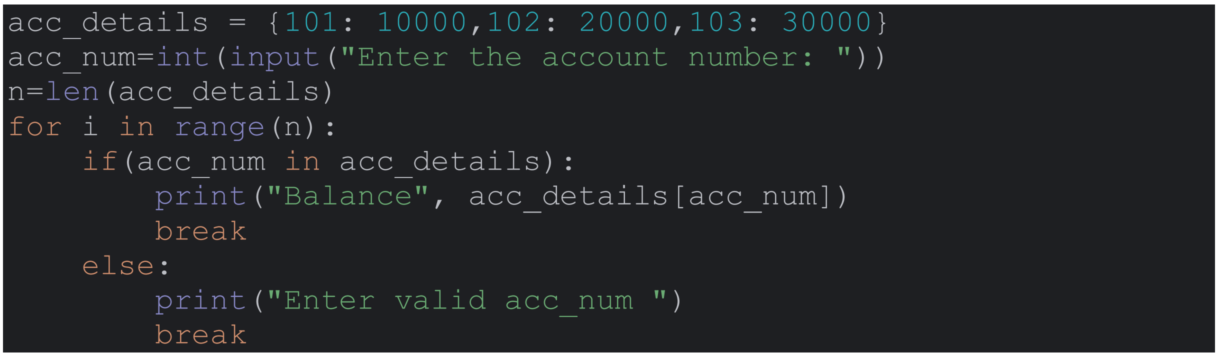
Tasks:

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

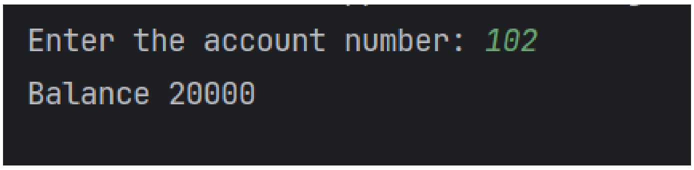
2. Use a loop (e.g., while loop) to repeatedly ask the user for their account number and balance until they enter a valid account number.

3. Validate the account number entered by the user.

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



```python
acc_details = {101: 10000,102: 20000,103: 30000}
acc_num=int(input("Enter the account number: "))
n=len(acc_details)
for i in range(n):
    if(acc_num in acc_details):
        print("Balance", acc_details[acc_num])
        break
    else:
        print("Enter valid acc_num ")
        break
```

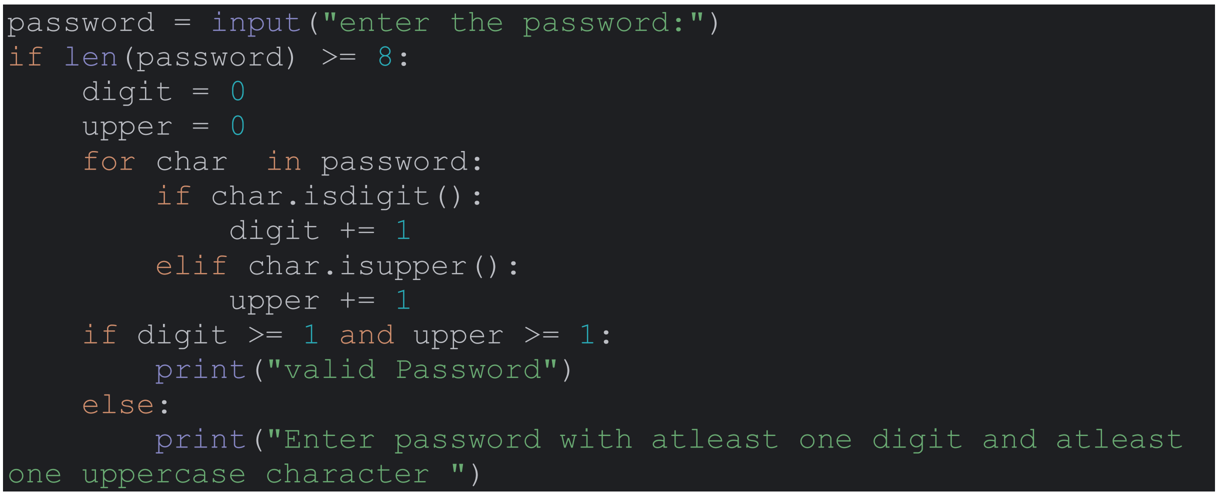


```
Enter the account number: 102
Balance 20000
```

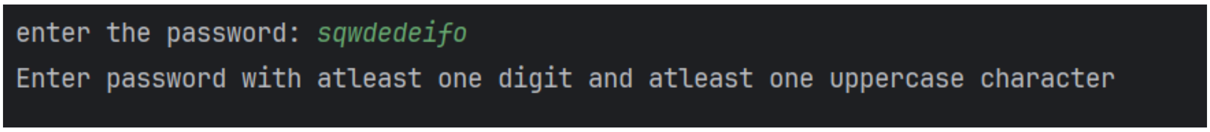
Task 5: Password Validation

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:

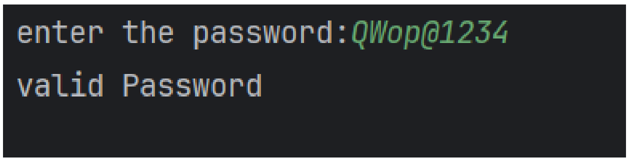
- The password must be at least 8 characters long.
- It must contain at least one uppercase letter.
- It must contain at least one digit.
- Display appropriate messages to indicate whether their password is valid or not.



```python
password = input("enter the password:")
if len(password) >= 8:
    digit = 0
    upper = 0
    for char  in password:
        if char.isdigit():
            digit += 1
        elif char.isupper():
            upper += 1
    if digit >= 1 and upper >= 1:
        print("valid Password")
    else:
        print("Enter password with atleast one digit and atleast
one uppercase character ")
```



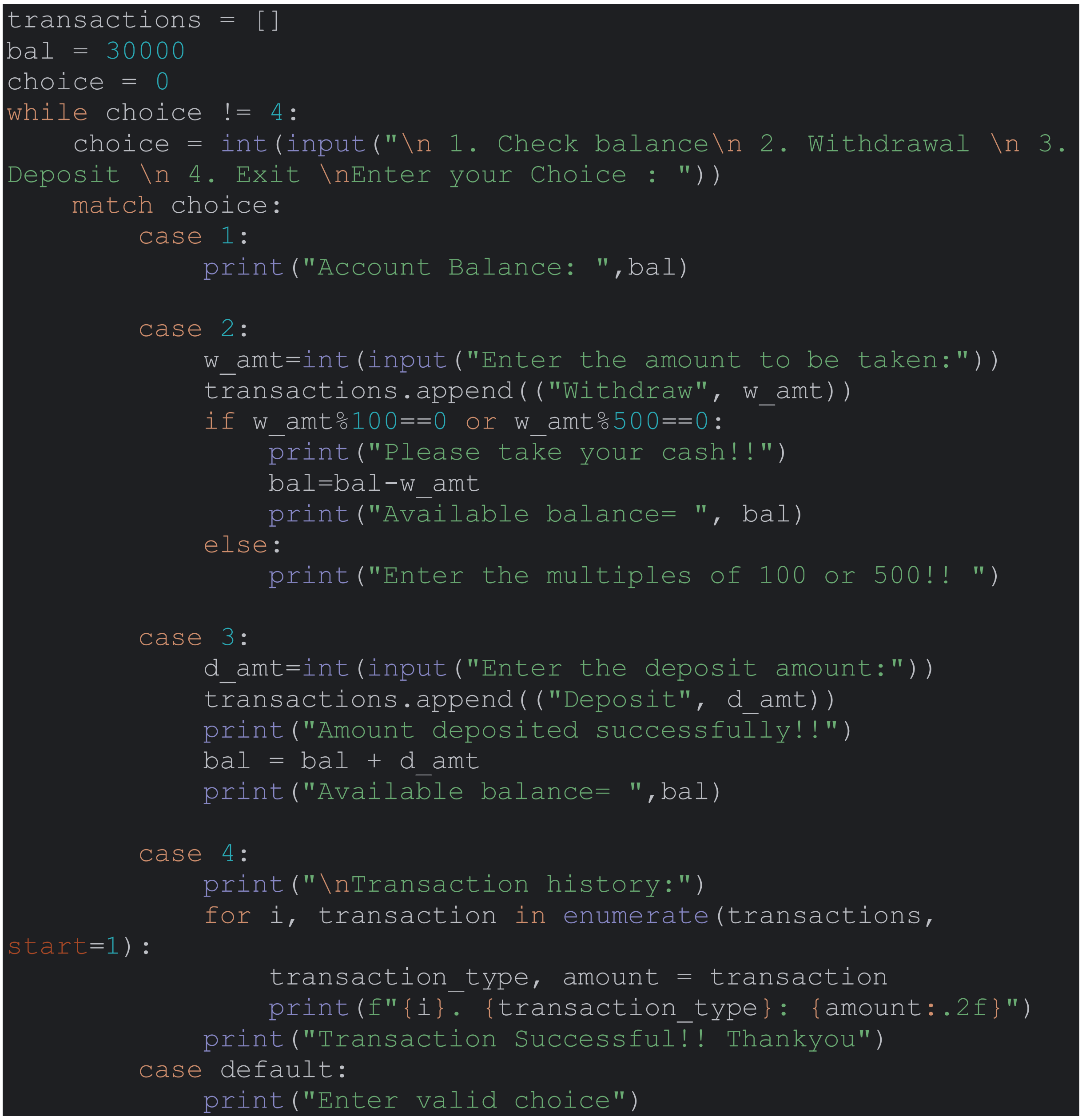
```
enter the password: sqwdedeifo
Enter password with atleast one digit and atleast one uppercase character
```



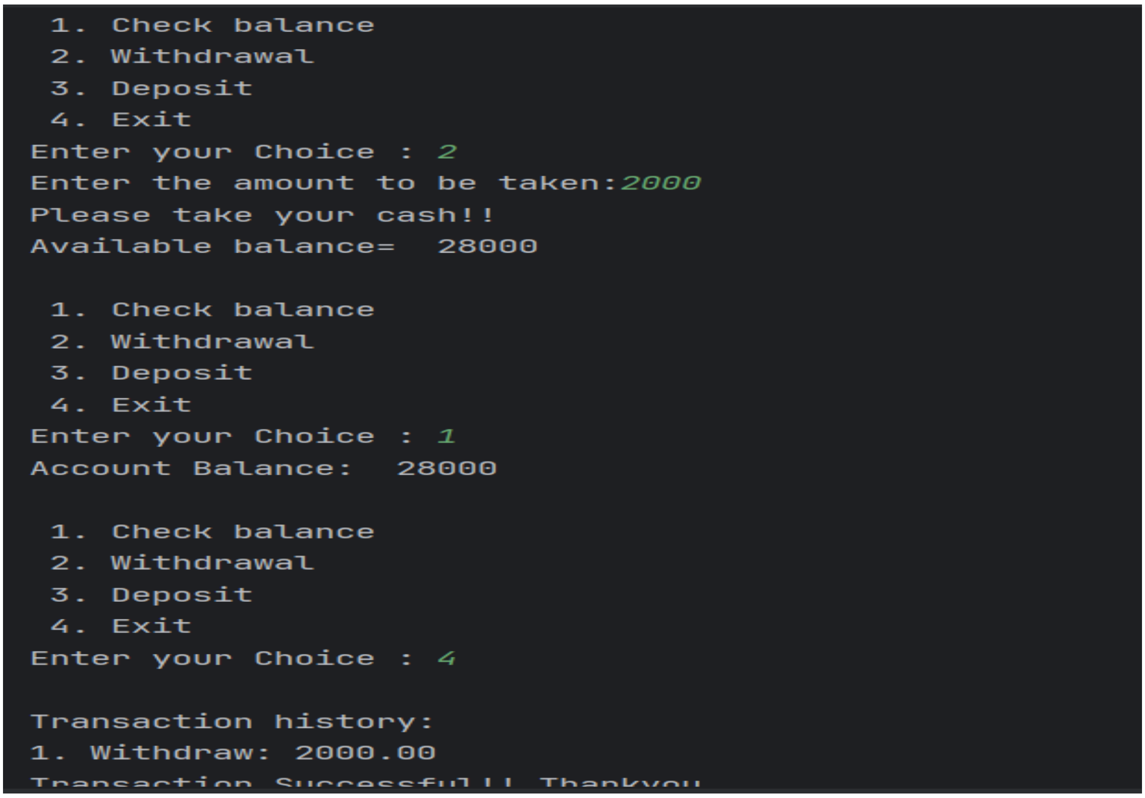
```
enter the password:QWop@1234
valid Password
```

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. Display the transaction history upon exit using looping statements.



```
transactions = []
bal = 30000
choice = 0
while choice != 4:
    choice = int(input("\n 1. Check balance\n 2. Withdrawal \n 3. Deposit \n 4. Exit \nEnter your Choice : "))
    match choice:
        case 1:
            print("Account Balance: ",bal)

        case 2:
            w_amt=int(input("Enter the amount to be taken:"))
            transactions.append(("Withdraw", w_amt))
            if w_amt%100==0 or w_amt%500==0:
                print("Please take your cash!!")
                bal=bal-w_amt
                print("Available balance= ", bal)
            else:
                print("Enter the multiples of 100 or 500!! ")

        case 3:
            d_amt=int(input("Enter the deposit amount:"))
            transactions.append(("Deposit", d_amt))
            print("Amount deposited successfully!!")
            bal = bal + d_amt
            print("Available balance= ",bal)

        case 4:
            print("\nTransaction history:")
            for i, transaction in enumerate(transactions, start=1):
                transaction_type, amount = transaction
                print(f"{i}. {transaction_type}: {amount:.2f}")
            print("Transaction Successful!! Thankyou")
        case default:
            print("Enter valid choice")
```



```
 1. Check balance
 2. Withdrawal
 3. Deposit
 4. Exit
Enter your Choice : 2
Enter the amount to be taken:2000
Please take your cash!!
Available balance=  28000

 1. Check balance
 2. Withdrawal
 3. Deposit
 4. Exit
Enter your Choice : 1
Account Balance:  28000

 1. Check balance
 2. Withdrawal
 3. Deposit
 4. Exit
Enter your Choice : 4

Transaction history:
1. Withdraw: 2000.00
Transaction Successful!! Thankyou
```

OOPS, Collections and Exception Handling

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

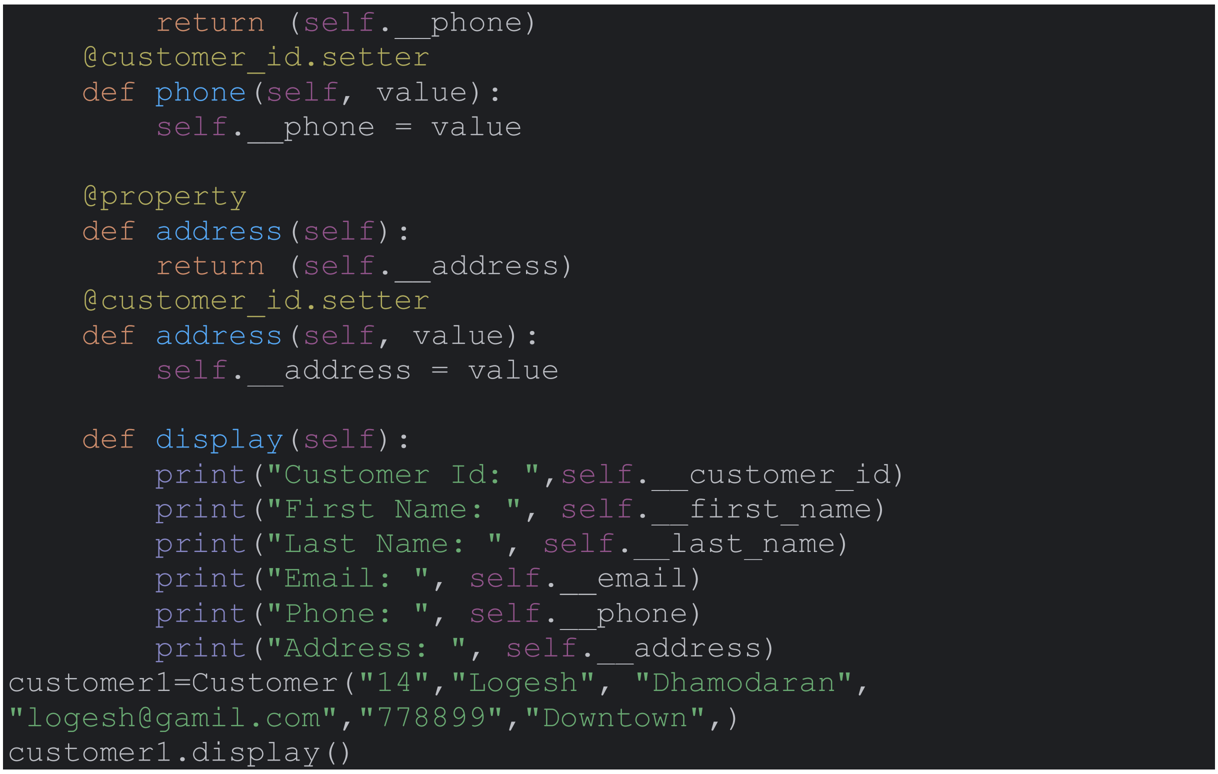
    - Attributes

        ○ Customer ID ○ First Name ○ Last Name ○ Email Address ○ Phone Number ○ Address

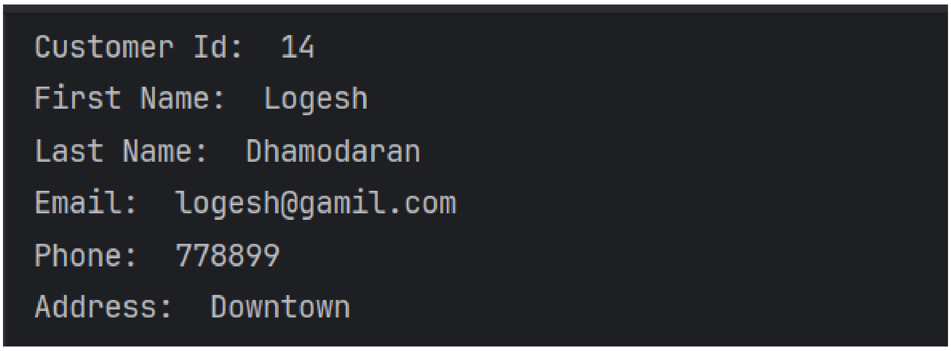
Constructor and Methods

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

```
class Customer:
    def __init__(self,customer_id,first_name,last_name,email,phone,address):
        self.__customer_id=customer_id
        self.__first_name=first_name
        self.__last_name=last_name
        self.__email=email
        self.__phone=phone
        self.__address=address
    @property
    def customer_id(self):
        return self.__customer_id
    @customer_id.setter
    def customer_id(self,value):
        self.__customer_id=value

    @property
    def first_name(self):
        return self.__first_name
    @customer_id.setter
    def first_name(self, value):
        self.__first_name = value

    @property
    def last_name(self):
        return self.__last_name
    @customer_id.setter
    def last_name(self, value):
        self.__last_name = value

    @property
    def email(self):
        return (self.__email)
    @customer_id.setter
    def email(self, value):
        self.__email = value

    @property
    def phone(self):
```



```
        return (self.__phone)
    @customer_id.setter
    def phone(self, value):
        self.__phone = value

    @property
    def address(self):
        return (self.__address)
    @customer_id.setter
    def address(self, value):
        self.__address = value

    def display(self):
        print("Customer Id: ",self.__customer_id)
        print("First Name: ", self.__first_name)
        print("Last Name: ", self.__last_name)
        print("Email: ", self.__email)
        print("Phone: ", self.__phone)
        print("Address: ", self.__address)
customer1=Customer("14","Logesh", "Dhamodaran",
"logesh@gamil.com","778899","Downtown",)
customer1.display()
```



```
Customer Id:  14
First Name:  Logesh
Last Name:  Dhamodaran
Email:  logesh@gamil.com
Phone:  778899
Address:  Downtown
```

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

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

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

```
class Accounts:
    def __init__(self,acc_num,acc_type,acc_bal):
        self.__acc_num=acc_num
        self.__acc_type=acc_type
        self.__acc_bal=acc_bal
    @property
    def acc_num(self):
        return self.__acc_num
    @acc_num.setter
    def acc_num(self,value):
        self.__acc_num=value

    @property
    def acc_type(self):
        return self.__acc_type
    @acc_type.setter
    def acc_type(self, value):
        self.__acc_type = value

    @property
    def acc_bal(self):
        return self.__acc_bal
    @acc_bal.setter
    def acc_bal(self,value):
        self.__acc_bal = value

    def display(self):
        print("Account Number: ",self.__acc_num)
        print("Account type: ",self.__acc_type)
        print("Available Balance: ",self.__acc_bal)

    def deposit(self,amount):
        self.__acc_bal=self.__acc_bal+amount
        print("Available balance after deposit = 
",self.__acc_bal)

    def withdrawl(self,amount):
        self.__acc_bal=self.acc_bal-amount
        print("Available balance after withdrawl = ",
self.__acc_bal)

    def calculate_intrest(self):
        int_rate=4.5/100
        int_amt=self.__acc_bal * int_rate
        print("Intrest Amount = ",int_amt)
```

```
class Bank:
    pass

Accounts1=Accounts(15,"Savings",50000)
Accounts1.display()
Accounts1.deposit(5000)
Accounts1.withdrawl(3000)
```

Task 8: Inheritance and polymorphism

1. 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.

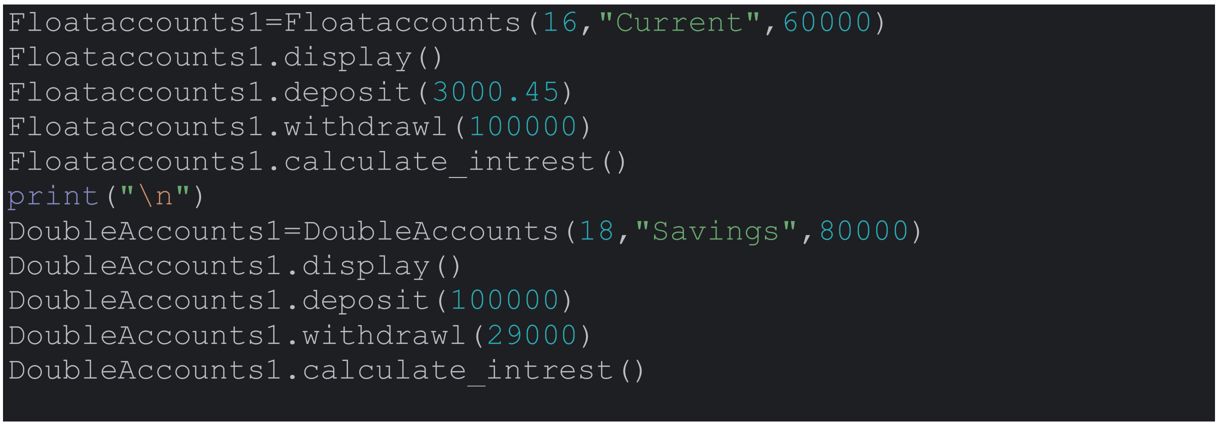
deposit(amount: int): Deposit the specified amount into the account.

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

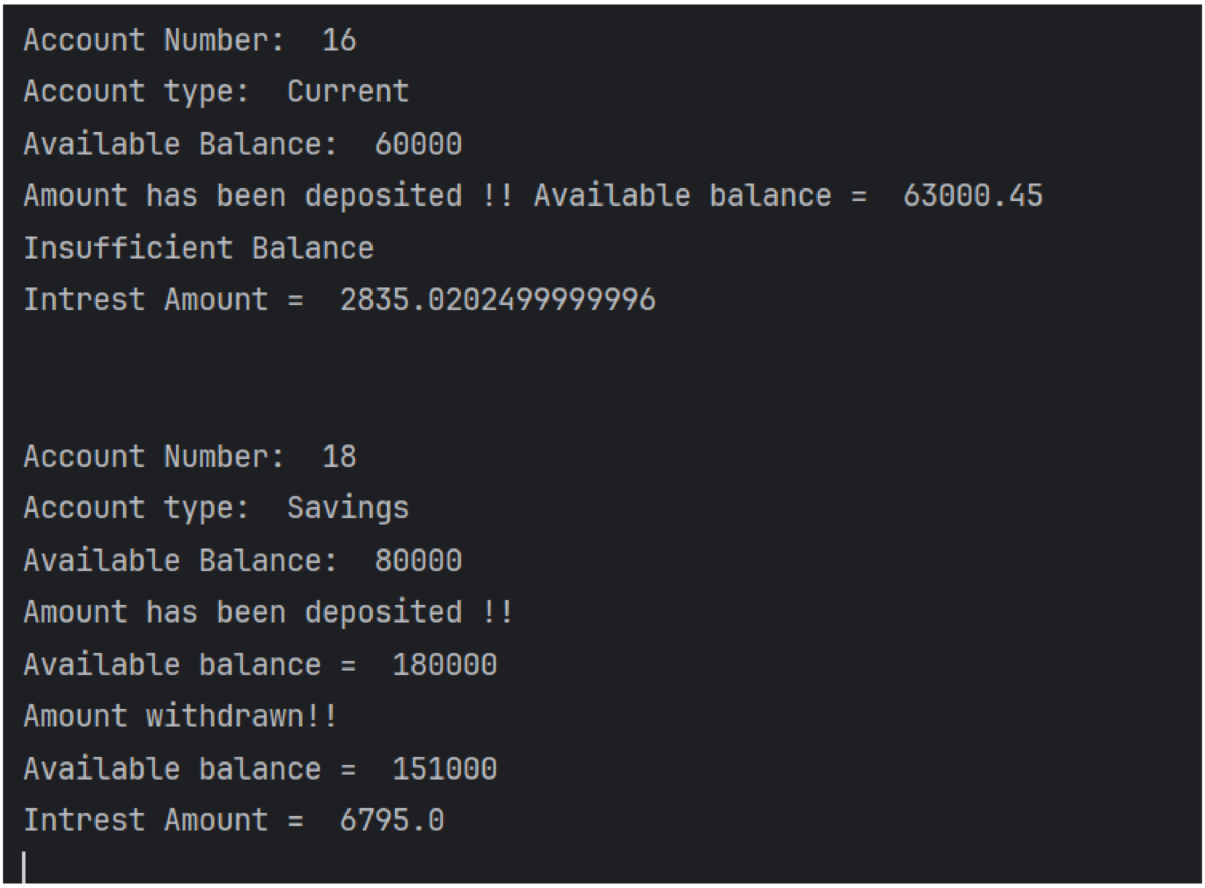
deposit(amount: double): Deposit the specified amount into the account.

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

```
class Floataccounts(Accounts):
    def deposit(self,amount):
        self.acc_bal=amount+self.acc_bal
        print("Amount has been deposited !! Available balance =
",self.acc_bal)
    def withdrawl(self,amount):
        if amount > self.acc_bal:
            print("Insufficient Balance")
        else:
            self.acc_bal=self.acc_bal-amount
            print("Amount withdrawn!! Available balance =
",self.acc_bal)

class DoubleAccounts(Accounts):
    def deposit(self,amount):
        self.acc_bal=amount+self.acc_bal
        print("Amount has been deposited !!" )
        print("Available balance = ",self.acc_bal)
    def withdrawl(self,amount):
        if amount > self.acc_bal:
            print("Insufficient Balance")
        else:
            self.acc_bal=self.acc_bal-amount
            print("Amount withdrawn!! ")
            print("Available balance = ", self.acc_bal)
```



```
Floataccounts1=Floataccounts(16,"Current",60000)
Floataccounts1.display()
Floataccounts1.deposit(3000.45)
Floataccounts1.withdrawl(100000)
Floataccounts1.calculate_intrest()
print("\n")
DoubleAccounts1=DoubleAccounts(18,"Savings",80000)
DoubleAccounts1.display()
DoubleAccounts1.deposit(100000)
DoubleAccounts1.withdrawl(29000)
DoubleAccounts1.calculate_intrest()
```

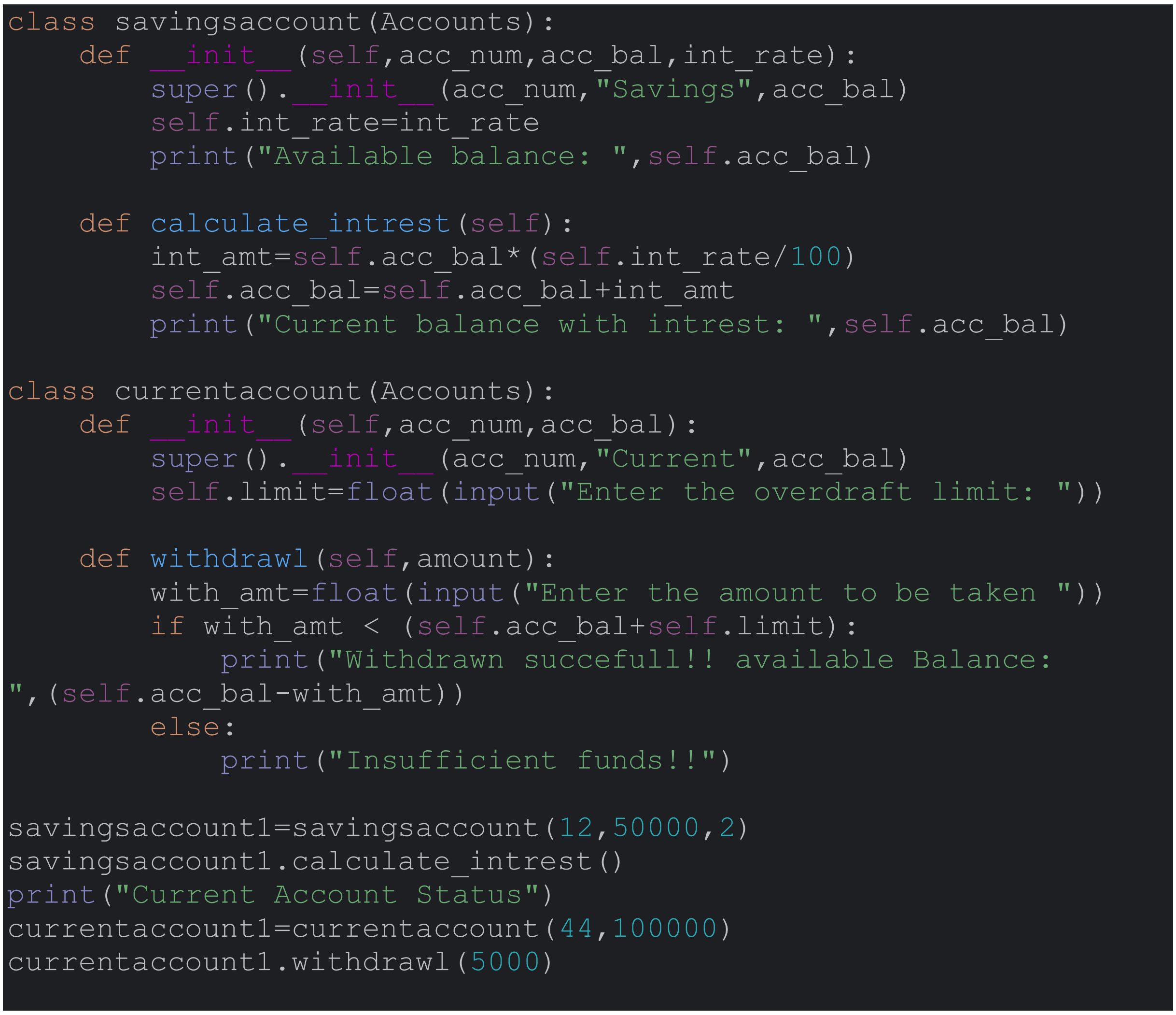


```
Account Number:  16
Account type:  Current
Available Balance:  60000
Amount has been deposited !! Available balance =  63000.45
Insufficient Balance
Intrest Amount =  2835.0202499999996


Account Number:  18
Account type:  Savings
Available Balance:  80000
Amount has been deposited !!
Available balance =  180000
Amount withdrawn!!
Available balance =  151000
Intrest Amount =  6795.0
```

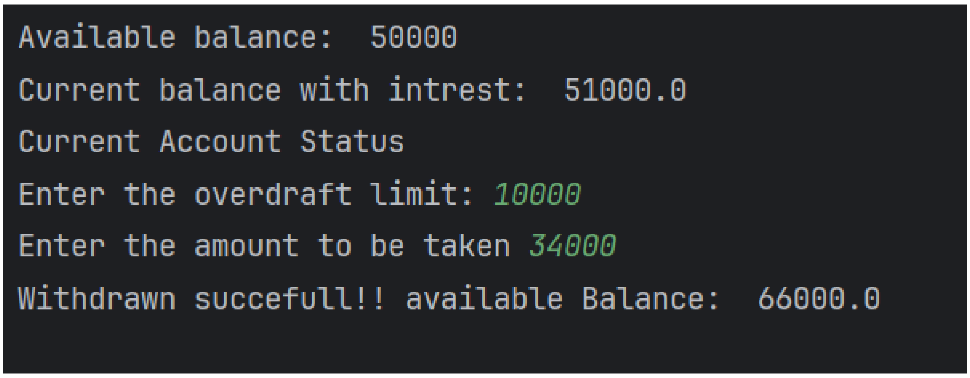
2. Create Subclasses for Specific Account Types

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

○ **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).



```
class savingsaccount(Accounts):
    def __init__(self,acc_num,acc_bal,int_rate):
        super().__init__(acc_num,"Savings",acc_bal)
        self.int_rate=int_rate
        print("Available balance: ",self.acc_bal)

    def calculate_intrest(self):
        int_amt=self.acc_bal*(self.int_rate/100)
        self.acc_bal=self.acc_bal+int_amt
        print("Current balance with intrest: ",self.acc_bal)

class currentaccount(Accounts):
    def __init__(self,acc_num,acc_bal):
        super().__init__(acc_num,"Current",acc_bal)
        self.limit=float(input("Enter the overdraft limit: "))

    def withdrawl(self,amount):
        with_amt=float(input("Enter the amount to be taken "))
        if with_amt < (self.acc_bal+self.limit):
            print("Withdrawn succefull!! available Balance: 
",(self.acc_bal-with_amt))
        else:
            print("Insufficient funds!!")

savingsaccount1=savingsaccount(12,50000,2)
savingsaccount1.calculate_intrest()
print("Current Account Status")
currentaccount1=currentaccount(44,100000)
currentaccount1.withdrawl(5000)
```

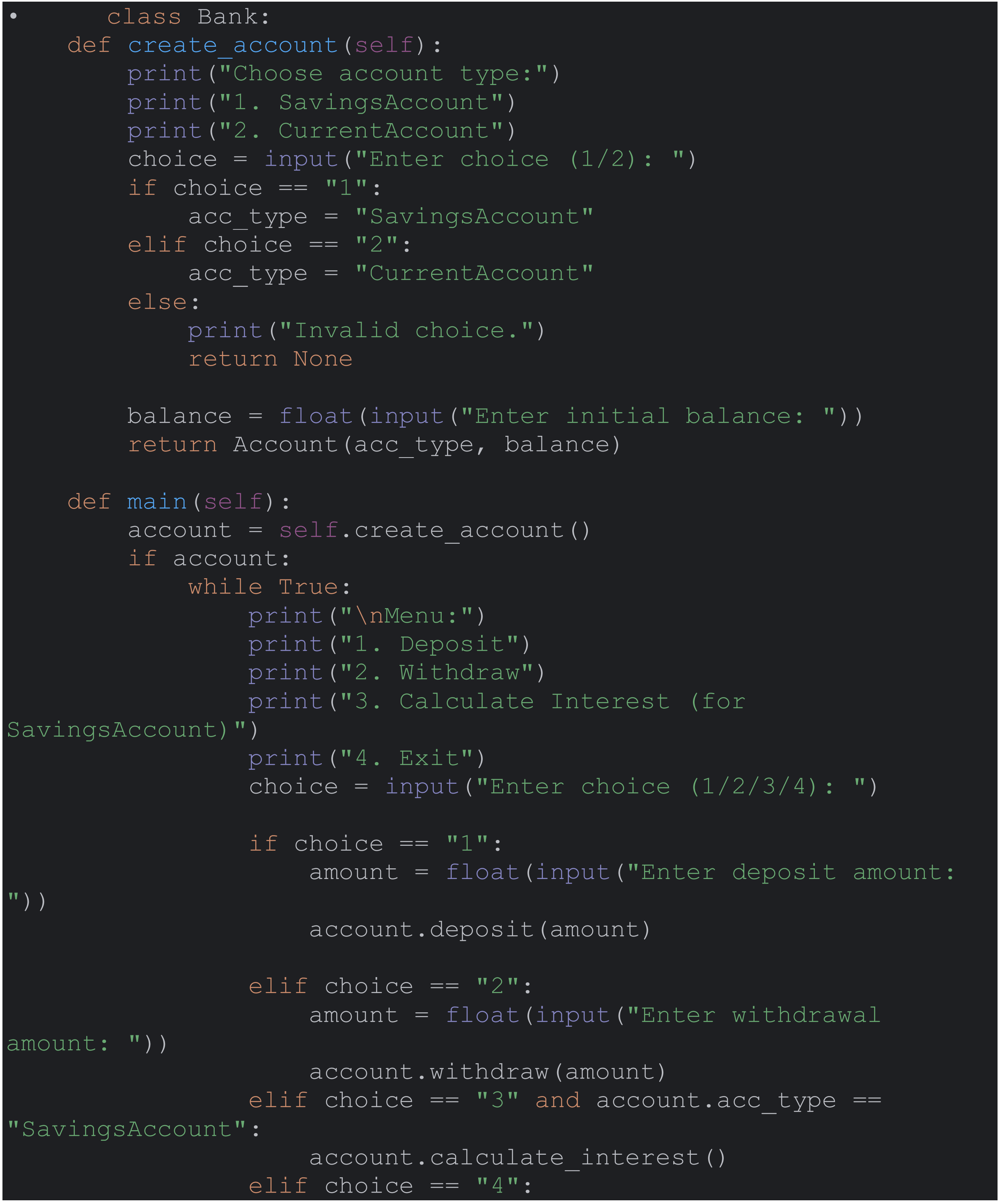


```
Available balance:  50000
Current balance with intrest:  51000.0
Current Account Status
Enter the overdraft limit: 10000
Enter the amount to be taken 34000
Withdrawn succefull!! available Balance:  66000.0
```

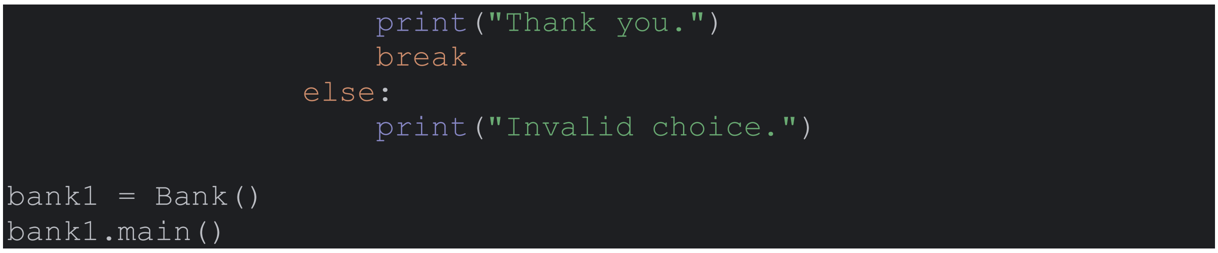
3. Create a **Bank** class to represent the banking system. Perform the following operation in main method: Display menu for user to create object for account class by calling parameter constructor. Menu should display options `SavingsAccount` and `CurrentAccount`. user can choose any one option to create account. use switch case for implementation.

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

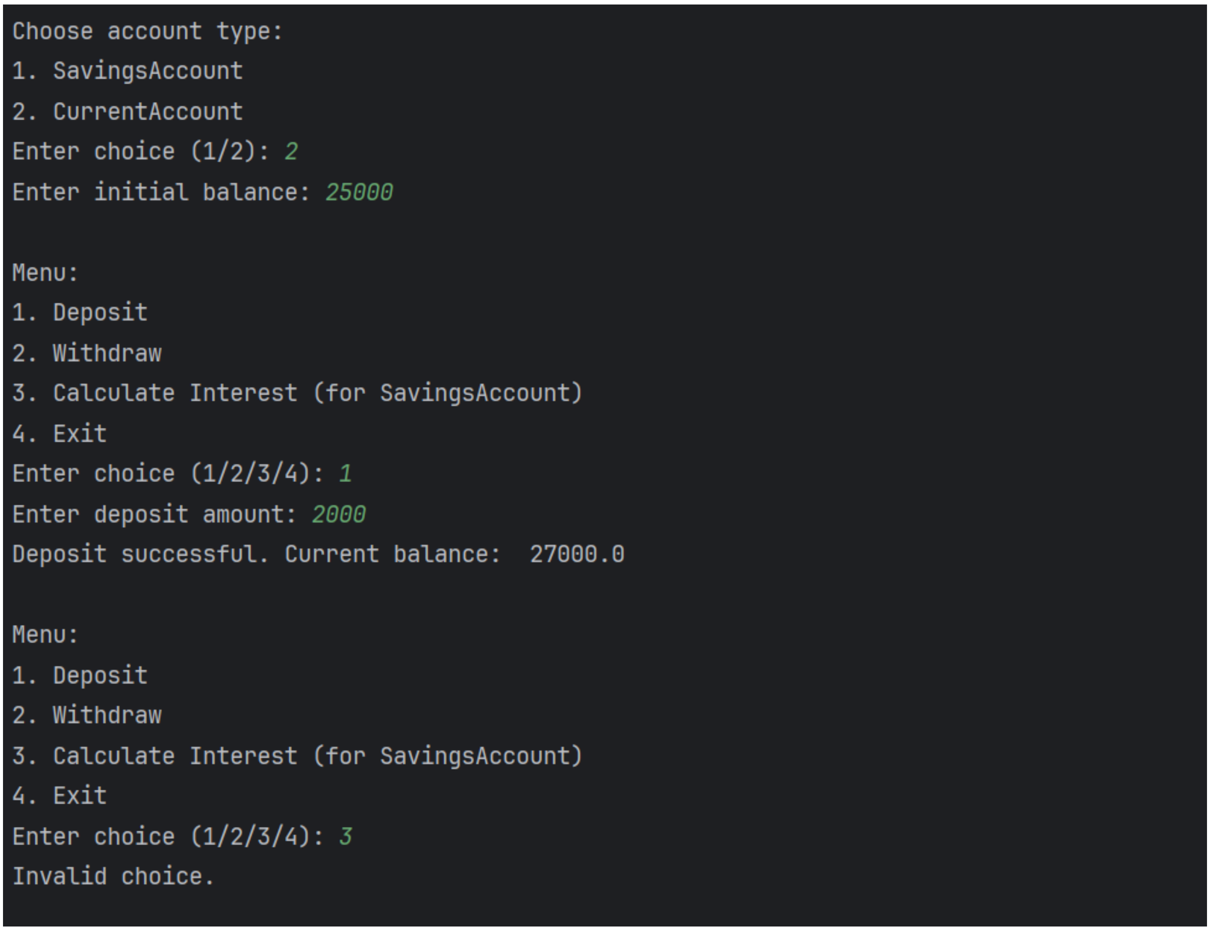
withdraw(amount: float): Withdraw the specified amount from the account. For saving account withdraw amount only if there is sufficient fund else display insufficient balance.
For Current Account withdraw limit can exceed the available balance and should not exceed the overdraft limit.
calculate_interest():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



```
•     class Bank:
  def create_account(self):
    print("Choose account type:")
    print("1. SavingsAccount")
    print("2. CurrentAccount")
    choice = input("Enter choice (1/2): ")
    if choice == "1":
      acc_type = "SavingsAccount"
    elif choice == "2":
      acc_type = "CurrentAccount"
    else:
      print("Invalid choice.")
      return None

    balance = float(input("Enter initial balance: "))
    return Account(acc_type, balance)

  def main(self):
    account = self.create_account()
    if account:
      while True:
        print("\nMenu:")
        print("1. Deposit")
        print("2. Withdraw")
        print("3. Calculate Interest (for
SavingsAccount)")
        print("4. Exit")
        choice = input("Enter choice (1/2/3/4): ")

        if choice == "1":
          amount = float(input("Enter deposit amount:
"))
          account.deposit(amount)

        elif choice == "2":
          amount = float(input("Enter withdrawal
amount: "))
          account.withdraw(amount)
        elif choice == "3" and account.acc_type ==
"SavingsAccount":
          account.calculate_interest()
        elif choice == "4":
```



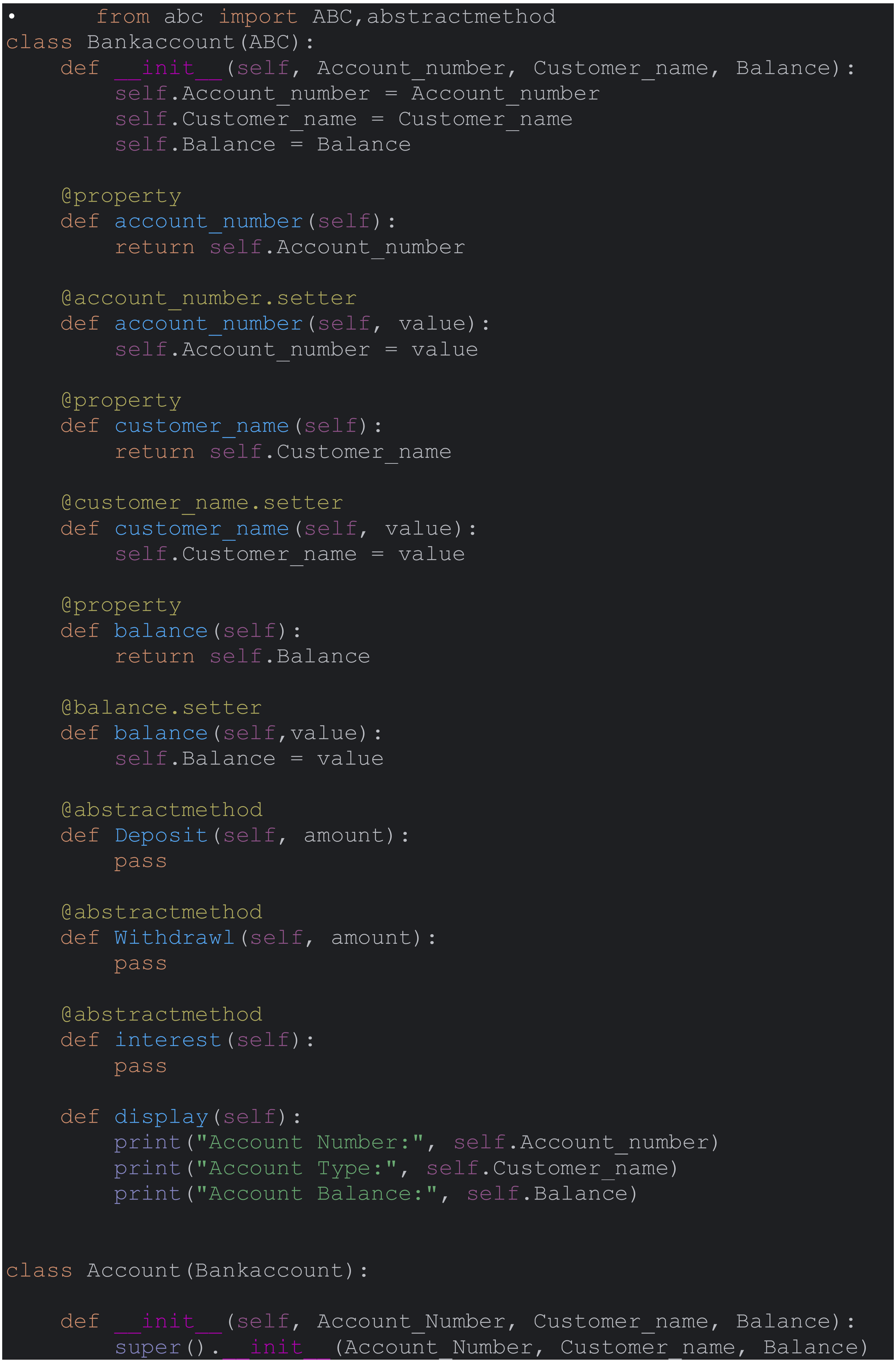
```
                    print("Thank you.")
                    break
                else:
                    print("Invalid choice.")

bank1 = Bank()
bank1.main()
```



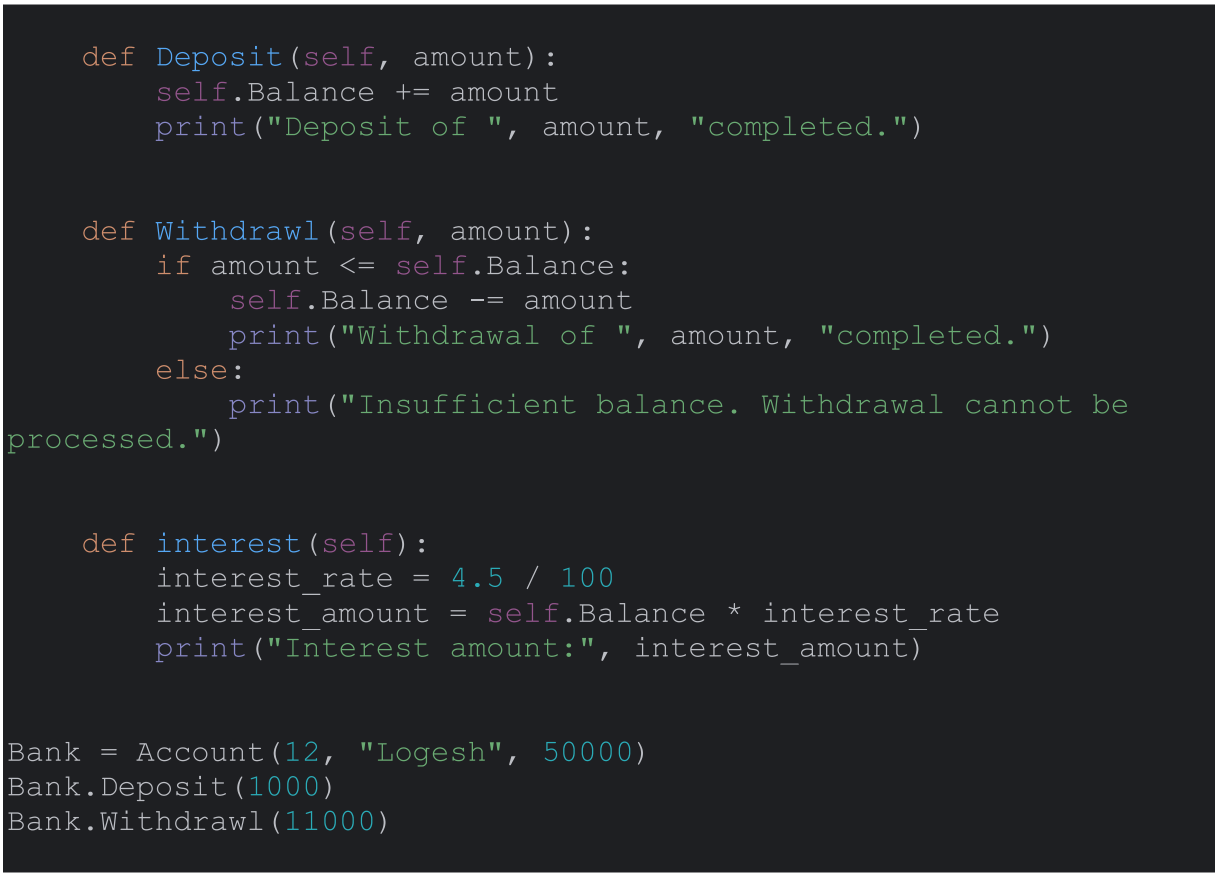
```
Choose account type:
1. SavingsAccount
2. CurrentAccount
Enter choice (1/2): 2
Enter initial balance: 25000

Menu:
1. Deposit
2. Withdraw
3. Calculate Interest (for SavingsAccount)
4. Exit
Enter choice (1/2/3/4): 1
Enter deposit amount: 2000
Deposit successful. Current balance:  27000.0

Menu:
1. Deposit
2. Withdraw
3. Calculate Interest (for SavingsAccount)
4. Exit
Enter choice (1/2/3/4): 3
Invalid choice.
```

Task 9: Abstraction

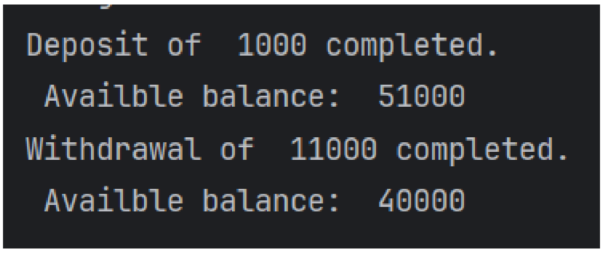
- 1. Create an abstract class BankAccount that represents a generic bank account. It should include the following attributes and methods: Attributes: ○ Account number.
- ○ Customer name.
- ○ Balance.
- Constructors: ○ Implement default constructors and overload the constructor with Account attributes, generate getter and setter, print all information of attribute methods for the attributes.
Abstract methods: ○ **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.



```
•      from abc import ABC,abstractmethod
class Bankaccount(ABC):
    def __init__(self, Account_number, Customer_name, Balance):
        self.Account_number = Account_number
        self.Customer_name = Customer_name
        self.Balance = Balance

    @property
    def account_number(self):
        return self.Account_number

    @account_number.setter
    def account_number(self, value):
        self.Account_number = value

    @property
    def customer_name(self):
        return self.Customer_name

    @customer_name.setter
    def customer_name(self, value):
        self.Customer_name = value

    @property
    def balance(self):
        return self.Balance

    @balance.setter
    def balance(self,value):
        self.Balance = value

    @abstractmethod
    def Deposit(self, amount):
        pass

    @abstractmethod
    def Withdrawl(self, amount):
        pass

    @abstractmethod
    def interest(self):
        pass

    def display(self):
        print("Account Number:", self.Account_number)
        print("Account Type:", self.Customer_name)
        print("Account Balance:", self.Balance)


class Account(Bankaccount):

    def __init__(self, Account_Number, Customer_name, Balance):
        super().__init__(Account_Number, Customer_name, Balance)
```



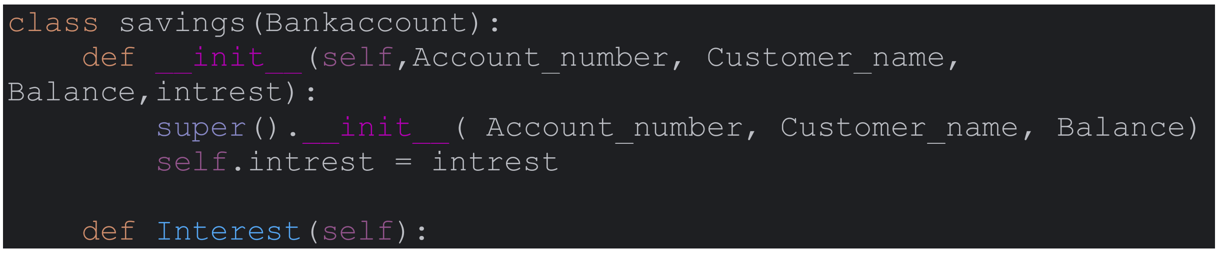
```
    def Deposit(self, amount):
        self.Balance += amount
        print("Deposit of ", amount, "completed.")


    def Withdrawl(self, amount):
        if amount <= self.Balance:
            self.Balance -= amount
            print("Withdrawal of ", amount, "completed.")
        else:
            print("Insufficient balance. Withdrawal cannot be 
processed.")


    def interest(self):
        interest_rate = 4.5 / 100
        interest_amount = self.Balance * interest_rate
        print("Interest amount:", interest_amount)


Bank = Account(12, "Logesh", 50000)
Bank.Deposit(1000)
Bank.Withdrawl(11000)
```



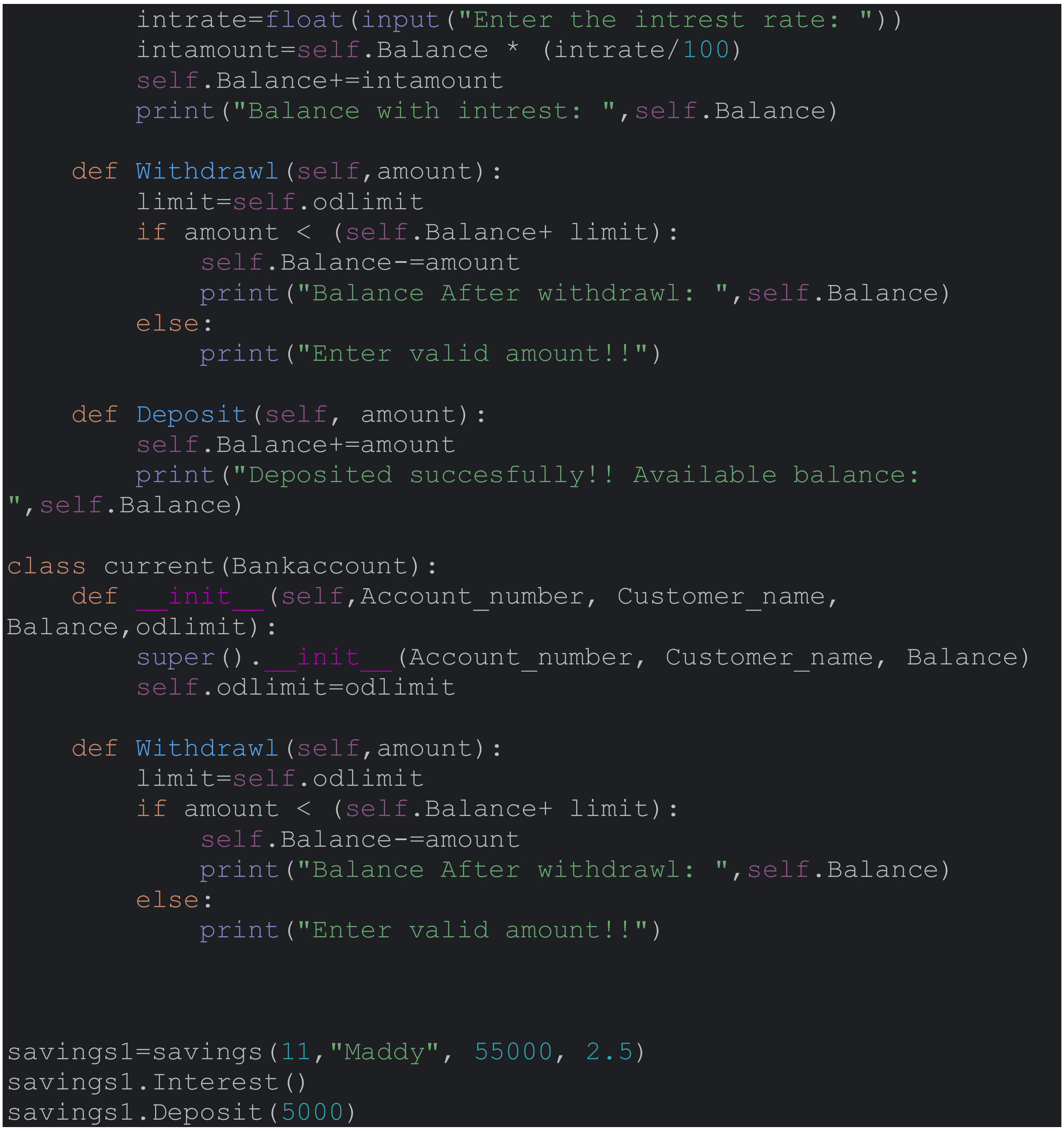
```
Deposit of  1000 completed.
 Availble balance:  51000
Withdrawal of  11000 completed.
 Availble balance:  40000
```

2.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.

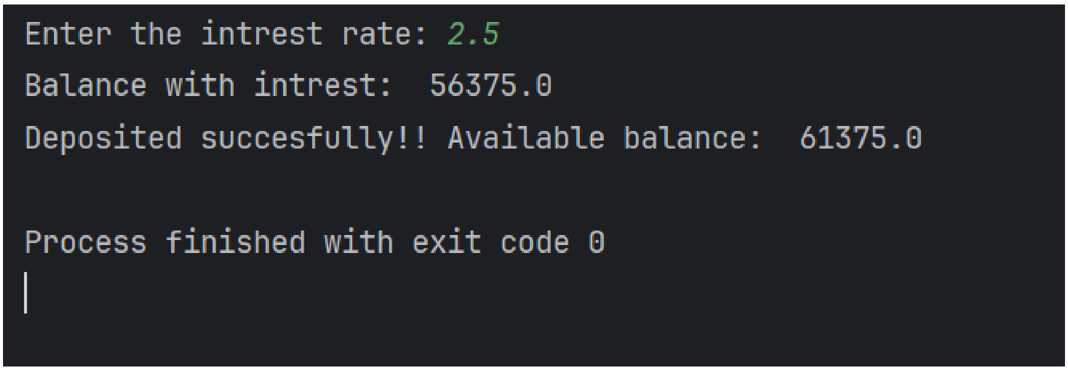
CurrentAccount: A current account with no interest. Implement the withdraw() method to allow overdraft up to a certain limit (configure a constant for the overdraft limit).



```
class savings(Bankaccount):
    def __init__(self,Account_number, Customer_name, 
Balance,intrest):
        super().__init__( Account_number, Customer_name, Balance)
        self.intrest = intrest

    def Interest(self):
```



```
        intrate=float(input("Enter the intrest rate: "))
        intamount=self.Balance * (intrate/100)
        self.Balance+=intamount
        print("Balance with intrest: ",self.Balance)

    def Withdrawl(self,amount):
        limit=self.odlimit
        if amount < (self.Balance+ limit):
            self.Balance-=amount
            print("Balance After withdrawl: ",self.Balance)
        else:
            print("Enter valid amount!!")

    def Deposit(self, amount):
        self.Balance+=amount
        print("Deposited succesfully!! Available balance:
",self.Balance)

class current(Bankaccount):
    def __init__(self,Account_number, Customer_name,
Balance,odlimit):
        super().__init__(Account_number, Customer_name, Balance)
        self.odlimit=odlimit

    def Withdrawl(self,amount):
        limit=self.odlimit
        if amount < (self.Balance+ limit):
            self.Balance-=amount
            print("Balance After withdrawl: ",self.Balance)
        else:
            print("Enter valid amount!!")



savings1=savings(11,"Maddy", 55000, 2.5)
savings1.Interest()
savings1.Deposit(5000)
```

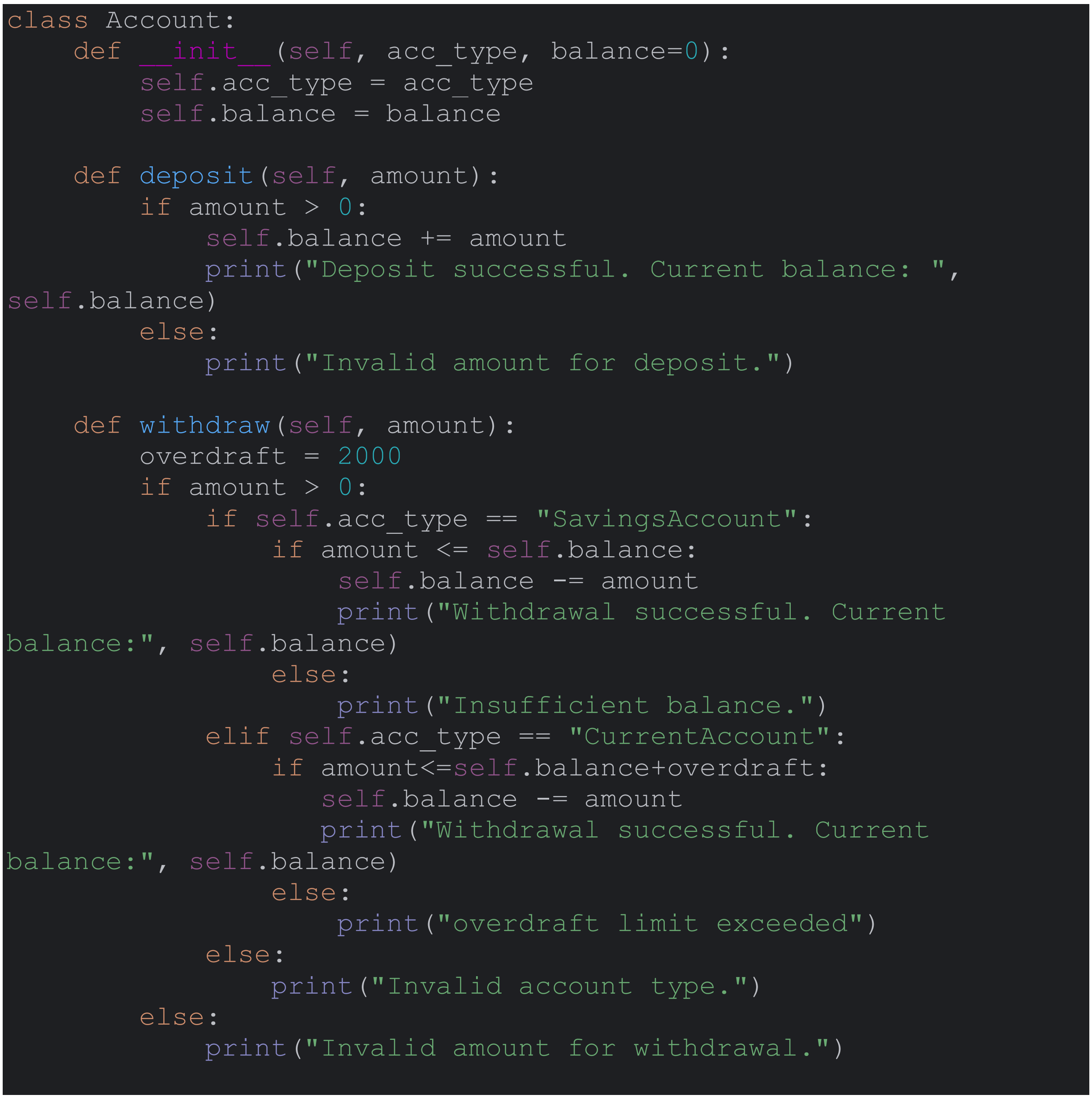


```
Enter the intrest rate: 2.5
Balance with intrest:  56375.0
Deposited succesfully!! Available balance:  61375.0

Process finished with exit code 0
```

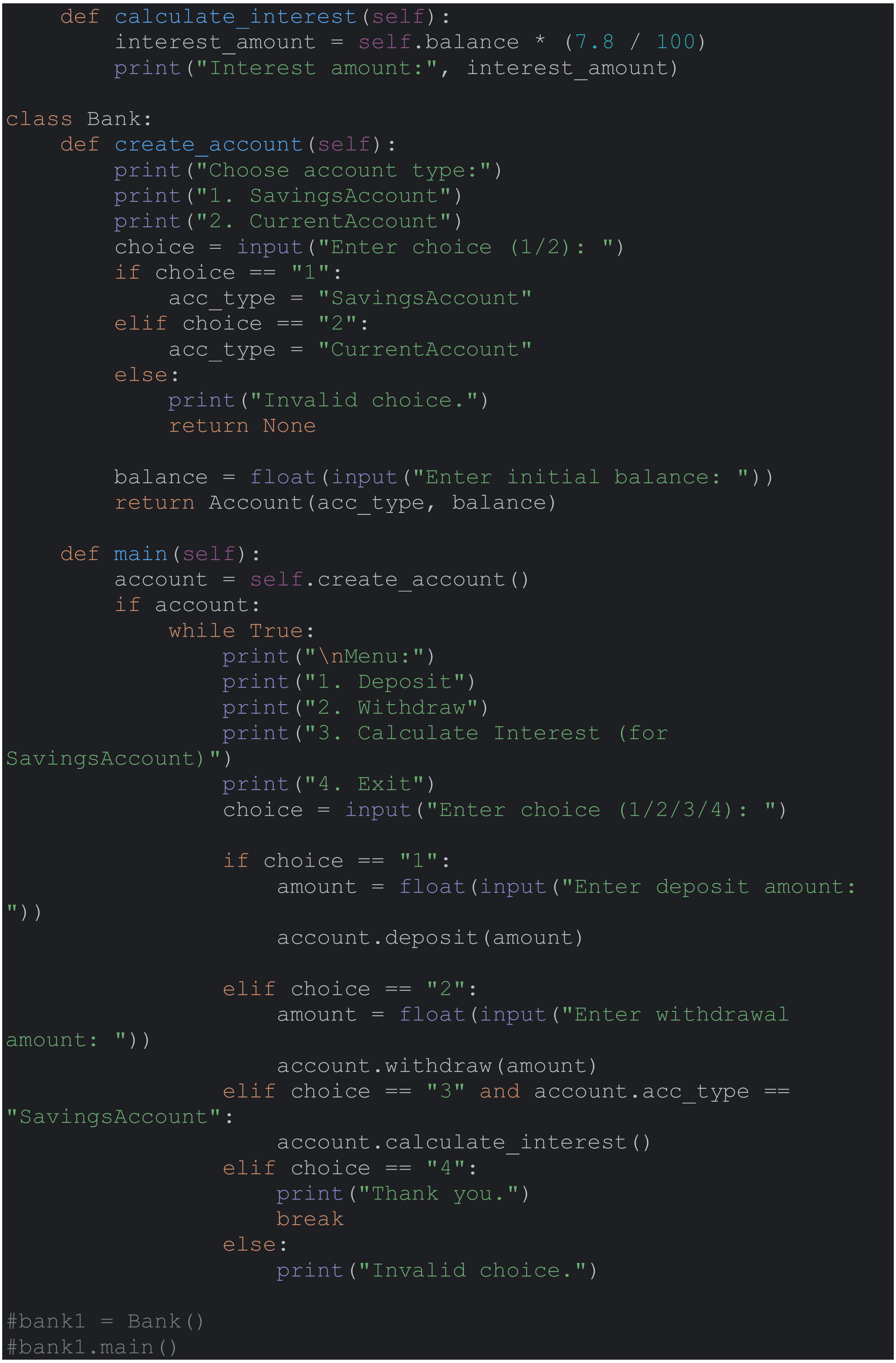
3. Create a Bank class to represent the banking system. Perform the following operation in main method: Display menu for user to create object for account class by calling parameter constructor. Menu should display options `SavingsAccount` and `CurrentAccount`. user can choose any one option to create account. use switch case for implementation. create_account should display sub menu to choose type of accounts. ○ *Hint: Account acc = new SavingsAccount(); or Account acc = new CurrentAccount();*

- 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.

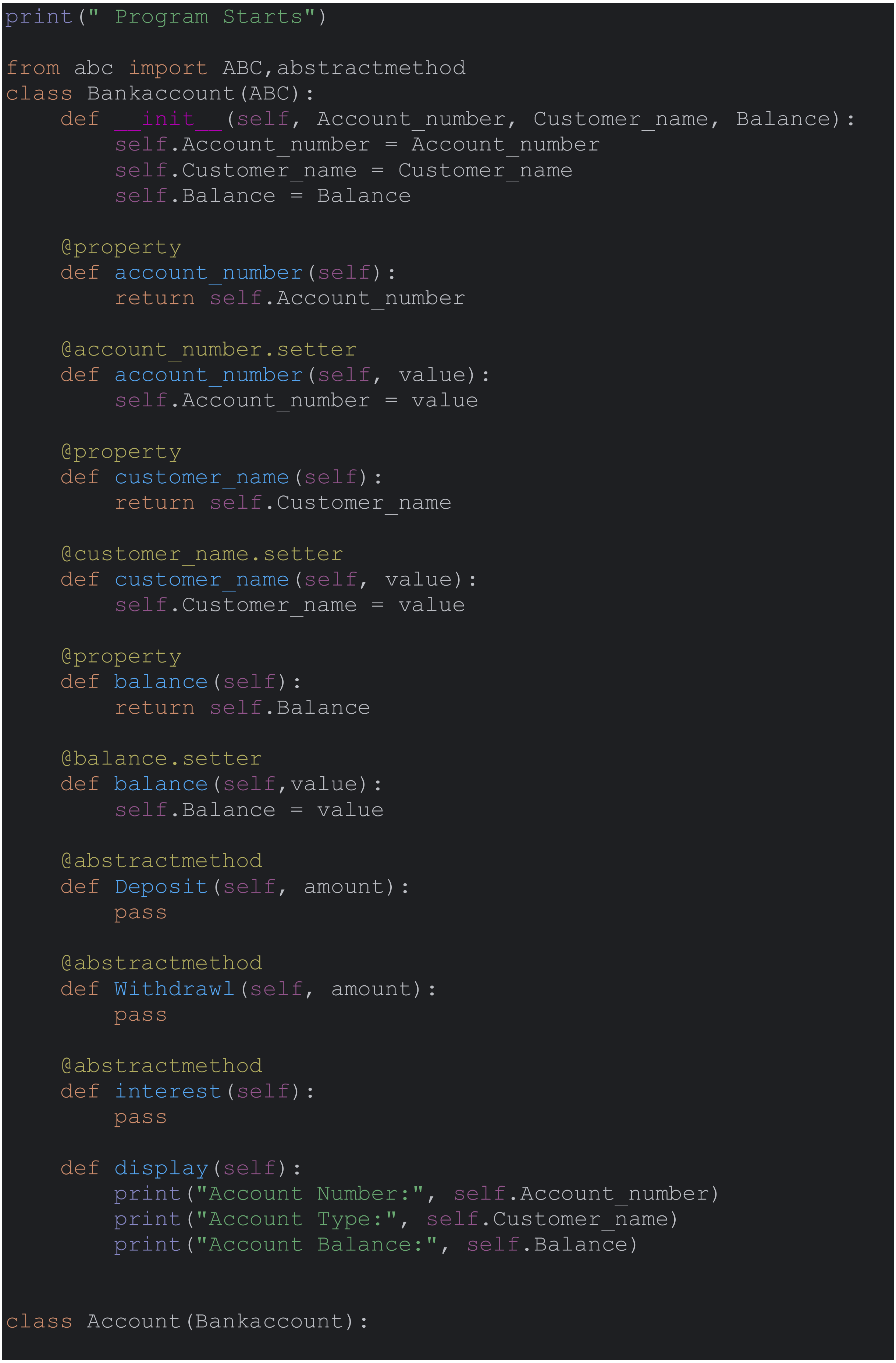
- calculate_interest(): Calculate and add interest to the account balance for savings accounts.



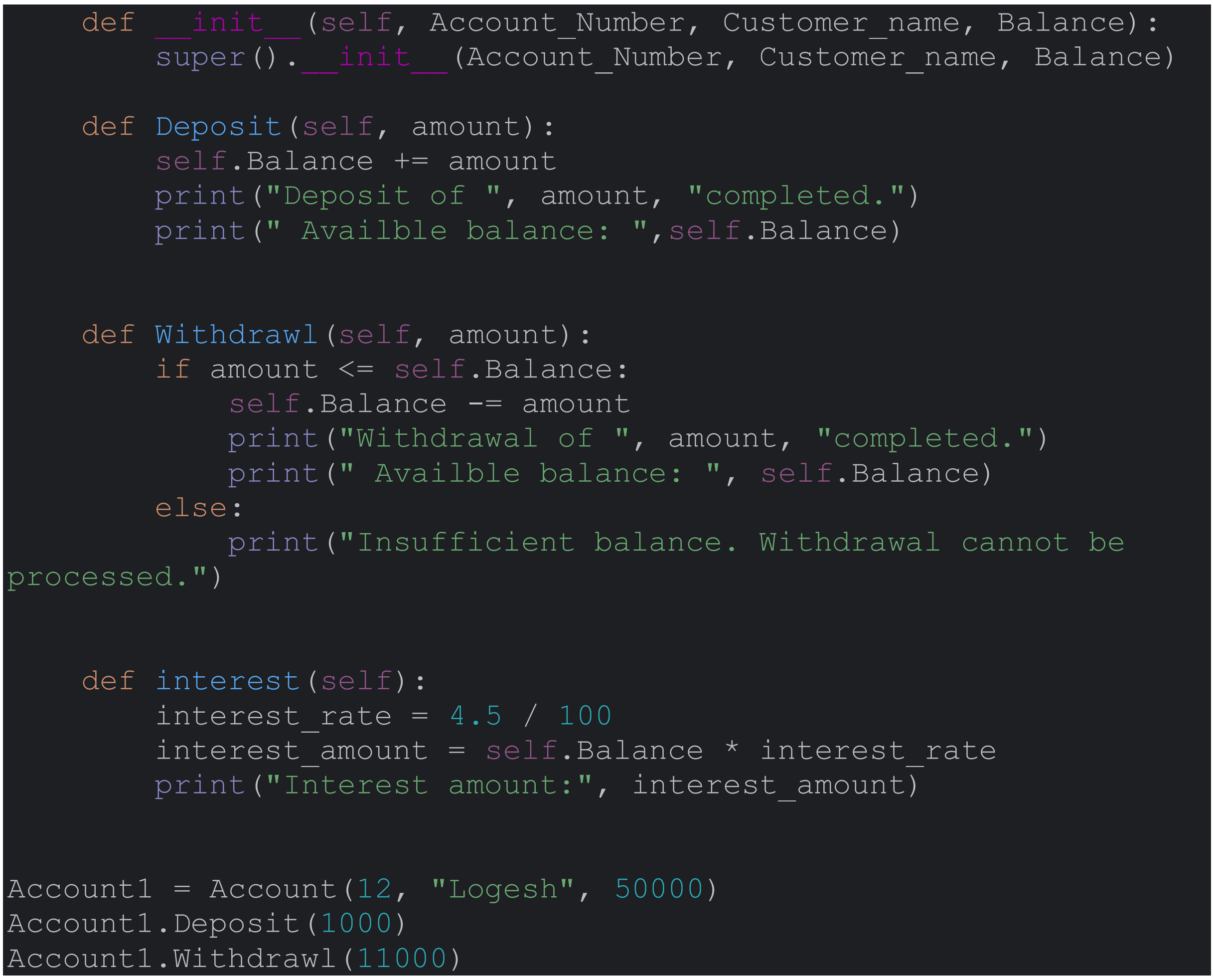
```
class Account:
    def __init__(self, acc_type, balance=0):
        self.acc_type = acc_type
        self.balance = balance

    def deposit(self, amount):
        if amount > 0:
            self.balance += amount
            print("Deposit successful. Current balance: ",
self.balance)
        else:
            print("Invalid amount for deposit.")

    def withdraw(self, amount):
        overdraft = 2000
        if amount > 0:
            if self.acc_type == "SavingsAccount":
                if amount <= self.balance:
                    self.balance -= amount
                    print("Withdrawal successful. Current
balance:", self.balance)
                else:
                    print("Insufficient balance.")
            elif self.acc_type == "CurrentAccount":
                if amount<=self.balance+overdraft:
                   self.balance -= amount
                   print("Withdrawal successful. Current
balance:", self.balance)
                else:
                    print("overdraft limit exceeded")
            else:
                print("Invalid account type.")
        else:
            print("Invalid amount for withdrawal.")
```



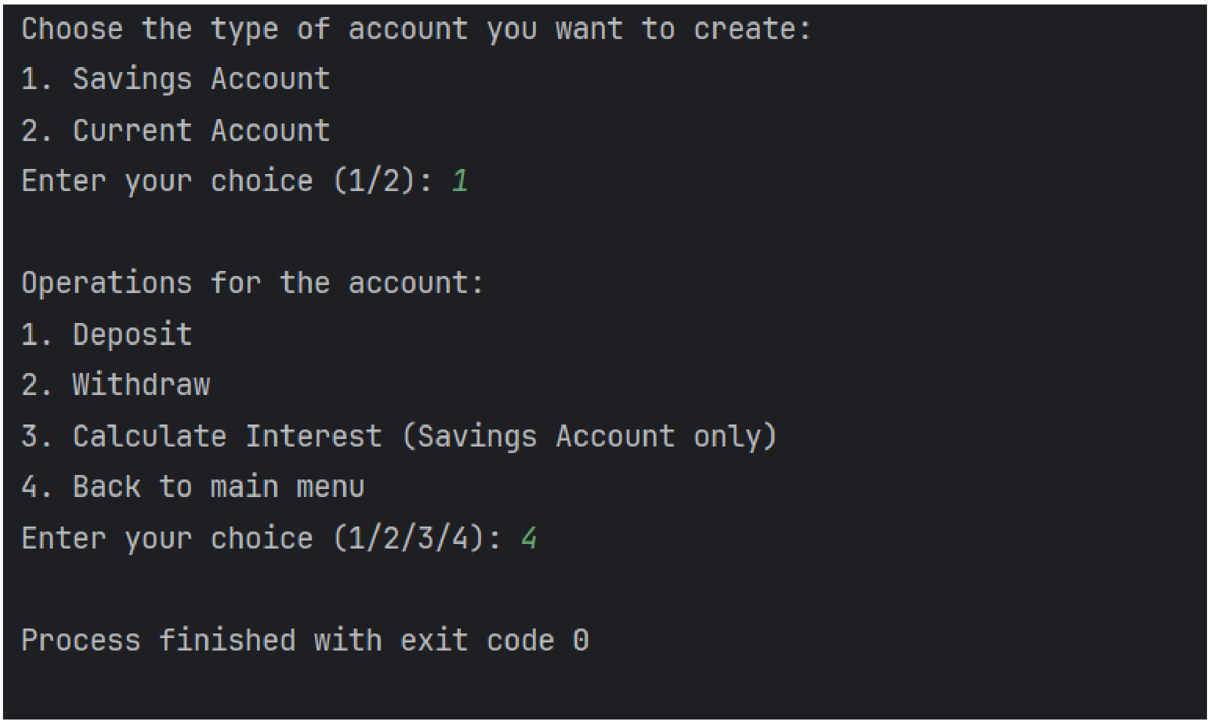
```
    def calculate_interest(self):
        interest_amount = self.balance * (7.8 / 100)
        print("Interest amount:", interest_amount)

class Bank:
    def create_account(self):
        print("Choose account type:")
        print("1. SavingsAccount")
        print("2. CurrentAccount")
        choice = input("Enter choice (1/2): ")
        if choice == "1":
            acc_type = "SavingsAccount"
        elif choice == "2":
            acc_type = "CurrentAccount"
        else:
            print("Invalid choice.")
            return None

        balance = float(input("Enter initial balance: "))
        return Account(acc_type, balance)

    def main(self):
        account = self.create_account()
        if account:
            while True:
                print("\nMenu:")
                print("1. Deposit")
                print("2. Withdraw")
                print("3. Calculate Interest (for
SavingsAccount)")
                print("4. Exit")
                choice = input("Enter choice (1/2/3/4): ")

                if choice == "1":
                    amount = float(input("Enter deposit amount:
"))
                    account.deposit(amount)

                elif choice == "2":
                    amount = float(input("Enter withdrawal
amount: "))
                    account.withdraw(amount)
                elif choice == "3" and account.acc_type ==
"SavingsAccount":
                    account.calculate_interest()
                elif choice == "4":
                    print("Thank you.")
                    break
                else:
                    print("Invalid choice.")

#bank1 = Bank()
#bank1.main()
```



```python
print(" Program Starts")

from abc import ABC,abstractmethod
class Bankaccount(ABC):
    def __init__(self, Account_number, Customer_name, Balance):
        self.Account_number = Account_number
        self.Customer_name = Customer_name
        self.Balance = Balance

    @property
    def account_number(self):
        return self.Account_number

    @account_number.setter
    def account_number(self, value):
        self.Account_number = value

    @property
    def customer_name(self):
        return self.Customer_name

    @customer_name.setter
    def customer_name(self, value):
        self.Customer_name = value

    @property
    def balance(self):
        return self.Balance

    @balance.setter
    def balance(self,value):
        self.Balance = value

    @abstractmethod
    def Deposit(self, amount):
        pass

    @abstractmethod
    def Withdrawl(self, amount):
        pass

    @abstractmethod
    def interest(self):
        pass

    def display(self):
        print("Account Number:", self.Account_number)
        print("Account Type:", self.Customer_name)
        print("Account Balance:", self.Balance)


class Account(Bankaccount):
```



```
    def __init__(self, Account_Number, Customer_name, Balance):
        super().__init__(Account_Number, Customer_name, Balance)

    def Deposit(self, amount):
        self.Balance += amount
        print("Deposit of ", amount, "completed.")
        print(" Availble balance: ",self.Balance)


    def Withdrawl(self, amount):
        if amount <= self.Balance:
            self.Balance -= amount
            print("Withdrawal of ", amount, "completed.")
            print(" Availble balance: ", self.Balance)
        else:
            print("Insufficient balance. Withdrawal cannot be
processed.")


    def interest(self):
        interest_rate = 4.5 / 100
        interest_amount = self.Balance * interest_rate
        print("Interest amount:", interest_amount)


Account1 = Account(12, "Logesh", 50000)
Account1.Deposit(1000)
Account1.Withdrawl(11000)
```



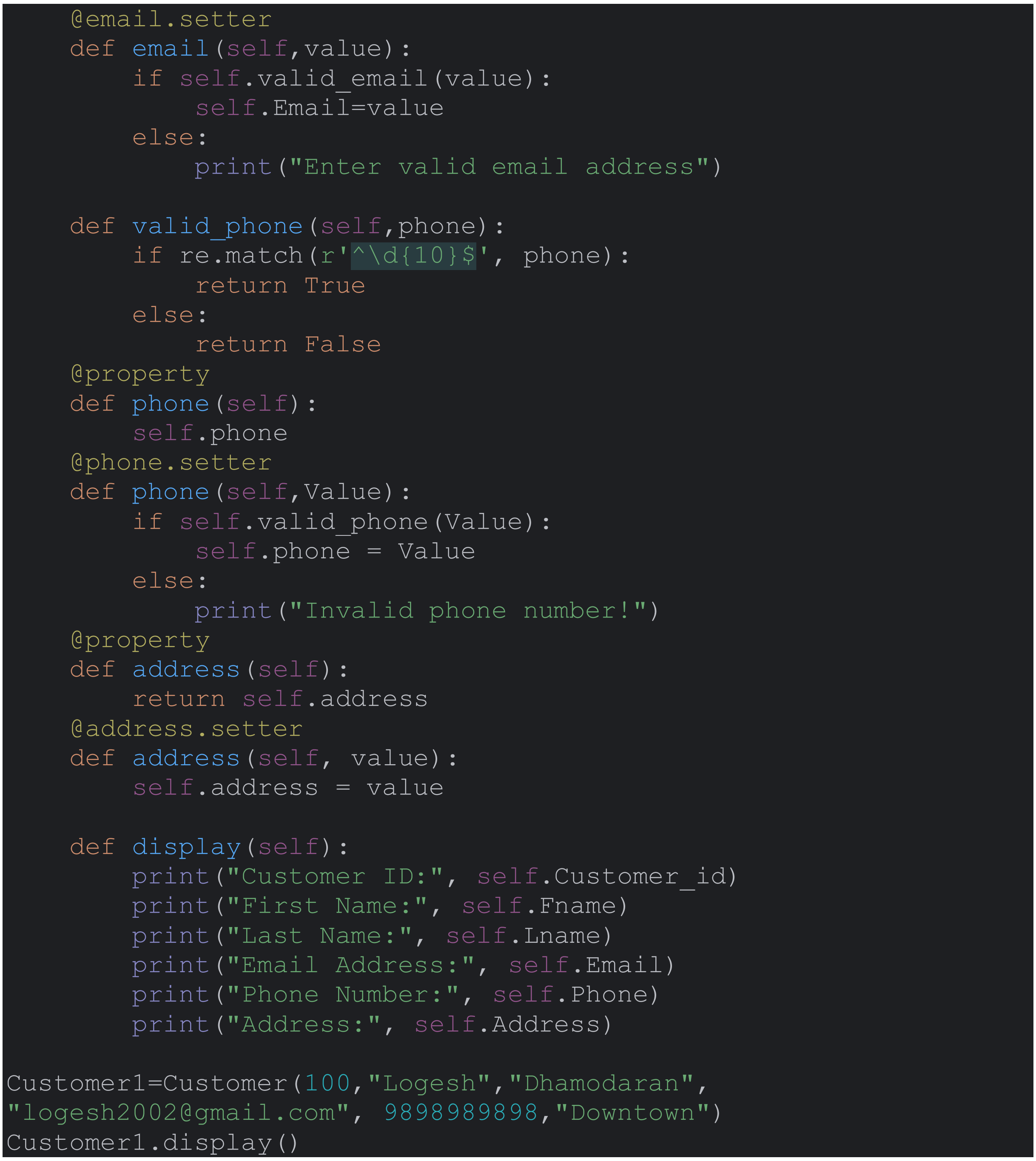
```
Choose the type of account you want to create:
1. Savings Account
2. Current Account
Enter your choice (1/2): 1

Operations for the account:
1. Deposit
2. Withdraw
3. Calculate Interest (Savings Account only)
4. Back to main menu
Enter your choice (1/2/3/4): 4


Process finished with exit code 0
```

Task 10: Has A Relation / Association

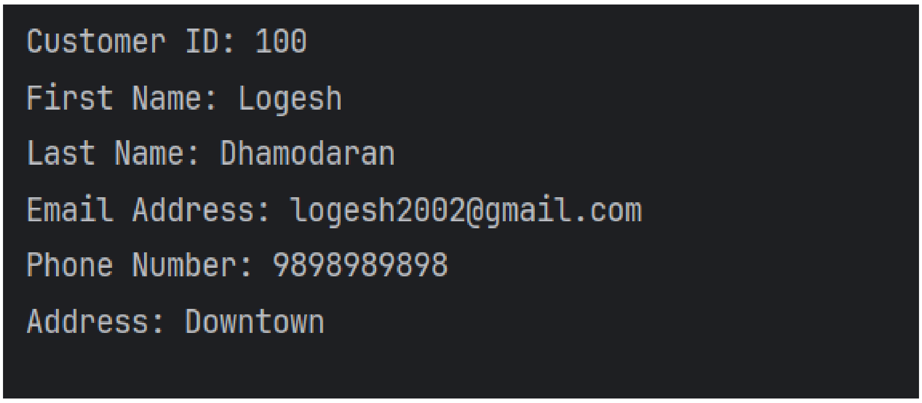
1. Create a `Customer` class with the following attributes:

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

```
import re
class Customer:
    def
__init__(self,Customer_id,Fname,Lname,Email,Phone,Address):
        self.Customer_id = Customer_id
        self.Fname=Fname
        self.Lname = Lname
        self.Email = Email
        self.Phone = Phone
        self.Address=Address

    @property
    def cid(self):
        return self.Customer_id
    @cid.setter
    def cid(self,value):
        self.Customer_id=value

    @property
    def fname(self):
        return self.Fname
    @fname.setter
    def fname(self, value):
        self.Fname = value
    @property
    def lname(self):
        return self.Lname

    @lname.setter
    def lname(self, value):
        self.Lname = value
    def valid_email(email):
        if re.match(r'^[\w\.-]+@[\w\.-]+$', email):
            return True
        else:
            return False
    @property
    def email(self):
        return self.Email
```



```python
    @email.setter
    def email(self,value):
        if self.valid_email(value):
            self.Email=value
        else:
            print("Enter valid email address")

    def valid_phone(self,phone):
        if re.match(r'^\d{10}$', phone):
            return True
        else:
            return False
    @property
    def phone(self):
        self.phone
    @phone.setter
    def phone(self,Value):
        if self.valid_phone(Value):
            self.phone = Value
        else:
            print("Invalid phone number!")
    @property
    def address(self):
        return self.address
    @address.setter
    def address(self, value):
        self.address = value

    def display(self):
        print("Customer ID:", self.Customer_id)
        print("First Name:", self.Fname)
        print("Last Name:", self.Lname)
        print("Email Address:", self.Email)
        print("Phone Number:", self.Phone)
        print("Address:", self.Address)

Customer1=Customer(100,"Logesh","Dhamodaran",
"logesh2002@gmail.com", 9898989898,"Downtown")
Customer1.display()
```



```
Customer ID: 100
First Name: Logesh
Last Name: Dhamodaran
Email Address: logesh2002@gmail.com
Phone Number: 9898989898
Address: Downtown
```

2.

Create an `Account` class with the following attributes: Account Number (a unique identifier).

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

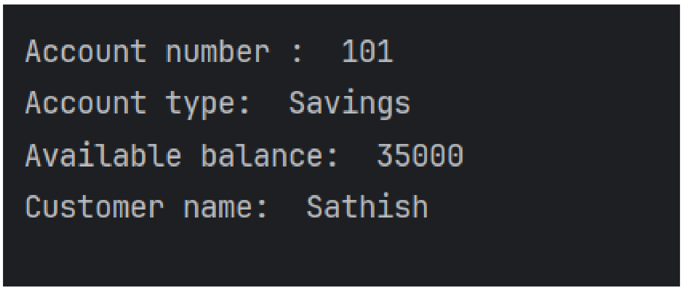
Account Balance

Customer (the customer who owns the account)

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.

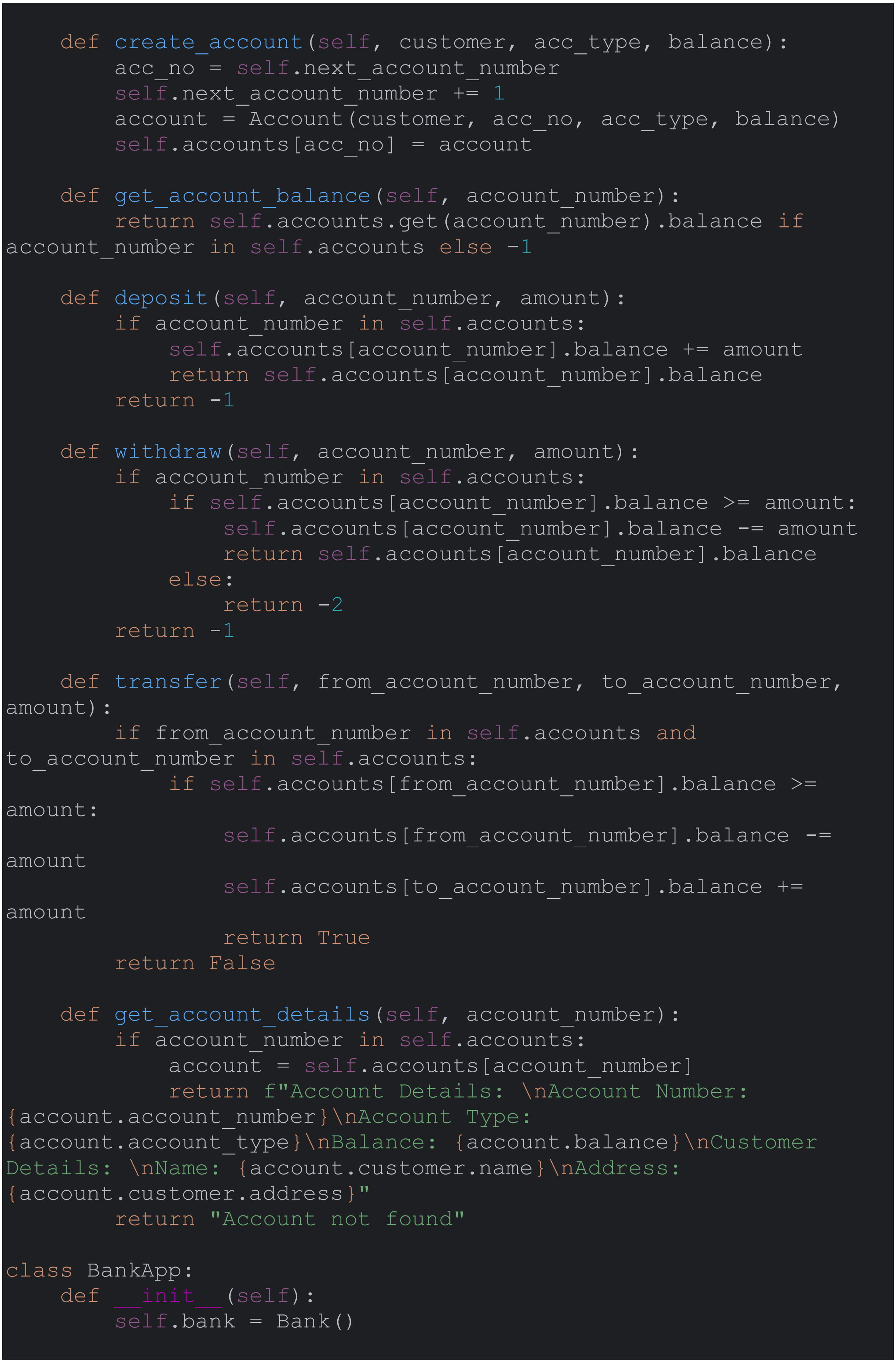
```
class Account:
    def __init__(self, account_number, account_type, balance, 
customer):
        self.account_number = account_number
        self.account_type = account_type
        self.balance = balance
        self.customer = customer
    @property
    def acc_number(self):
        return self.account_number

    @acc_number.setter
    def acc_number(self, account_number):
        self.account_number = account_number

    @property
    def acc_type(self):
        return self.account_type
    @acc_type.setter
    def acc_type(self, account_type):
        self.account_type = account_type
    @property
    def set_bal(self):
        return self.balance
    @set_bal.setter
    def set_bal(self, balance):
        self.balance = balance
    @property
    def cust(self):
        return self.customer
    @cust.setter
    def cust(self, customer):
        self.customer = customer
    def display(self):
        print("Account number : ",self.account_number)
        print("Account type: ",self.account_type)
        print("Available balance: ",self.balance)
        print("Customer name: ",self.customer)
Account1 = Account(101,"Savings", 35000,"Sathish")
Account1.display()
```



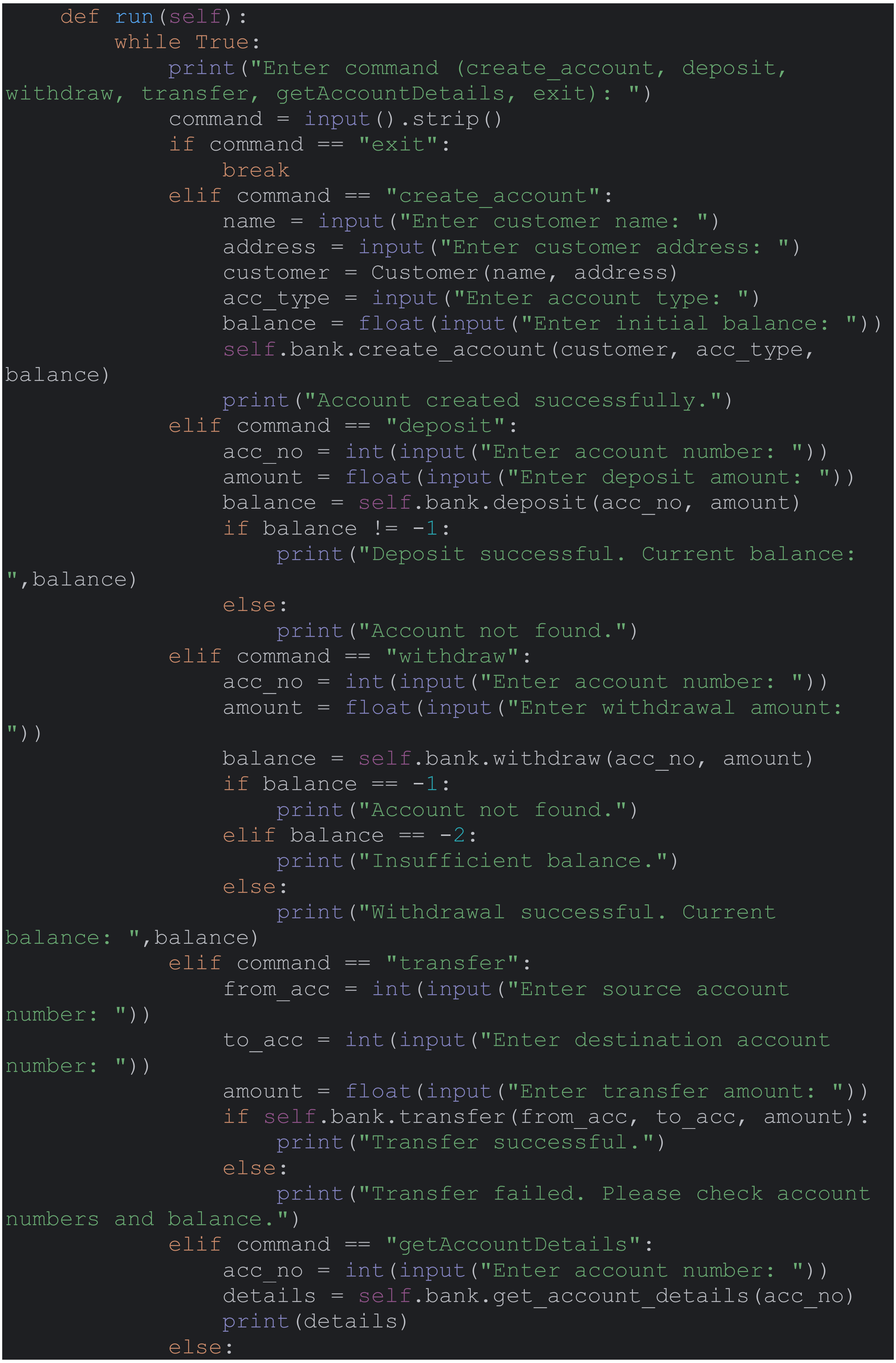
```
Account number :  101
Account type:  Savings
Available balance:  35000
Customer name:  Sathish
```

3. 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.

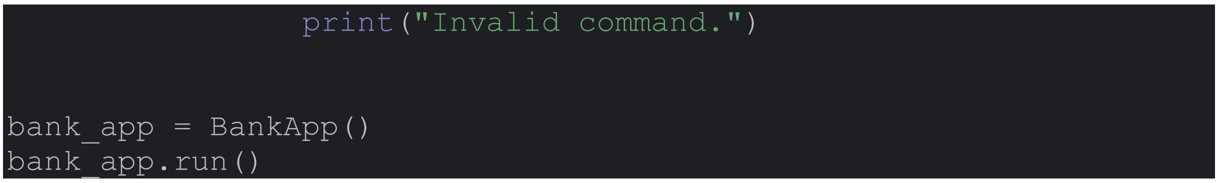
```
class Customers:
    def __init__(self, customer_id, first_name, last_name, email,
phone, address):
        self.customer_id = customer_id
        self.first_name = first_name
        self.last_name = last_name
        self.email = email
        self.phone = phone
        self.address = address
class Accounts:
    account_counter = 1000  # Starting account number

    def __init__(self, customer, account_type, balance):
        Account.account_counter += 1
        self.account_number = Account.account_counter
        self.customer = customer
        self.account_type = account_type
        self.balance = balance
class Customer:
    def __init__(self, name, address):
        self.name = name
        self.address = address


class Account:
    def __init__(self, customer, account_number, account_type,
balance):
        self.customer = customer
        self.account_number = account_number
        self.account_type = account_type
        self.balance = balance

class Bank:
    def __init__(self):
        self.next_account_number = 1001
        self.accounts = {}
```



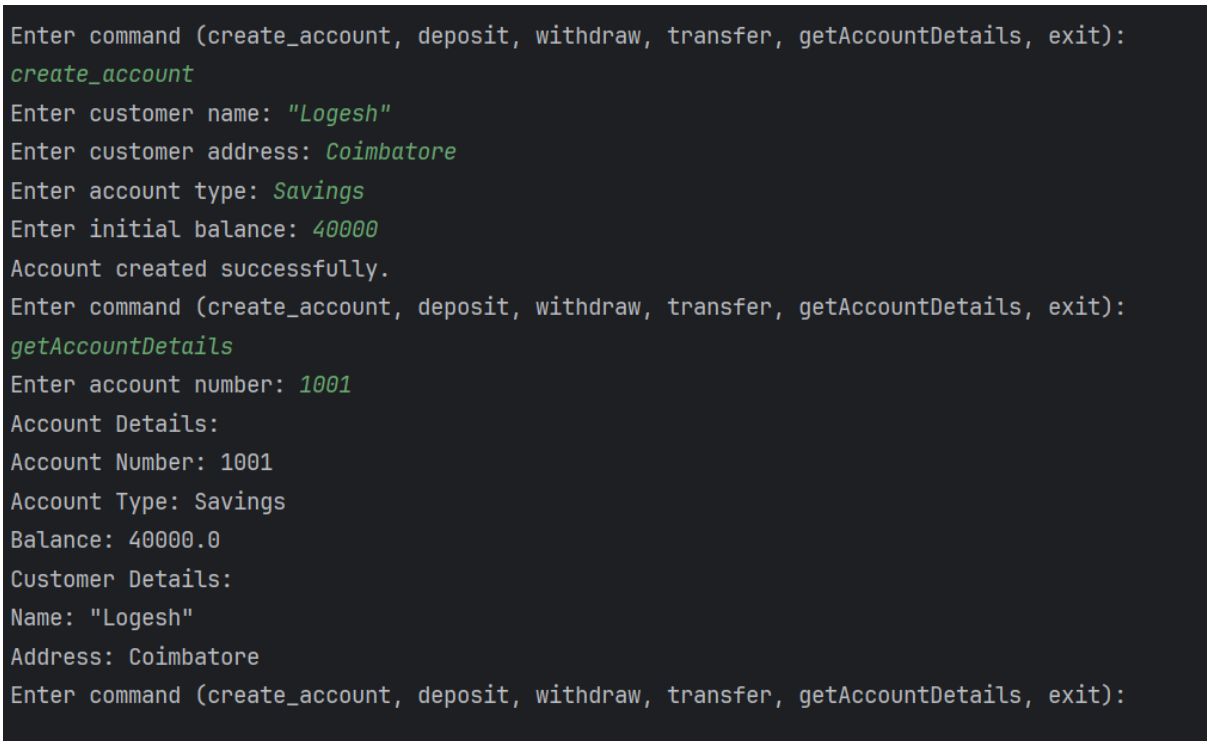
```
    def create_account(self, customer, acc_type, balance):
        acc_no = self.next_account_number
        self.next_account_number += 1
        account = Account(customer, acc_no, acc_type, balance)
        self.accounts[acc_no] = account

    def get_account_balance(self, account_number):
        return self.accounts.get(account_number).balance if
account_number in self.accounts else -1

    def deposit(self, account_number, amount):
        if account_number in self.accounts:
            self.accounts[account_number].balance += amount
            return self.accounts[account_number].balance
        return -1

    def withdraw(self, account_number, amount):
        if account_number in self.accounts:
            if self.accounts[account_number].balance >= amount:
                self.accounts[account_number].balance -= amount
                return self.accounts[account_number].balance
            else:
                return -2
        return -1

    def transfer(self, from_account_number, to_account_number,
amount):
        if from_account_number in self.accounts and
to_account_number in self.accounts:
            if self.accounts[from_account_number].balance >=
amount:
                self.accounts[from_account_number].balance -=
amount
                self.accounts[to_account_number].balance +=
amount
                return True
        return False

    def get_account_details(self, account_number):
        if account_number in self.accounts:
            account = self.accounts[account_number]
            return f"Account Details: \nAccount Number:
{account.account_number}\nAccount Type:
{account.account_type}\nBalance: {account.balance}\nCustomer
Details: \nName: {account.customer.name}\nAddress:
{account.customer.address}"
        return "Account not found"

class BankApp:
    def __init__(self):
        self.bank = Bank()
```



```
    def run(self):
        while True:
            print("Enter command (create_account, deposit,
withdraw, transfer, getAccountDetails, exit): ")
            command = input().strip()
            if command == "exit":
                break
            elif command == "create_account":
                name = input("Enter customer name: ")
                address = input("Enter customer address: ")
                customer = Customer(name, address)
                acc_type = input("Enter account type: ")
                balance = float(input("Enter initial balance: "))
                self.bank.create_account(customer, acc_type,
balance)
                print("Account created successfully.")
            elif command == "deposit":
                acc_no = int(input("Enter account number: "))
                amount = float(input("Enter deposit amount: "))
                balance = self.bank.deposit(acc_no, amount)
                if balance != -1:
                    print("Deposit successful. Current balance:
",balance)
                else:
                    print("Account not found.")
            elif command == "withdraw":
                acc_no = int(input("Enter account number: "))
                amount = float(input("Enter withdrawal amount:
"))
                balance = self.bank.withdraw(acc_no, amount)
                if balance == -1:
                    print("Account not found.")
                elif balance == -2:
                    print("Insufficient balance.")
                else:
                    print("Withdrawal successful. Current
balance: ",balance)
            elif command == "transfer":
                from_acc = int(input("Enter source account
number: "))
                to_acc = int(input("Enter destination account
number: "))
                amount = float(input("Enter transfer amount: "))
                if self.bank.transfer(from_acc, to_acc, amount):
                    print("Transfer successful.")
                else:
                    print("Transfer failed. Please check account
numbers and balance.")
            elif command == "getAccountDetails":
                acc_no = int(input("Enter account number: "))
                details = self.bank.get_account_details(acc_no)
                print(details)
            else:
```



```
                print("Invalid command.")


bank_app = BankApp()
bank_app.run()
```



```
Enter command (create_account, deposit, withdraw, transfer, getAccountDetails, exit): 
create_account
Enter customer name: "Logesh"
Enter customer address: Coimbatore
Enter account type: Savings
Enter initial balance: 40000
Account created successfully.
Enter command (create_account, deposit, withdraw, transfer, getAccountDetails, exit): 
getAccountDetails
Enter account number: 1001
Account Details:
Account Number: 1001
Account Type: Savings
Balance: 40000.0
Customer Details:
Name: "Logesh"
Address: Coimbatore
Enter command (create_account, deposit, withdraw, transfer, getAccountDetails, 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.

Account Number (a unique identifier).

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

Account Balance

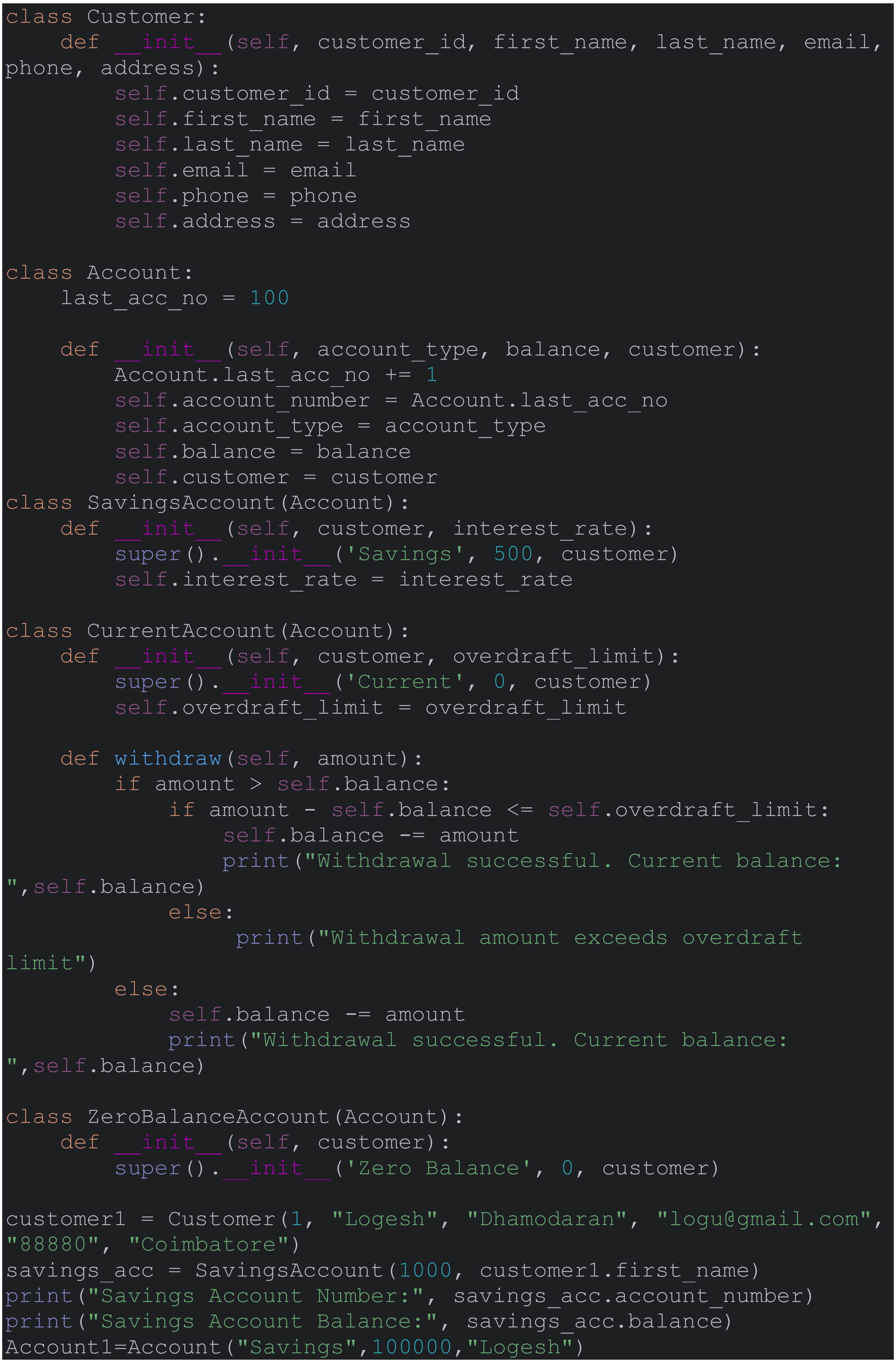
Customer (the customer who owns the account)

lastAccNo

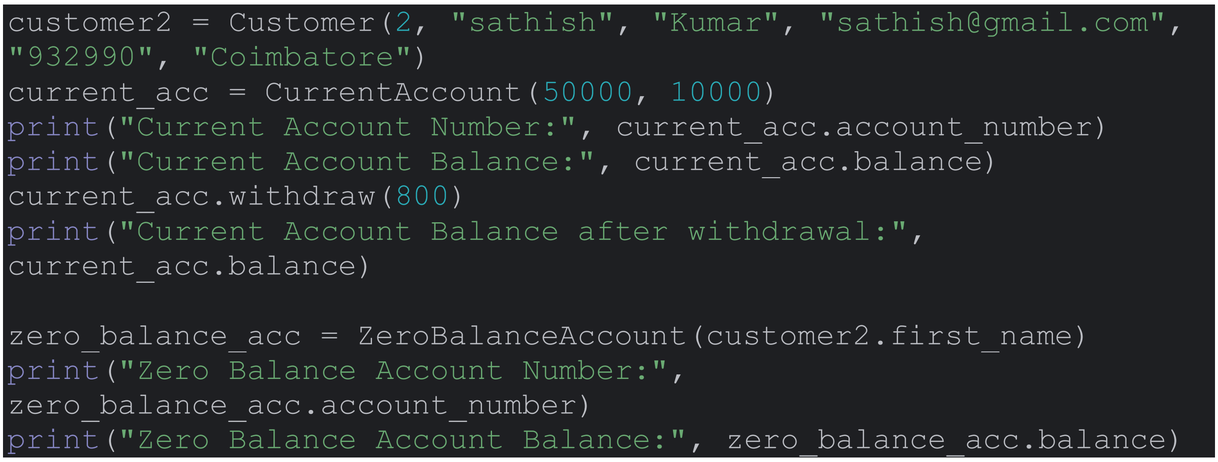
3. Create three child classes that inherit the Account class and each class must contain below mentioned attribute: **SavingsAccount:** A savings account that includes an additional attribute for interest rate. Saving account should be created with minimum balance 500.

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.

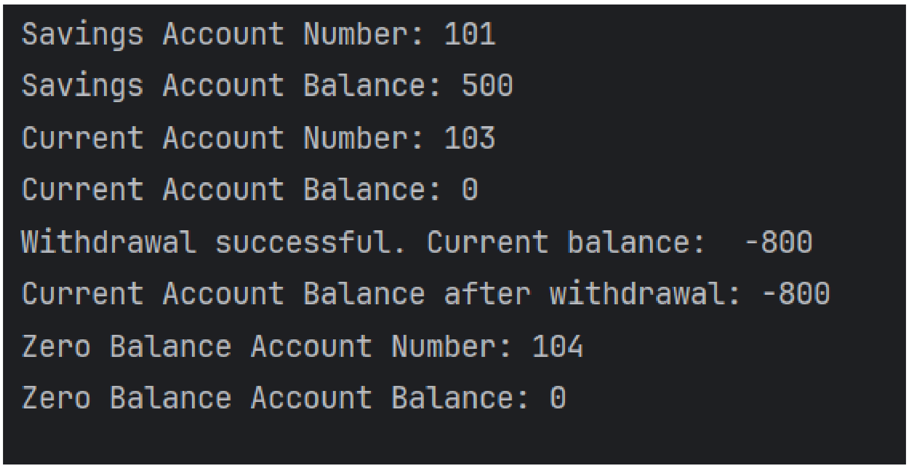
ZeroBalanceAccount: ZeroBalanceAccount can be created with Zero balance.



```
class Customer:
    def __init__(self, customer_id, first_name, last_name, email,
phone, address):
        self.customer_id = customer_id
        self.first_name = first_name
        self.last_name = last_name
        self.email = email
        self.phone = phone
        self.address = address

class Account:
    last_acc_no = 100

    def __init__(self, account_type, balance, customer):
        Account.last_acc_no += 1
        self.account_number = Account.last_acc_no
        self.account_type = account_type
        self.balance = balance
        self.customer = customer
class SavingsAccount(Account):
    def __init__(self, customer, interest_rate):
        super().__init__('Savings', 500, customer)
        self.interest_rate = interest_rate

class CurrentAccount(Account):
    def __init__(self, customer, overdraft_limit):
        super().__init__('Current', 0, customer)
        self.overdraft_limit = overdraft_limit

    def withdraw(self, amount):
        if amount > self.balance:
            if amount - self.balance <= self.overdraft_limit:
                self.balance -= amount
                print("Withdrawal successful. Current balance:
",self.balance)
            else:
                 print("Withdrawal amount exceeds overdraft
limit")
        else:
            self.balance -= amount
            print("Withdrawal successful. Current balance:
",self.balance)

class ZeroBalanceAccount(Account):
    def __init__(self, customer):
        super().__init__('Zero Balance', 0, customer)

customer1 = Customer(1, "Logesh", "Dhamodaran", "logu@gmail.com",
"88880", "Coimbatore")
savings_acc = SavingsAccount(1000, customer1.first_name)
print("Savings Account Number:", savings_acc.account_number)
print("Savings Account Balance:", savings_acc.balance)
Account1=Account("Savings",100000,"Logesh")
```



```
customer2 = Customer(2, "sathish", "Kumar", "sathish@gmail.com",
"932990", "Coimbatore")
current_acc = CurrentAccount(50000, 10000)
print("Current Account Number:", current_acc.account_number)
print("Current Account Balance:", current_acc.balance)
current_acc.withdraw(800)
print("Current Account Balance after withdrawal:",
current_acc.balance)

zero_balance_acc = ZeroBalanceAccount(customer2.first_name)
print("Zero Balance Account Number:",
zero_balance_acc.account_number)
print("Zero Balance Account Balance:", zero_balance_acc.balance)
```



```
Savings Account Number: 101
Savings Account Balance: 500
Current Account Number: 103
Current Account Balance: 0
Withdrawal successful. Current balance:  -800
Current Account Balance after withdrawal: -800
Zero Balance Account Number: 104
Zero Balance Account Balance: 0
```

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

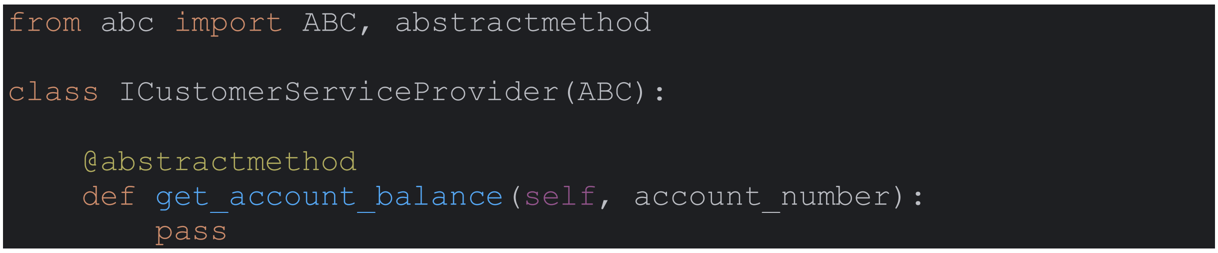
get_account_balance(account_number: long): Retrieve the balance of an account given its account number. should return the current balance of account.

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.



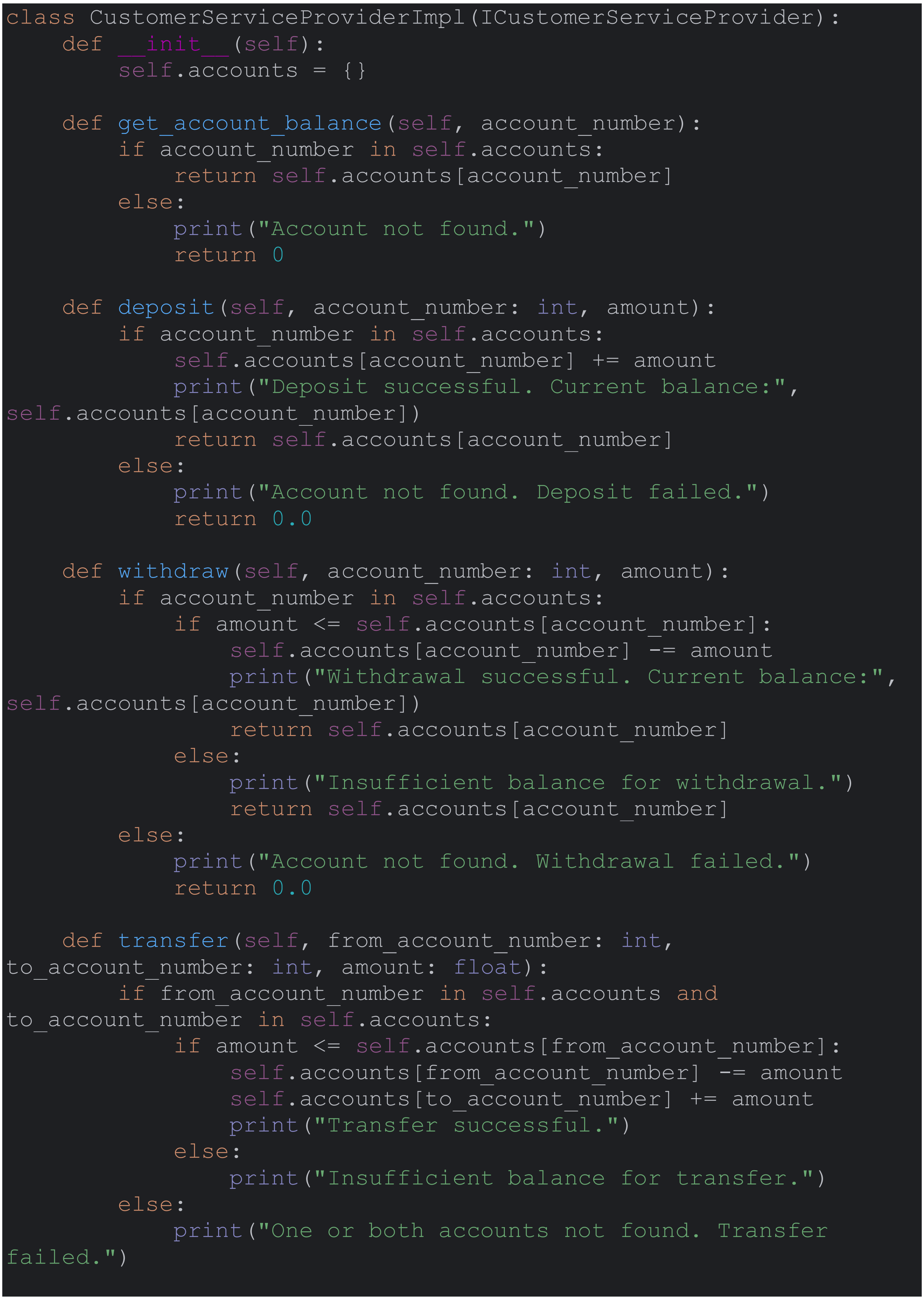
```
from abc import ABC, abstractmethod

class ICustomerServiceProvider(ABC):

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

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

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

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

5. 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():Account[] accounts: List all accounts in the bank. Limited. All rights www.hexaware.com • • calculateInterest(): the calculate_interest() method to calculate interest based on the balance and interest rate

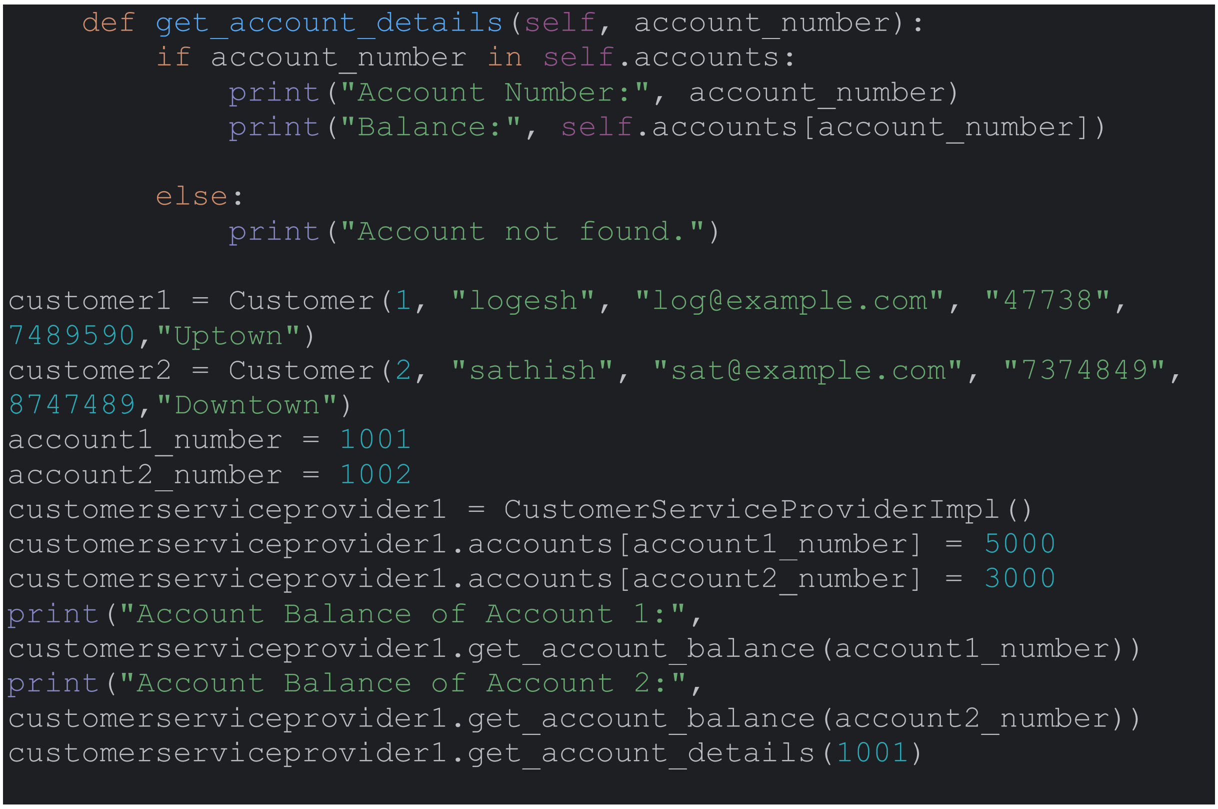
```
from abc import ABC, abstractmethod

class IBankServiceProvider(ABC):

    @abstractmethod
    def create_account(self, customer, accNo: int, accType: str, 
balance: float):
        pass

    @abstractmethod
    def list_accounts(self) -> list:
        pass

    @abstractmethod
    def calculate_interest(self) -> float:
        pass
```

6.

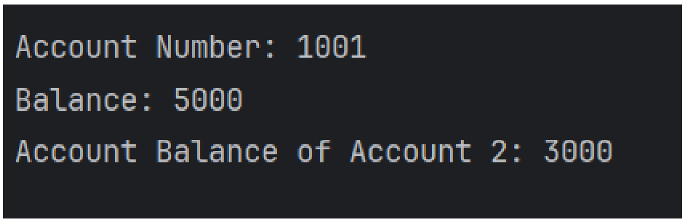
Create **CustomerServiceProviderImpl** class which implements I**CustomerServiceProvider** provide all implementation methods.



```
class CustomerServiceProviderImpl(ICustomerServiceProvider):
    def __init__(self):
        self.accounts = {}

    def get_account_balance(self, account_number):
        if account_number in self.accounts:
            return self.accounts[account_number]
        else:
            print("Account not found.")
            return 0

    def deposit(self, account_number: int, amount):
        if account_number in self.accounts:
            self.accounts[account_number] += amount
            print("Deposit successful. Current balance:",
self.accounts[account_number])
            return self.accounts[account_number]
        else:
            print("Account not found. Deposit failed.")
            return 0.0

    def withdraw(self, account_number: int, amount):
        if account_number in self.accounts:
            if amount <= self.accounts[account_number]:
                self.accounts[account_number] -= amount
                print("Withdrawal successful. Current balance:",
self.accounts[account_number])
                return self.accounts[account_number]
            else:
                print("Insufficient balance for withdrawal.")
                return self.accounts[account_number]
        else:
            print("Account not found. Withdrawal failed.")
            return 0.0

    def transfer(self, from_account_number: int,
to_account_number: int, amount: float):
        if from_account_number in self.accounts and
to_account_number in self.accounts:
            if amount <= self.accounts[from_account_number]:
                self.accounts[from_account_number] -= amount
                self.accounts[to_account_number] += amount
                print("Transfer successful.")
            else:
                print("Insufficient balance for transfer.")
        else:
            print("One or both accounts not found. Transfer
failed.")
```



```
    def get_account_details(self, account_number):
        if account_number in self.accounts:
            print("Account Number:", account_number)
            print("Balance:", self.accounts[account_number])

        else:
            print("Account not found.")

customer1 = Customer(1, "logesh", "log@example.com", "47738",
7489590,"Uptown")
customer2 = Customer(2, "sathish", "sat@example.com", "7374849",
8747489,"Downtown")
account1_number = 1001
account2_number = 1002
customerserviceprovider1 = CustomerServiceProviderImpl()
customerserviceprovider1.accounts[account1_number] = 5000
customerserviceprovider1.accounts[account2_number] = 3000
print("Account Balance of Account 1:",
customerserviceprovider1.get_account_balance(account1_number))
print("Account Balance of Account 2:",
customerserviceprovider1.get_account_balance(account2_number))
customerserviceprovider1.get_account_details(1001)
```



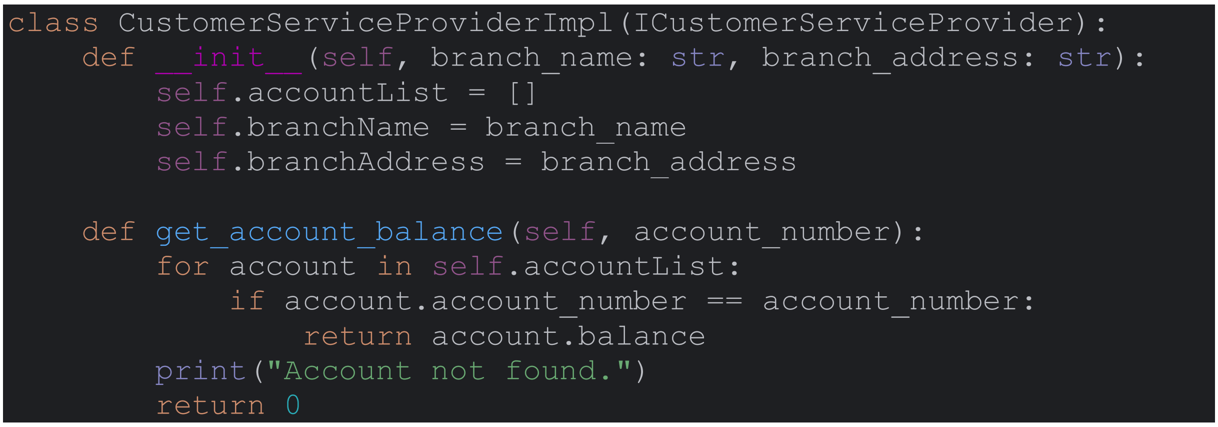
```
Account Number: 1001
Balance: 5000
Account Balance of Account 2: 3000
```

7.

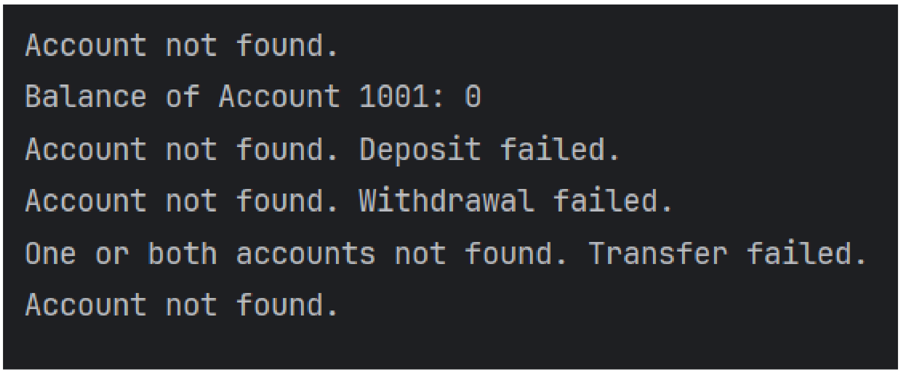
Create **BankServiceProviderImpl** class which inherits from **CustomerServiceProviderImpl and implements IBankServiceProvider**

Attributes ○ accountList: Array of **Accounts** to store any account objects.

○ branchName and branchAddress as String objects.



```
class CustomerServiceProviderImpl(ICustomerServiceProvider):
    def __init__(self, branch_name: str, branch_address: str):
        self.accountList = []
        self.branchName = branch_name
        self.branchAddress = branch_address

    def get_account_balance(self, account_number):
        for account in self.accountList:
            if account.account_number == account_number:
                return account.balance
        print("Account not found.")
        return 0
    def deposit(self, account_number: int, amount: float):
        for account in self.accountList:
            if account.account_number == account_number:
                account.balance += amount
                print("Deposit successful. Current balance:", account.balance)
                return account.balance
        print("Account not found. Deposit failed.")
        return 0

    def withdraw(self, account_number: int, amount):
        for account in self.accountList:
            if account.account_number == account_number:
                if amount <= account.balance:
                    account.balance -= amount
                    print("Withdrawal successful. Current balance:", account.balance)
                    return account.balance
                else:
                    print("Insufficient balance for withdrawal.")
                    return account.balance
        print("Account not found. Withdrawal failed.")
        return 0

    def transfer(self, from_account_number: int, to_account_number: int, amount: float):
        from_account = None
        to_account = None
        for account in self.accountList:
            if account.account_number == from_account_number:
                from_account = account
            elif account.account_number == to_account_number:
                to_account = account

        if from_account and to_account:
            if amount <= from_account.balance:
                from_account.balance -= amount
                to_account.balance += amount
                print("Transfer successful.")
            else:
                print("Insufficient balance for transfer.")
        else:
            print("One or both accounts not found. Transfer failed.")

    def get_account_details(self, account_number):
        for account in self.accountList:
            if account.account_number == account_number:
                print("Account Number:", account.account_number)
                print("Balance:", account.balance)

                return
        print("Account not found.")


customer_service_provider = CustomerServiceProviderImpl("Main
Branch", "123 Main St")


balance =
customer_service_provider.get_account_balance(account_number=1001
)
print("Balance of Account 1001:", balance)
new_balance = customer_service_provider.deposit(1002, 500)
new_balance = customer_service_provider.withdraw(1003, 200)
customer_service_provider.transfer(1001, 1001, 100)
customer_service_provider.get_account_details(account_number=1001
)
```



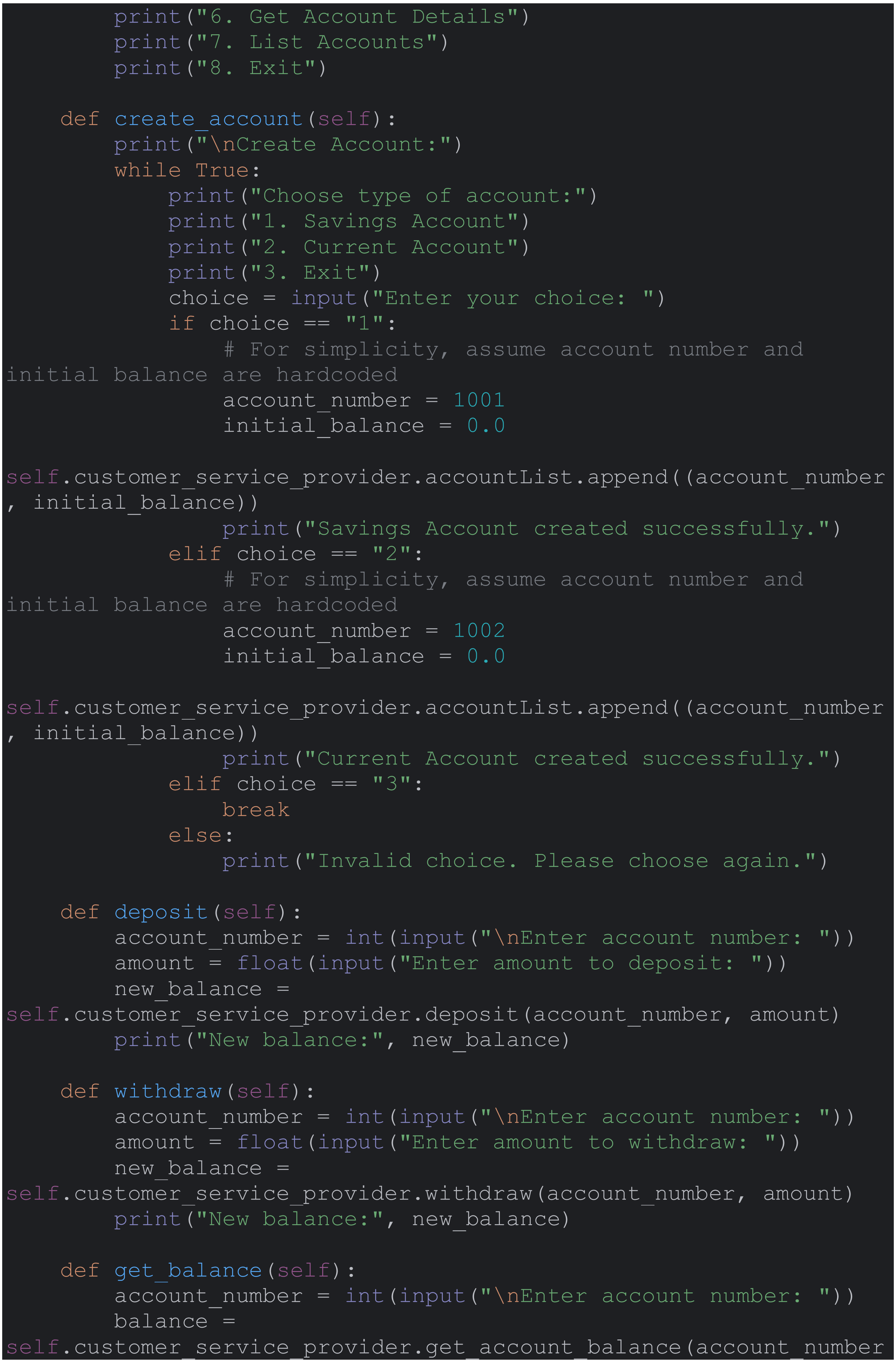
```
Account not found.
Balance of Account 1001: 0
Account not found. Deposit failed.
Account not found. Withdrawal failed.
One or both accounts not found. Transfer failed.
Account not found.
```

8 Create **BankApp** class and perform following operation:

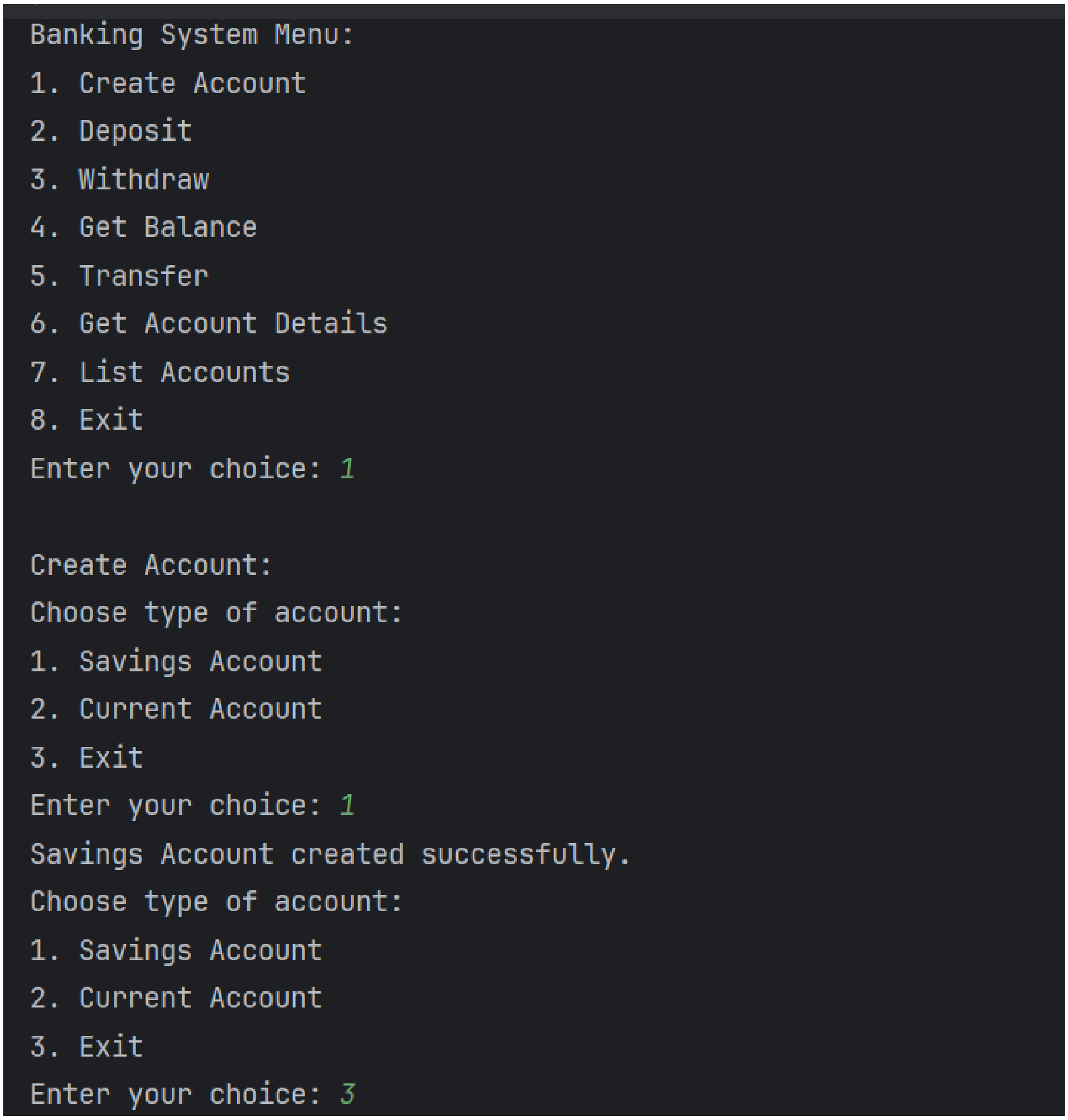
•main method to simulate the banking system. Allow the user to interact with the system by entering choice from menu such as "create_account", "deposit", "withdraw", "get_balance", "transfer", "getAccountDetails", "ListAccounts" and "exit."

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

```
class BankApp:
    def __init__(self):
        self.customer_service_provider =
CustomerServiceProviderImpl(branch_name="Main Branch",
branch_address="123 Main St")

    def display_menu(self):
        print("\nBanking System Menu:")
        print("1. Create Account")
        print("2. Deposit")
        print("3. Withdraw")
        print("4. Get Balance")
        print("5. Transfer")
```



```
        print("6. Get Account Details")
        print("7. List Accounts")
        print("8. Exit")

    def create_account(self):
        print("\nCreate Account:")
        while True:
            print("Choose type of account:")
            print("1. Savings Account")
            print("2. Current Account")
            print("3. Exit")
            choice = input("Enter your choice: ")
            if choice == "1":
                # For simplicity, assume account number and
initial balance are hardcoded
                account_number = 1001
                initial_balance = 0.0

self.customer_service_provider.accountList.append((account_number
, initial_balance))
                print("Savings Account created successfully.")
            elif choice == "2":
                # For simplicity, assume account number and
initial balance are hardcoded
                account_number = 1002
                initial_balance = 0.0

self.customer_service_provider.accountList.append((account_number
, initial_balance))
                print("Current Account created successfully.")
            elif choice == "3":
                break
            else:
                print("Invalid choice. Please choose again.")

    def deposit(self):
        account_number = int(input("\nEnter account number: "))
        amount = float(input("Enter amount to deposit: "))
        new_balance =
self.customer_service_provider.deposit(account_number, amount)
        print("New balance:", new_balance)

    def withdraw(self):
        account_number = int(input("\nEnter account number: "))
        amount = float(input("Enter amount to withdraw: "))
        new_balance =
self.customer_service_provider.withdraw(account_number, amount)
        print("New balance:", new_balance)

    def get_balance(self):
        account_number = int(input("\nEnter account number: "))
        balance =
self.customer_service_provider.get_account_balance(account_number
)
        print("Current balance:", balance)

    def transfer(self):
        from_account_number = int(input("\nEnter sender's account 
number: "))
        to_account_number = int(input("Enter receiver's account 
number: "))
        amount = float(input("Enter amount to transfer: "))

self.customer_service_provider.transfer(from_account_number, 
to_account_number, amount)

    def get_account_details(self):
        account_number = int(input("\nEnter account number: "))

self.customer_service_provider.get_account_details(account_number
)

    def list_accounts(self):
        print("\nList of Accounts:")
        for account_number, balance in 
self.customer_service_provider.accounts.items():
            print("Account Number:", account_number, "- 
Balance:", balance)

    def run(self):
        while True:
            self.display_menu()
            choice = input("Enter your choice: ")
            if choice == "1":
                self.create_account()
            elif choice == "2":
                self.deposit()
            elif choice == "3":
                self.withdraw()
            elif choice == "4":
                self.get_balance()
            elif choice == "5":
                self.transfer()
            elif choice == "6":
                self.get_account_details()
            elif choice == "7":
                self.list_accounts()
            elif choice == "8":
                print("Exiting...")
                break
            else:
                print("Invalid choice. Please choose again.")


bank_app = BankApp()
bank_app.run()
```



```
Banking System Menu:
1. Create Account
2. Deposit
3. Withdraw
4. Get Balance
5. Transfer
6. Get Account Details
7. List Accounts
8. Exit
Enter your choice: 1

Create Account:
Choose type of account:
1. Savings Account
2. Current Account
3. Exit
Enter your choice: 1
Savings Account created successfully.
Choose type of account:
1. Savings Account
2. Current Account
3. Exit
Enter your choice: 3
```

9. Create I**BankRepository** interface/abstract class which include following methods to interact with database.

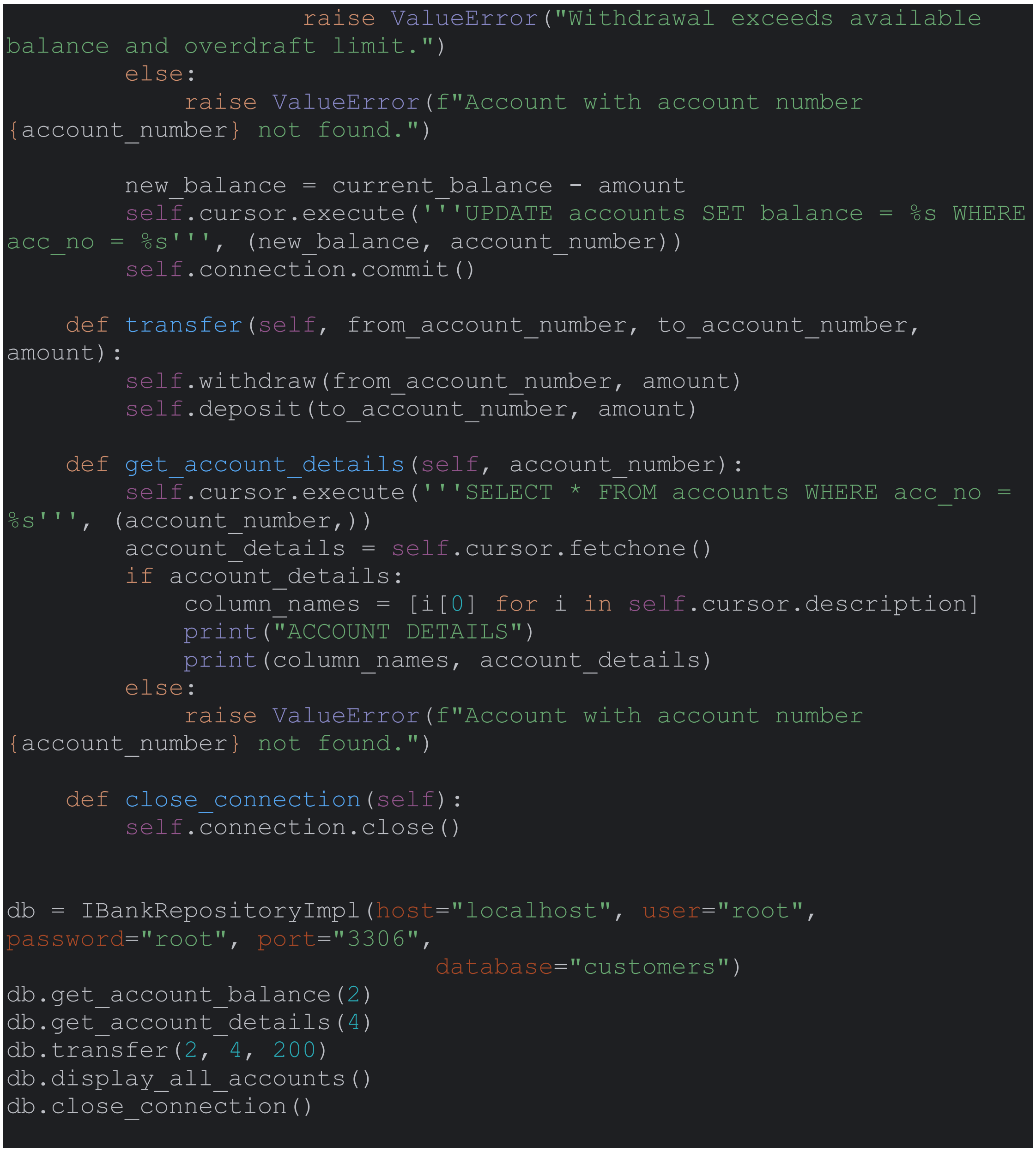
- **createAccount(customer: Customer, accNo: long, accType: String, balance: float)**: Create a new bank account for the given customer with the initial balance and store in database.
- **listAccounts()**: List<Account> accountsList: List all accounts in the bank from database.
- **calculateInterest():** 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.
    - 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. check the balance from account in database and new balance should updated in Database.
- **getAccountDetails(account_number: long):** Should return the account and customer details from databse.
- **getTransations(account_number: long, FromDate:Date, ToDate: Date):** Should return the list of transaction between two dates from database.

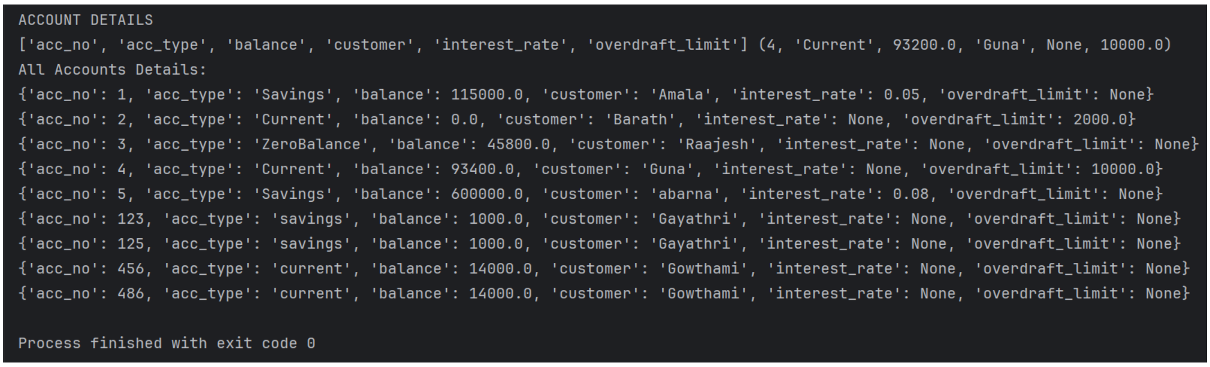
2.
```
import mysql.connector
from abc import ABC, abstractmethod


class IBankRepository(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

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

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

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

```
class IBankRepositoryImpl(IBankRepository):
    def __init__(self, host, user, password, port, database):
        self.connection = mysql.connector.connect(
            host=host,
            user=user,
            password=password,
            port=port,
            database=database
        )
        self.cursor = self.connection.cursor()

    def display_all_accounts(self):
        self.cursor.execute('''SELECT * FROM accounts''')
        all_accounts = self.cursor.fetchall()
        if all_accounts:
            print("All Accounts Details:")
            for account in all_accounts:
                column_names = [i[0] for i in
self.cursor.description]
                account_details = dict(zip(column_names, account))
                print(account_details)
        else:
            print("No accounts found in the database.")

    def get_account_balance(self, account_number):
        self.cursor.execute("SELECT balance FROM accounts WHERE
acc_no = %s", (account_number,))
        balance = self.cursor.fetchone()
        if balance:
            return balance[0]
            print(f"The balance of {account_number} is {balance}")
        else:
            raise ValueError(f"Account with account number
{account_number} not found.")

    def deposit(self, account_number, amount):
        current_balance = self.get_account_balance(account_number)
        new_balance = current_balance + amount
        self.cursor.execute('''UPDATE accounts SET balance = %s WHERE
acc_no = %s''', (new_balance, account_number))
        self.connection.commit()

    def withdraw(self, account_number, amount):
        current_balance = self.get_account_balance(account_number)
        self.cursor.execute('''SELECT acc_type, overdraft_limit FROM
accounts WHERE acc_no = %s''', (account_number,))
        account_info = self.cursor.fetchone()
        if account_info:
            acc_type, overdraft_limit = account_info
            if acc_type == 'Savings':
                if current_balance - amount < 500:
                    raise ValueError("Withdrawal violates minimum
balance rule.")
            elif acc_type == 'Current':
                available_balance = current_balance + overdraft_limit
                if amount > available_balance:
```



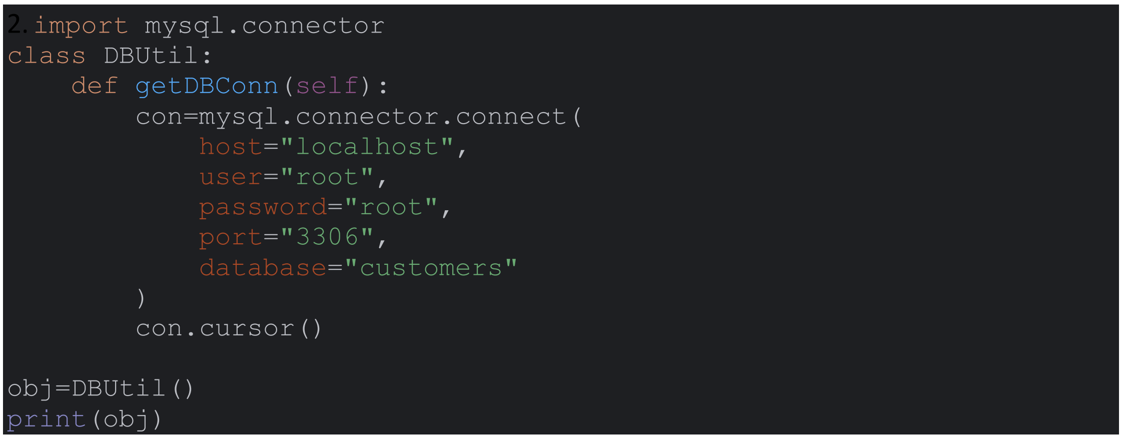
```
                    raise ValueError("Withdrawal exceeds available
balance and overdraft limit.")
        else:
            raise ValueError(f"Account with account number
{account_number} not found.")

        new_balance = current_balance - amount
        self.cursor.execute('''UPDATE accounts SET balance = %s WHERE
acc_no = %s''', (new_balance, account_number))
        self.connection.commit()

    def transfer(self, from_account_number, to_account_number,
amount):
        self.withdraw(from_account_number, amount)
        self.deposit(to_account_number, amount)

    def get_account_details(self, account_number):
        self.cursor.execute('''SELECT * FROM accounts WHERE acc_no =
%s''', (account_number,))
        account_details = self.cursor.fetchone()
        if account_details:
            column_names = [i[0] for i in self.cursor.description]
            print("ACCOUNT DETAILS")
            print(column_names, account_details)
        else:
            raise ValueError(f"Account with account number
{account_number} not found.")

    def close_connection(self):
        self.connection.close()


db = IBankRepositoryImpl(host="localhost", user="root",
password="root", port="3306",
                             database="customers")
db.get_account_balance(2)
db.get_account_details(4)
db.transfer(2, 4, 200)
db.display_all_accounts()
db.close_connection()
```



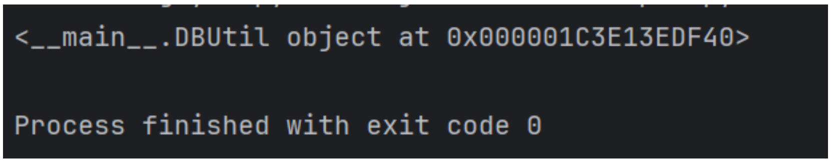
```
ACCOUNT DETAILS
['acc_no', 'acc_type', 'balance', 'customer', 'interest_rate', 'overdraft_limit'] (4, 'Current', 93200.0, 'Guna', None, 10000.0)
All Accounts Details:
{'acc_no': 1, 'acc_type': 'Savings', 'balance': 115000.0, 'customer': 'Amala', 'interest_rate': 0.05, 'overdraft_limit': None}
{'acc_no': 2, 'acc_type': 'Current', 'balance': 0.0, 'customer': 'Barath', 'interest_rate': None, 'overdraft_limit': 2000.0}
{'acc_no': 3, 'acc_type': 'ZeroBalance', 'balance': 45800.0, 'customer': 'Raajesh', 'interest_rate': None, 'overdraft_limit': None}
{'acc_no': 4, 'acc_type': 'Current', 'balance': 93400.0, 'customer': 'Guna', 'interest_rate': None, 'overdraft_limit': 10000.0}
{'acc_no': 5, 'acc_type': 'Savings', 'balance': 600000.0, 'customer': 'abarna', 'interest_rate': 0.08, 'overdraft_limit': None}
{'acc_no': 123, 'acc_type': 'savings', 'balance': 1000.0, 'customer': 'Gayathri', 'interest_rate': None, 'overdraft_limit': None}
{'acc_no': 125, 'acc_type': 'savings', 'balance': 1000.0, 'customer': 'Gayathri', 'interest_rate': None, 'overdraft_limit': None}
{'acc_no': 456, 'acc_type': 'current', 'balance': 14000.0, 'customer': 'Gowthami', 'interest_rate': None, 'overdraft_limit': None}
{'acc_no': 486, 'acc_type': 'current', 'balance': 14000.0, 'customer': 'Gowthami', 'interest_rate': None, 'overdraft_limit': None}

Process finished with exit code 0
```

Create **DBUtil** class and add the following method.

- **static getDBConn():Connection** Establish a connection to the database and return Connection reference



```
2.import mysql.connector
class DBUtil:
    def getDBConn(self):
        con=mysql.connector.connect(
            host="localhost",
            user="root",
            password="root",
            port="3306",
            database="customers"
        )
        con.cursor()

obj=DBUtil()
print(obj)
```

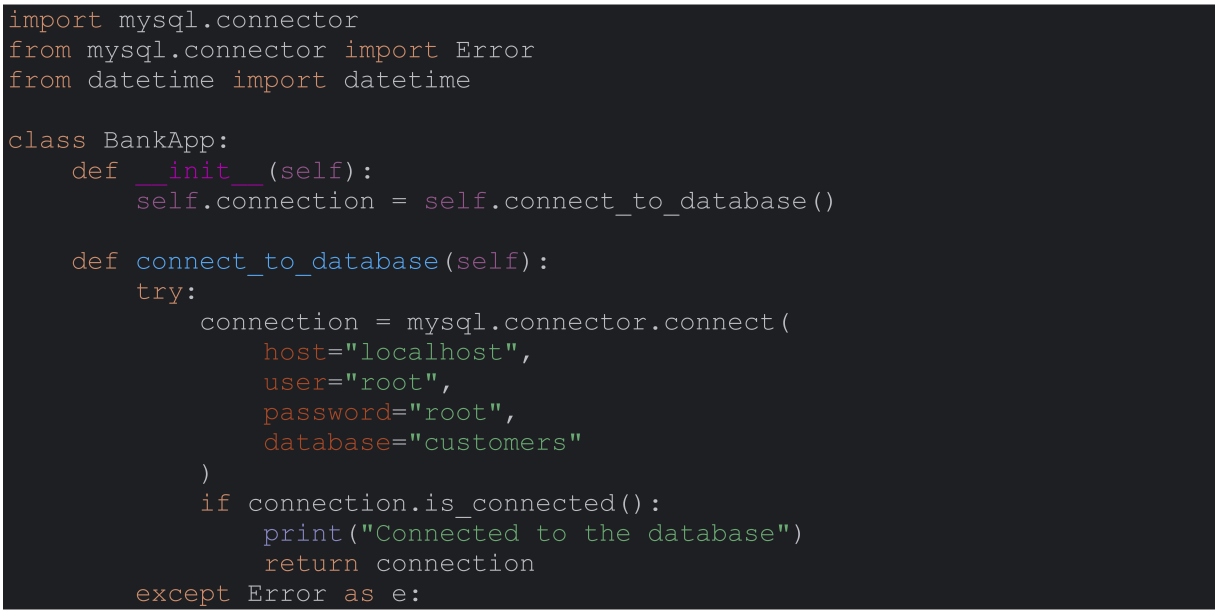


```
<__main__.DBUtil object at 0x000001C3E13EDF40>

Process finished with exit code 0
```

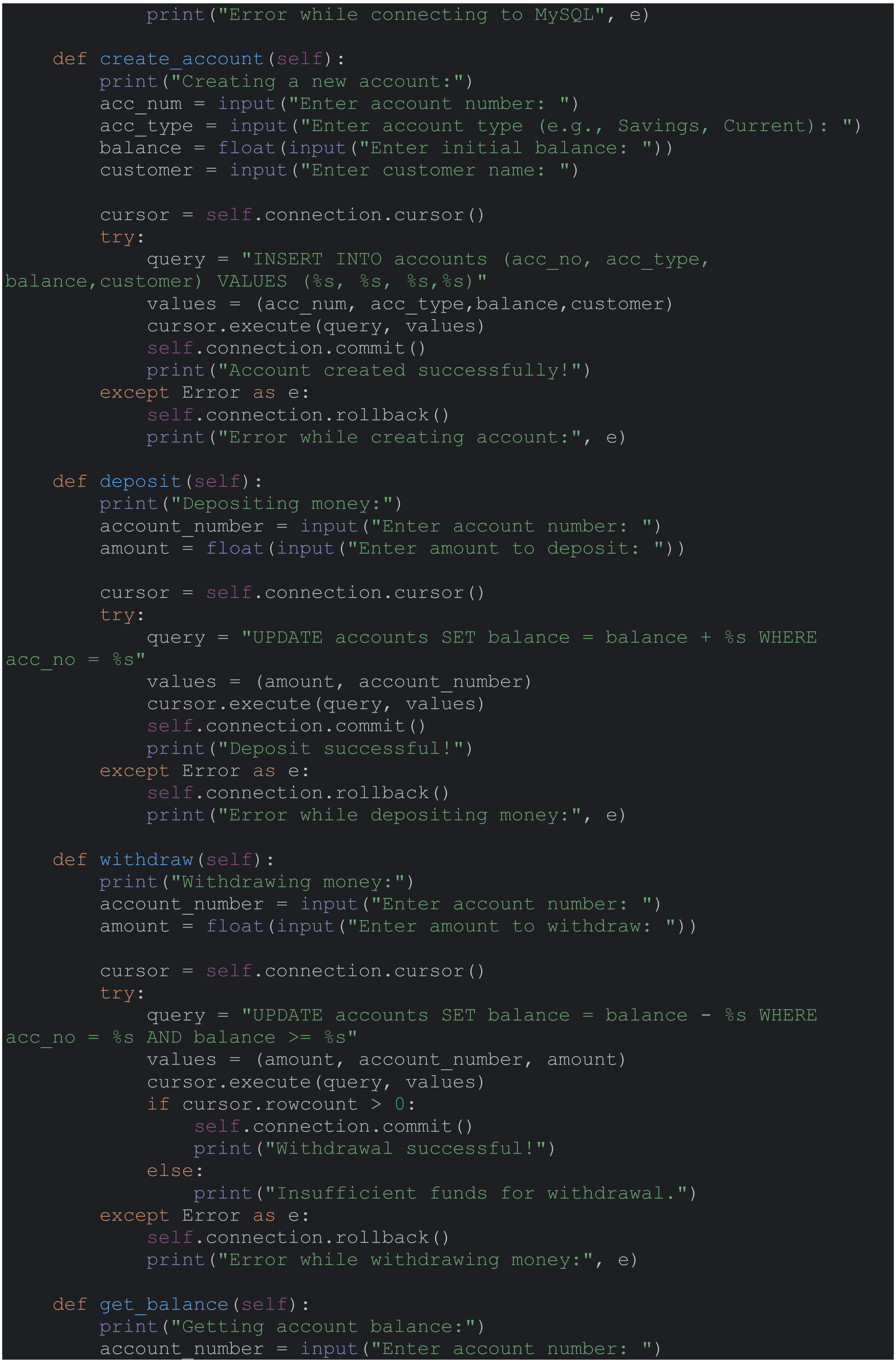
12.Create **BankApp** class and perform following operation:

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", "withdraw", "get_balance", "transfer", "getAccountDetails", "ListAccounts", "getTransactions" and "exit."

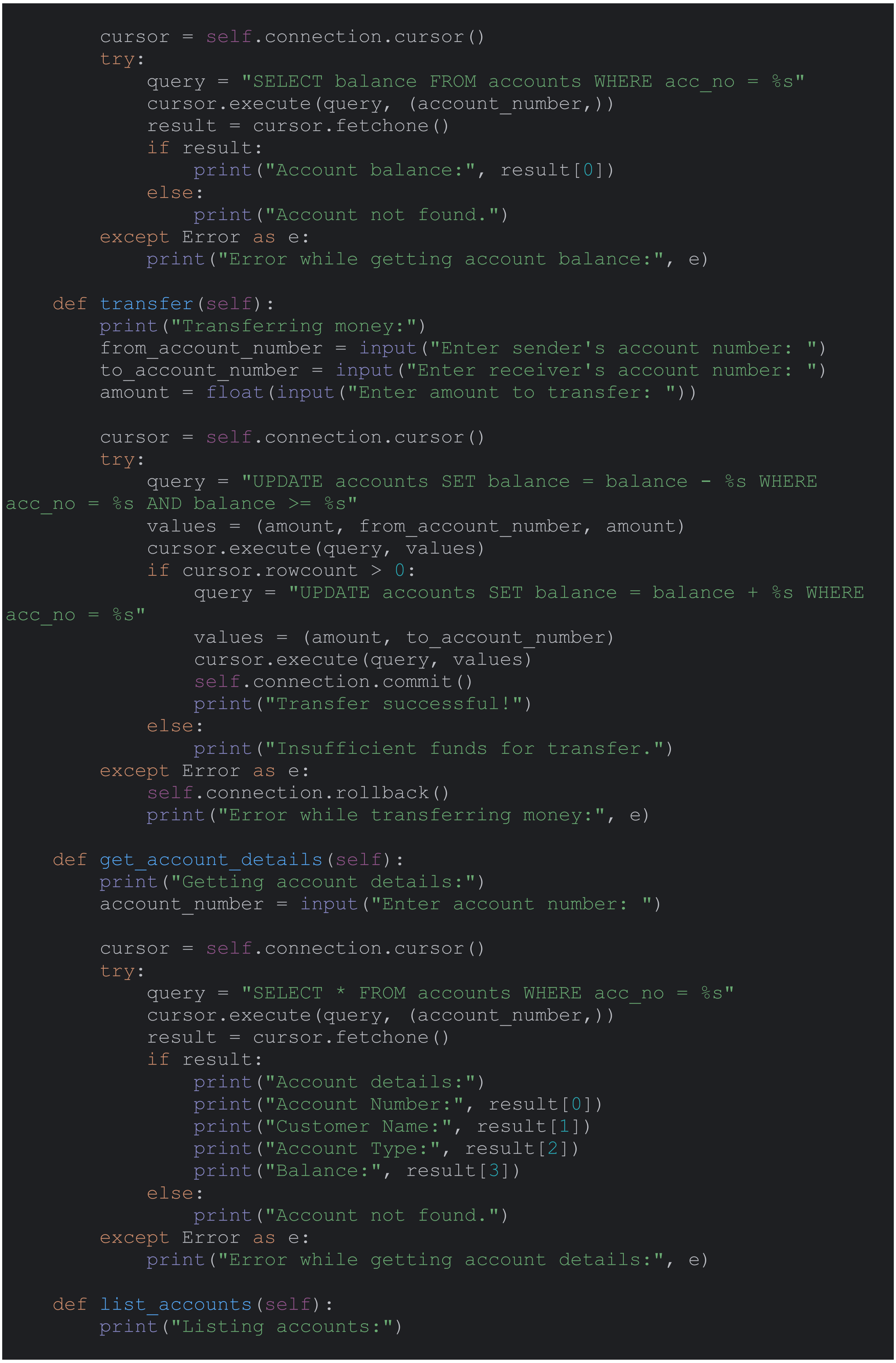
create_account should display sub menu to choose type of accounts and repeat this operation until user exit.



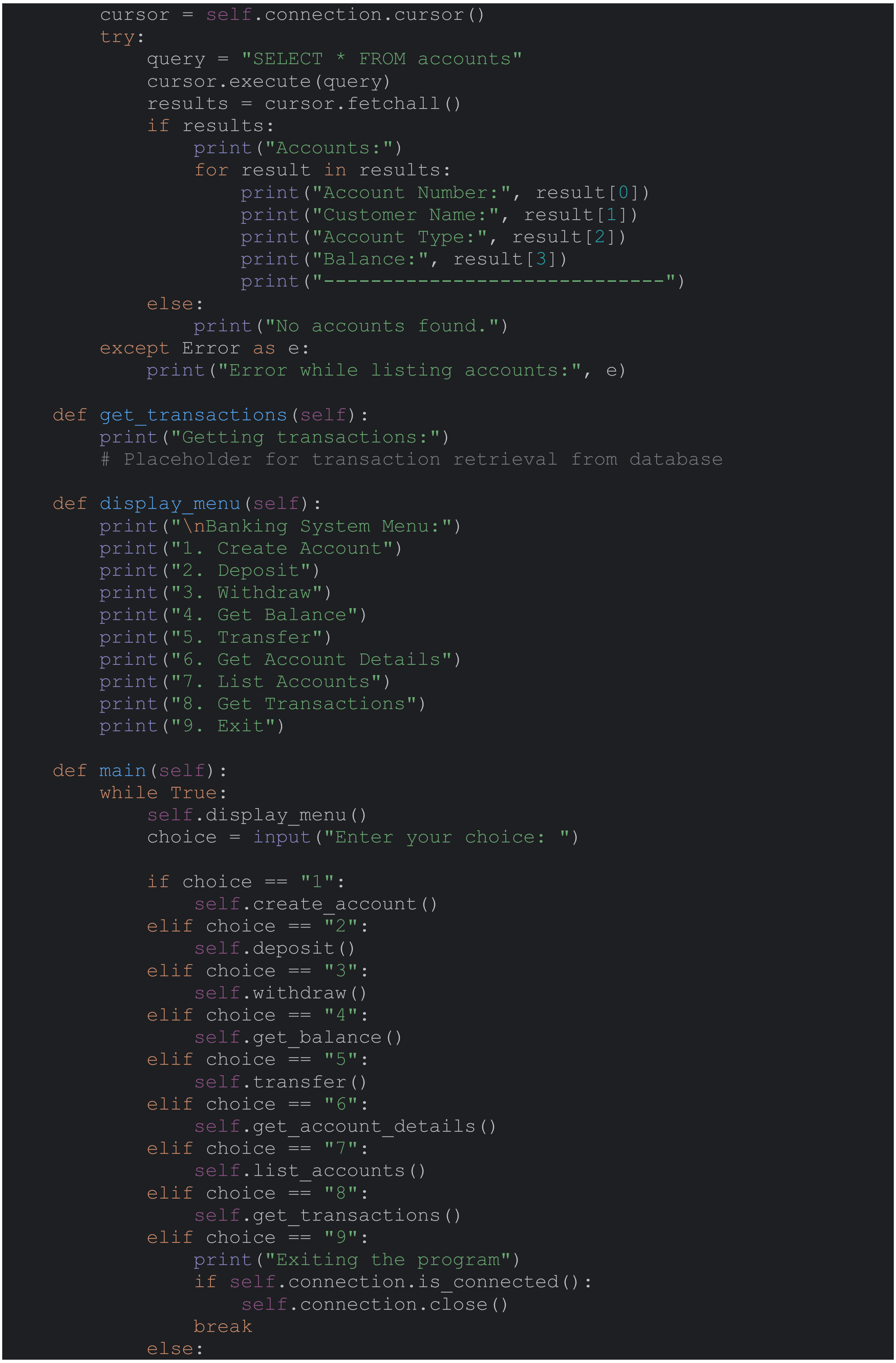
```
import mysql.connector
from mysql.connector import Error
from datetime import datetime

class BankApp:
    def __init__(self):
        self.connection = self.connect_to_database()

    def connect_to_database(self):
        try:
            connection = mysql.connector.connect(
                host="localhost",
                user="root",
                password="root",
                database="customers"
            )
            if connection.is_connected():
                print("Connected to the database")
                return connection
        except Error as e:
```



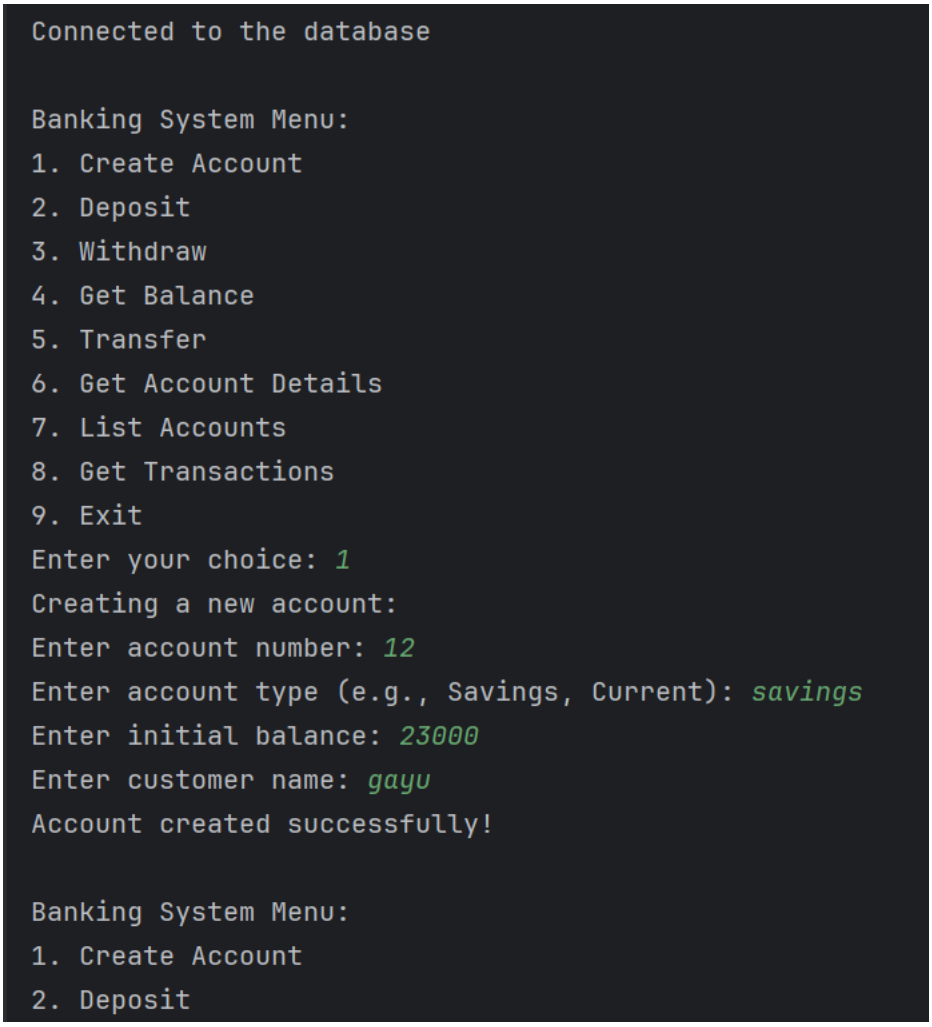
```
            print("Error while connecting to MySQL", e)

    def create_account(self):
        print("Creating a new account:")
        acc_num = input("Enter account number: ")
        acc_type = input("Enter account type (e.g., Savings, Current): ")
        balance = float(input("Enter initial balance: "))
        customer = input("Enter customer name: ")

        cursor = self.connection.cursor()
        try:
            query = "INSERT INTO accounts (acc_no, acc_type,
balance,customer) VALUES (%s, %s, %s,%s)"
            values = (acc_num, acc_type,balance,customer)
            cursor.execute(query, values)
            self.connection.commit()
            print("Account created successfully!")
        except Error as e:
            self.connection.rollback()
            print("Error while creating account:", e)

    def deposit(self):
        print("Depositing money:")
        account_number = input("Enter account number: ")
        amount = float(input("Enter amount to deposit: "))

        cursor = self.connection.cursor()
        try:
            query = "UPDATE accounts SET balance = balance + %s WHERE
acc_no = %s"
            values = (amount, account_number)
            cursor.execute(query, values)
            self.connection.commit()
            print("Deposit successful!")
        except Error as e:
            self.connection.rollback()
            print("Error while depositing money:", e)

    def withdraw(self):
        print("Withdrawing money:")
        account_number = input("Enter account number: ")
        amount = float(input("Enter amount to withdraw: "))

        cursor = self.connection.cursor()
        try:
            query = "UPDATE accounts SET balance = balance - %s WHERE
acc_no = %s AND balance >= %s"
            values = (amount, account_number, amount)
            cursor.execute(query, values)
            if cursor.rowcount > 0:
                self.connection.commit()
                print("Withdrawal successful!")
            else:
                print("Insufficient funds for withdrawal.")
        except Error as e:
            self.connection.rollback()
            print("Error while withdrawing money:", e)

    def get_balance(self):
        print("Getting account balance:")
        account_number = input("Enter account number: ")
```



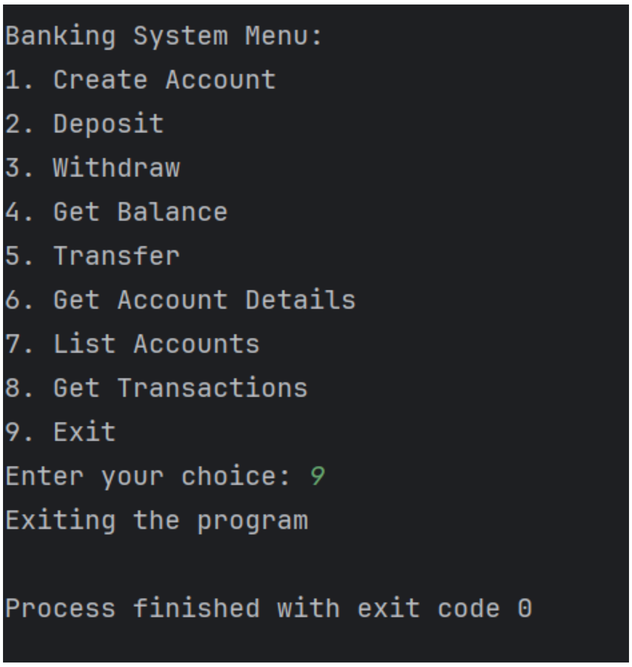
```
        cursor = self.connection.cursor()
        try:
            query = "SELECT balance FROM accounts WHERE acc_no = %s"
            cursor.execute(query, (account_number,))
            result = cursor.fetchone()
            if result:
                print("Account balance:", result[0])
            else:
                print("Account not found.")
        except Error as e:
            print("Error while getting account balance:", e)

    def transfer(self):
        print("Transferring money:")
        from_account_number = input("Enter sender's account number: ")
        to_account_number = input("Enter receiver's account number: ")
        amount = float(input("Enter amount to transfer: "))

        cursor = self.connection.cursor()
        try:
            query = "UPDATE accounts SET balance = balance - %s WHERE
acc_no = %s AND balance >= %s"
            values = (amount, from_account_number, amount)
            cursor.execute(query, values)
            if cursor.rowcount > 0:
                query = "UPDATE accounts SET balance = balance + %s WHERE
acc_no = %s"
                values = (amount, to_account_number)
                cursor.execute(query, values)
                self.connection.commit()
                print("Transfer successful!")
            else:
                print("Insufficient funds for transfer.")
        except Error as e:
            self.connection.rollback()
            print("Error while transferring money:", e)

    def get_account_details(self):
        print("Getting account details:")
        account_number = input("Enter account number: ")

        cursor = self.connection.cursor()
        try:
            query = "SELECT * FROM accounts WHERE acc_no = %s"
            cursor.execute(query, (account_number,))
            result = cursor.fetchone()
            if result:
                print("Account details:")
                print("Account Number:", result[0])
                print("Customer Name:", result[1])
                print("Account Type:", result[2])
                print("Balance:", result[3])
            else:
                print("Account not found.")
        except Error as e:
            print("Error while getting account details:", e)

    def list_accounts(self):
        print("Listing accounts:")
```



```
        cursor = self.connection.cursor()
        try:
            query = "SELECT * FROM accounts"
            cursor.execute(query)
            results = cursor.fetchall()
            if results:
                print("Accounts:")
                for result in results:
                    print("Account Number:", result[0])
                    print("Customer Name:", result[1])
                    print("Account Type:", result[2])
                    print("Balance:", result[3])
                    print("-----------------------------")
            else:
                print("No accounts found.")
        except Error as e:
            print("Error while listing accounts:", e)

    def get_transactions(self):
        print("Getting transactions:")
        # Placeholder for transaction retrieval from database

    def display_menu(self):
        print("\nBanking System Menu:")
        print("1. Create Account")
        print("2. Deposit")
        print("3. Withdraw")
        print("4. Get Balance")
        print("5. Transfer")
        print("6. Get Account Details")
        print("7. List Accounts")
        print("8. Get Transactions")
        print("9. Exit")

    def main(self):
        while True:
            self.display_menu()
            choice = input("Enter your choice: ")

            if choice == "1":
                self.create_account()
            elif choice == "2":
                self.deposit()
            elif choice == "3":
                self.withdraw()
            elif choice == "4":
                self.get_balance()
            elif choice == "5":
                self.transfer()
            elif choice == "6":
                self.get_account_details()
            elif choice == "7":
                self.list_accounts()
            elif choice == "8":
                self.get_transactions()
            elif choice == "9":
                print("Exiting the program")
                if self.connection.is_connected():
                    self.connection.close()
                break
            else:
                print("Invalid choice. Please try again.")


bank_app = BankApp()
bank_app.main()
```



```
Connected to the database

Banking System Menu:
1. Create Account
2. Deposit
3. Withdraw
4. Get Balance
5. Transfer
6. Get Account Details
7. List Accounts
8. Get Transactions
9. Exit
Enter your choice: 1
Creating a new account:
Enter account number: 12
Enter account type (e.g., Savings, Current): savings
Enter initial balance: 23000
Enter customer name: gayu
Account created successfully!

Banking System Menu:
1. Create Account
2. Deposit
```



```
Banking System Menu:
1. Create Account
2. Deposit
3. Withdraw
4. Get Balance
5. Transfer
6. Get Account Details
7. List Accounts
8. Get Transactions
9. Exit
Enter your choice: 9
Exiting the program

Process finished with exit code 0
```

Task 12: Exception Handling
throw the exception whenever needed and Handle in main method,

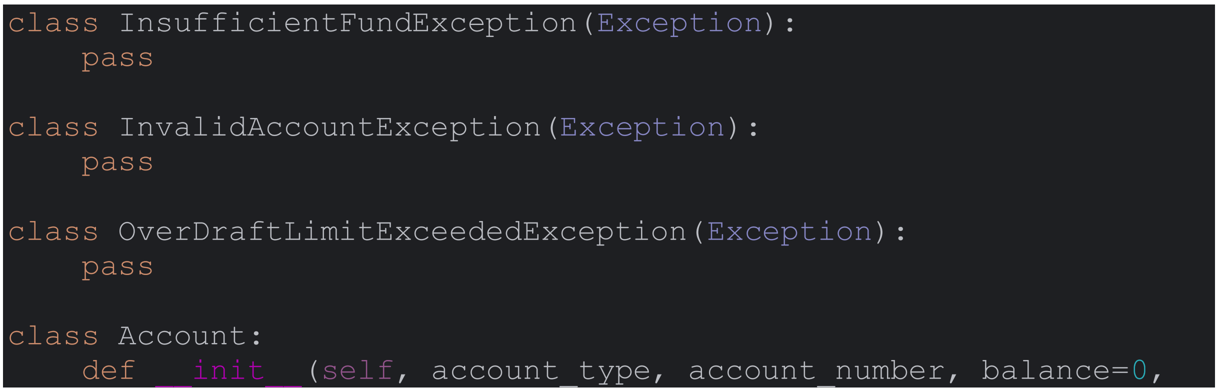
1. **InsufficientFundException** throw this exception when user try to withdraw amount or transfer amount to another account and the account runs out of money in the account.

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

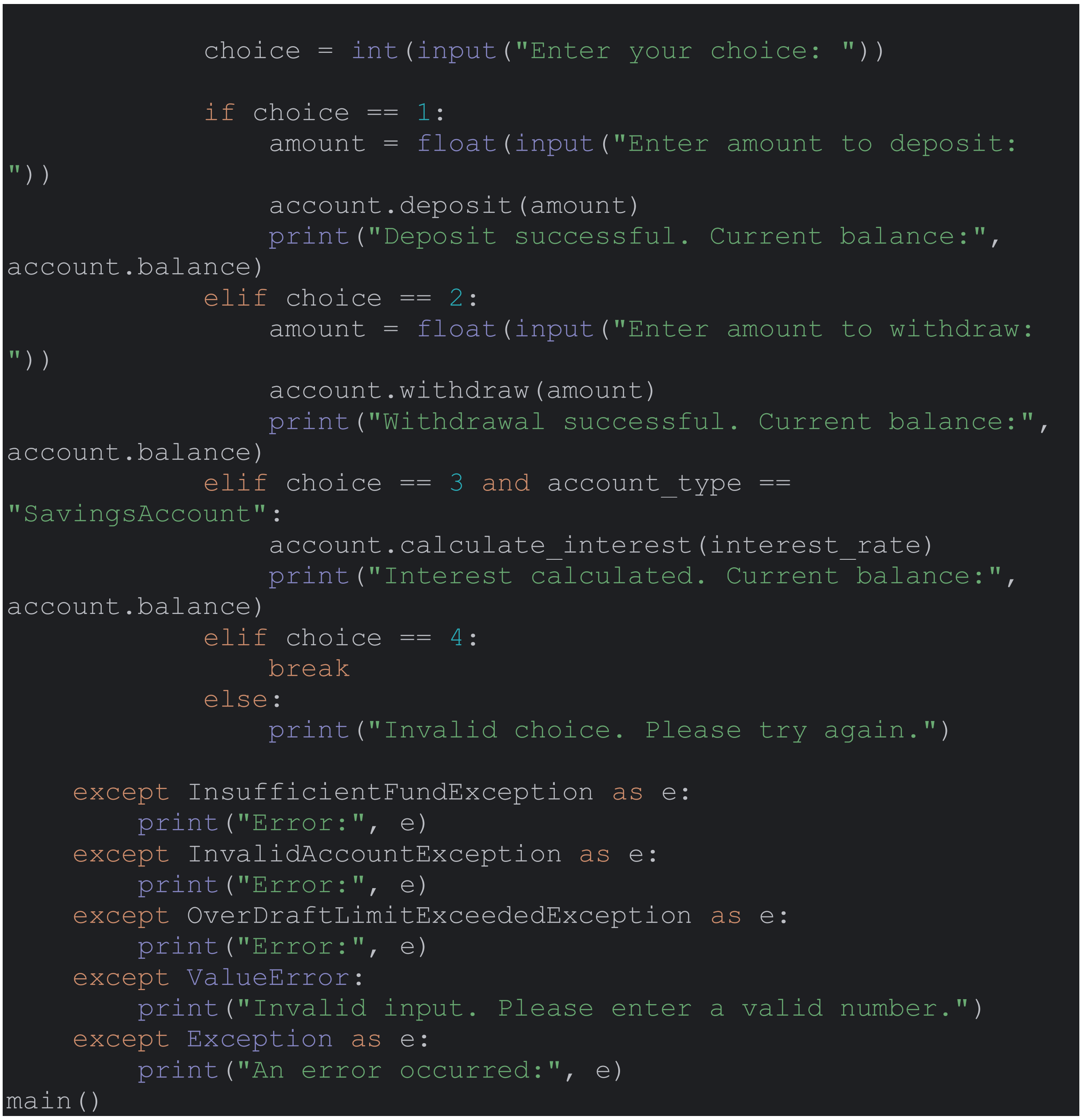
3. **OverDraftLimitExcededException** thow this exception when current account customer try to with draw amount from the current account.

4. **NullPointerException** handle in main method**.**

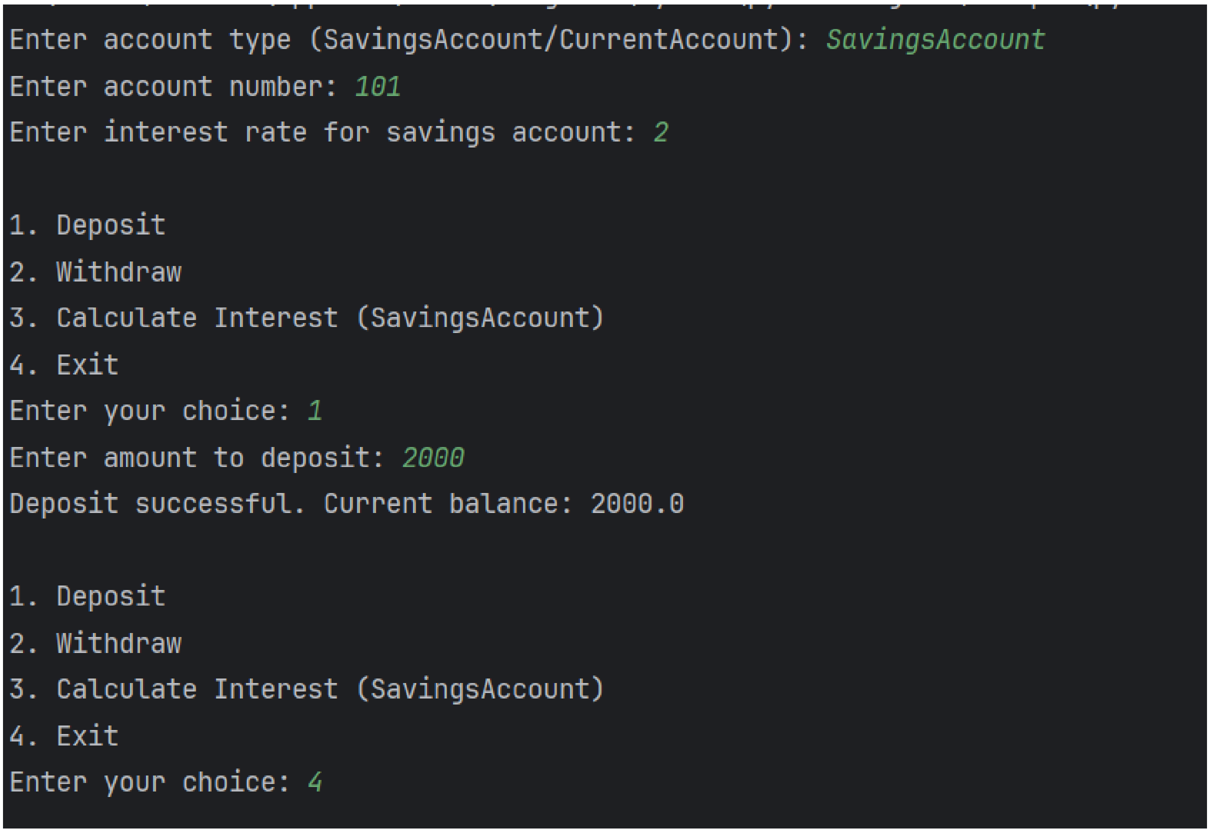
Throw these exceptions from the methods in HMBank class. Make necessary changes to accommodate these exception in the source code. Handle all these exceptions from the main program.



```
class InsufficientFundException(Exception):
    pass

class InvalidAccountException(Exception):
    pass

class OverDraftLimitExceededException(Exception):
    pass

class Account:
    def __init__(self, account_type, account_number, balance=0,
overdraft_limit=0):
        self.account_type = account_type
        self.account_number = account_number
        self.balance = balance
        self.overdraft_limit = overdraft_limit

    def deposit(self, amount):
        self.balance += amount

    def withdraw(self, amount):
        if self.account_type == "SavingsAccount":
            if self.balance < amount:
                raise InsufficientFundException("Insufficient balance in the account.")
            else:
                self.balance -= amount
        elif self.account_type == "CurrentAccount":
            if amount > (self.balance + self.overdraft_limit):
                raise OverDraftLimitExceededException("Withdrawal amount exceeds the overdraft limit.")
            else:
                self.balance -= amount

    def calculate_interest(self, interest_rate):
        if self.account_type == "SavingsAccount":
            interest = self.balance * interest_rate
            self.balance += interest

def main():
    try:
        account_type = input("Enter account type (SavingsAccount/CurrentAccount): ")
        account_number = int(input("Enter account number: "))
        if account_type not in ["SavingsAccount", "CurrentAccount"]:
            raise InvalidAccountException("Invalid account type.")

        if account_type == "SavingsAccount":
            interest_rate = float(input("Enter interest rate for savings account: "))
            account = Account(account_type, account_number)
        elif account_type == "CurrentAccount":
            overdraft_limit = float(input("Enter overdraft limit for current account: "))
            account = Account(account_type, account_number, overdraft_limit=overdraft_limit)

        while True:
            print("\n1. Deposit")
            print("2. Withdraw")
            print("3. Calculate Interest (SavingsAccount)")
            print("4. Exit")
```



```
            choice = int(input("Enter your choice: "))

            if choice == 1:
                amount = float(input("Enter amount to deposit:
"))
                account.deposit(amount)
                print("Deposit successful. Current balance:",
account.balance)
            elif choice == 2:
                amount = float(input("Enter amount to withdraw:
"))
                account.withdraw(amount)
                print("Withdrawal successful. Current balance:",
account.balance)
            elif choice == 3 and account_type ==
"SavingsAccount":
                account.calculate_interest(interest_rate)
                print("Interest calculated. Current balance:",
account.balance)
            elif choice == 4:
                break
            else:
                print("Invalid choice. Please try again.")

    except InsufficientFundException as e:
        print("Error:", e)
    except InvalidAccountException as e:
        print("Error:", e)
    except OverDraftLimitExceededException as e:
        print("Error:", e)
    except ValueError:
        print("Invalid input. Please enter a valid number.")
    except Exception as e:
        print("An error occurred:", e)
main()
```

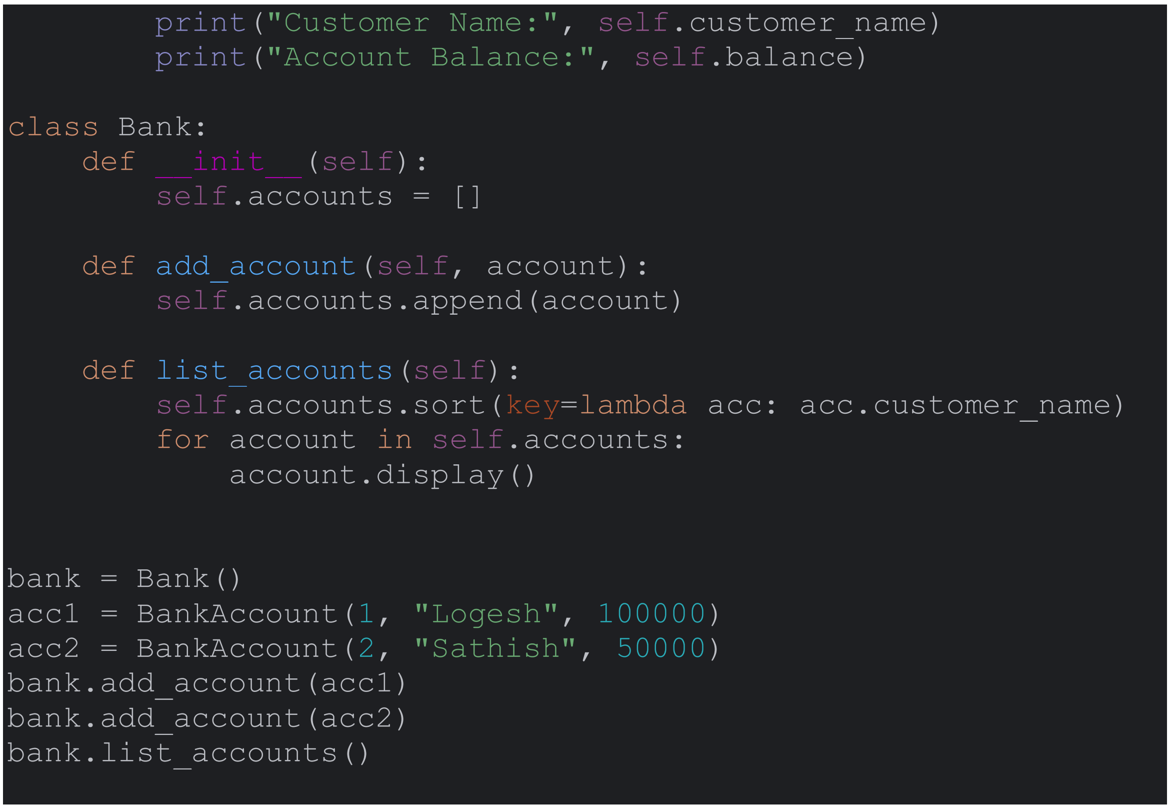


```
Enter account type (SavingsAccount/CurrentAccount): SavingsAccount
Enter account number: 101
Enter interest rate for savings account: 2

1. Deposit
2. Withdraw
3. Calculate Interest (SavingsAccount)
4. Exit
Enter your choice: 1
Enter amount to deposit: 2000
Deposit successful. Current balance: 2000.0

1. Deposit
2. Withdraw
3. Calculate Interest (SavingsAccount)
4. Exit
Enter your choice: 4
```

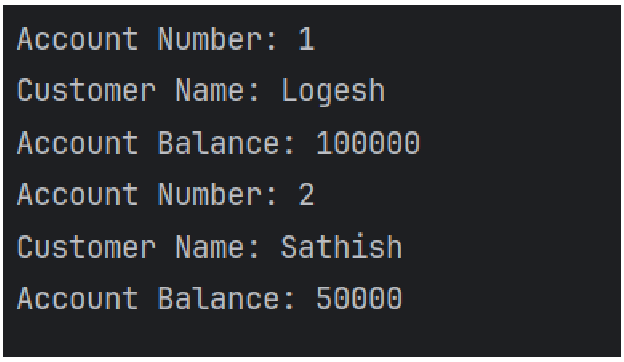
Task 13: Collection

1. From the previous task change the **HMBank** attribute Accounts to List of Accounts and perform the same operation.

```
2. class BankAccount:
       def __init__(self, account_number, customer_name,
   balance):
           self.account_number = account_number
           self.customer_name = customer_name
           self.balance = balance

       def deposit(self, amount):
           self.balance += amount

       def withdraw(self, amount):
           if amount <= self.balance:
               self.balance -= amount
               print("Balance After withdrawal: ",
   self.balance)
           else:
               print("Insufficient balance")

       def interest(self):
           intrate = float(input("Enter the interest rate: "))
           intamount = self.balance * (intrate / 100)
           self.balance += intamount
           print("Balance with interest: ", self.balance)
        def display(self):
         print("Account Number:", self.account_number)
```



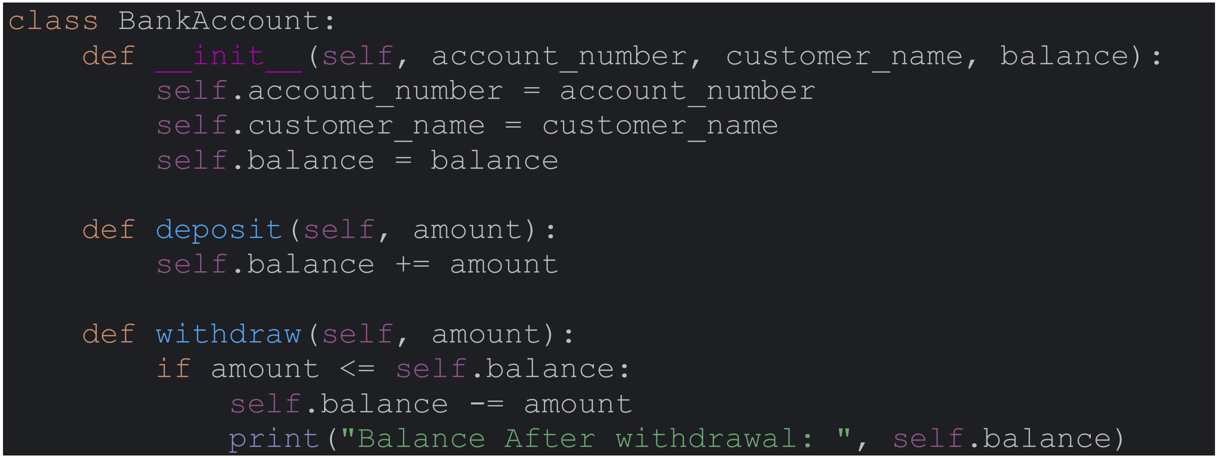
```
        print("Customer Name:", self.customer_name)
        print("Account Balance:", self.balance)

class Bank:
    def __init__(self):
        self.accounts = []

    def add_account(self, account):
        self.accounts.append(account)

    def list_accounts(self):
        self.accounts.sort(key=lambda acc: acc.customer_name)
        for account in self.accounts:
            account.display()


bank = Bank()
acc1 = BankAccount(1, "Logesh", 100000)
acc2 = BankAccount(2, "Sathish", 50000)
bank.add_account(acc1)
bank.add_account(acc2)
bank.list_accounts()
```



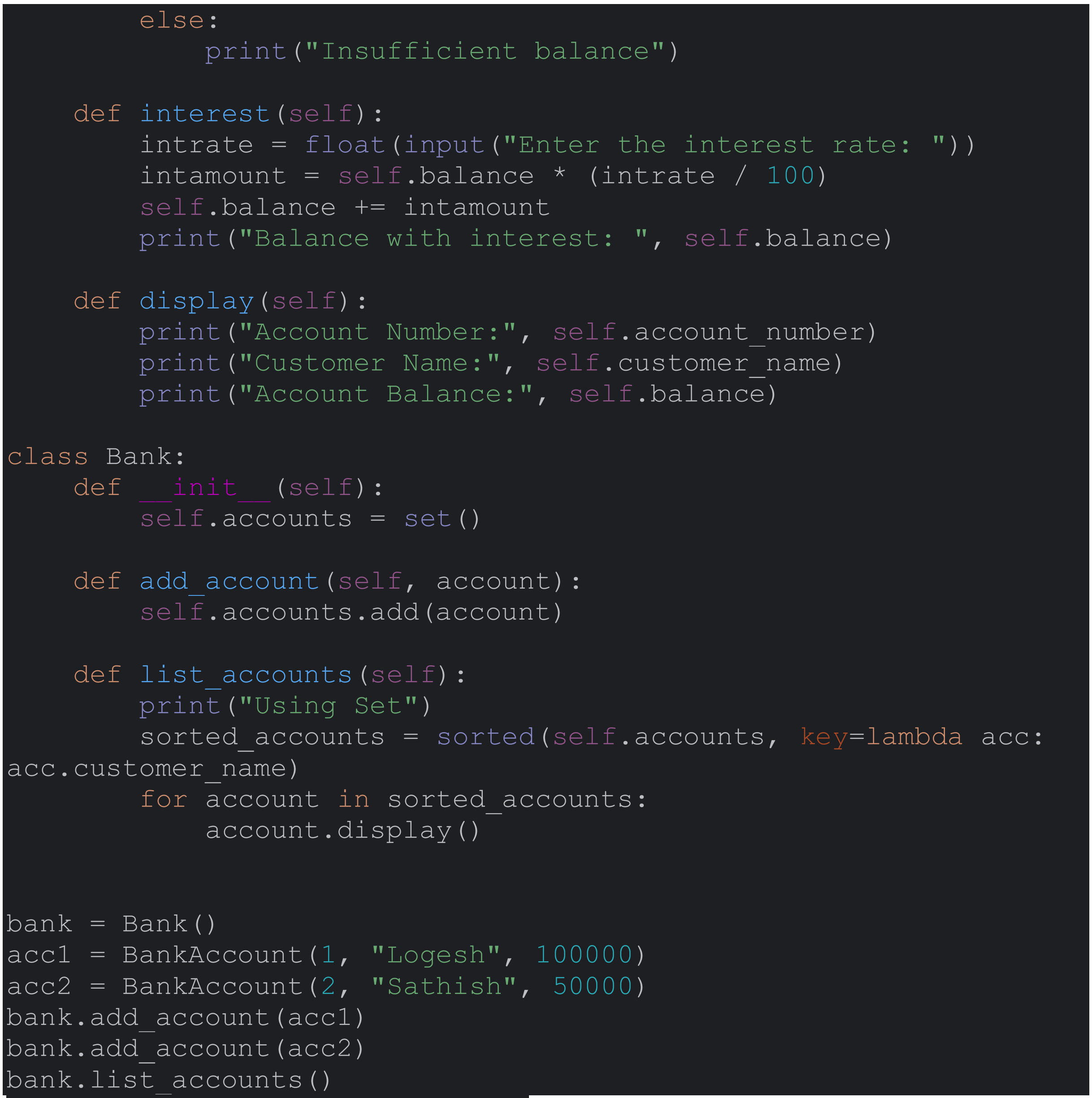
```
Account Number: 1
Customer Name: Logesh
Account Balance: 100000
Account Number: 2
Customer Name: Sathish
Account Balance: 50000
```

2. From the previous task change the **HMBank** attribute Accounts to Set of Accounts and perform the same operation. Avoid adding duplicate Account object to the set.

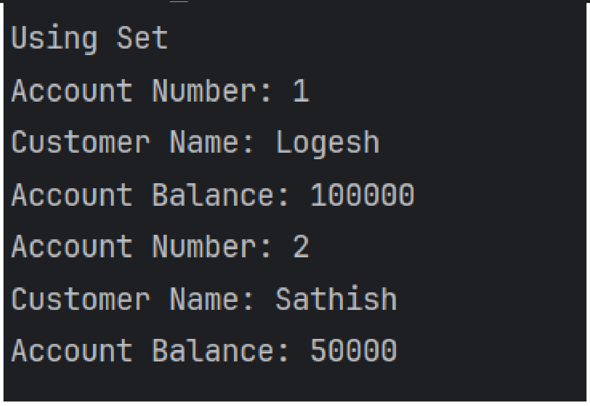
Create Comparator<Account> object to sort the accounts based on customer name when listAccounts() method called.



```
class BankAccount:
    def __init__(self, account_number, customer_name, balance):
        self.account_number = account_number
        self.customer_name = customer_name
        self.balance = balance

    def deposit(self, amount):
        self.balance += amount

    def withdraw(self, amount):
        if amount <= self.balance:
            self.balance -= amount
            print("Balance After withdrawal: ", self.balance)
```

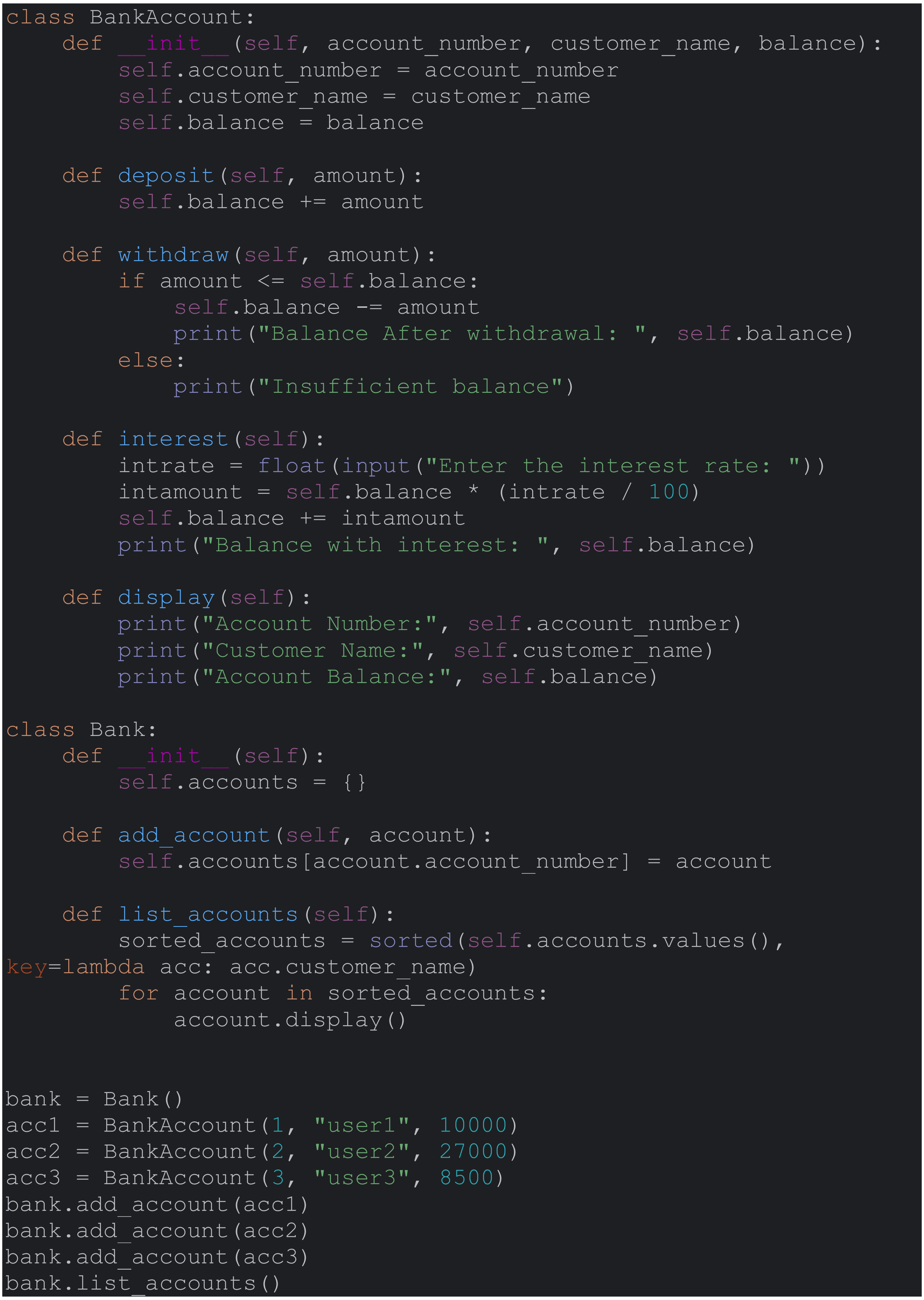


```
        else:
            print("Insufficient balance")

    def interest(self):
        intrate = float(input("Enter the interest rate: "))
        intamount = self.balance * (intrate / 100)
        self.balance += intamount
        print("Balance with interest: ", self.balance)

    def display(self):
        print("Account Number:", self.account_number)
        print("Customer Name:", self.customer_name)
        print("Account Balance:", self.balance)

class Bank:
    def __init__(self):
        self.accounts = set()

    def add_account(self, account):
        self.accounts.add(account)

    def list_accounts(self):
        print("Using Set")
        sorted_accounts = sorted(self.accounts, key=lambda acc:
acc.customer_name)
        for account in sorted_accounts:
            account.display()


bank = Bank()
acc1 = BankAccount(1, "Logesh", 100000)
acc2 = BankAccount(2, "Sathish", 50000)
bank.add_account(acc1)
bank.add_account(acc2)
bank.list_accounts()
```

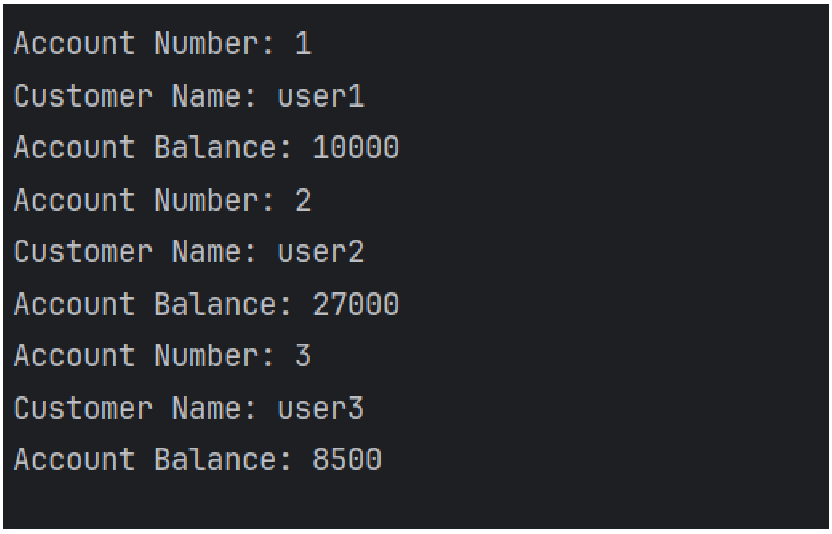


```
Using Set
Account Number: 1
Customer Name: Logesh
Account Balance: 100000
Account Number: 2
Customer Name: Sathish
Account Balance: 50000
```

3. From the previous task change the HMBank attribute Accounts to HashMap of Accounts and perform the same operation.



```
class BankAccount:
    def __init__(self, account_number, customer_name, balance):
        self.account_number = account_number
        self.customer_name = customer_name
        self.balance = balance

    def deposit(self, amount):
        self.balance += amount

    def withdraw(self, amount):
        if amount <= self.balance:
            self.balance -= amount
            print("Balance After withdrawal: ", self.balance)
        else:
            print("Insufficient balance")

    def interest(self):
        intrate = float(input("Enter the interest rate: "))
        intamount = self.balance * (intrate / 100)
        self.balance += intamount
        print("Balance with interest: ", self.balance)

    def display(self):
        print("Account Number:", self.account_number)
        print("Customer Name:", self.customer_name)
        print("Account Balance:", self.balance)

class Bank:
    def __init__(self):
        self.accounts = {}

    def add_account(self, account):
        self.accounts[account.account_number] = account

    def list_accounts(self):
        sorted_accounts = sorted(self.accounts.values(),
key=lambda acc: acc.customer_name)
        for account in sorted_accounts:
            account.display()


bank = Bank()
acc1 = BankAccount(1, "user1", 10000)
acc2 = BankAccount(2, "user2", 27000)
acc3 = BankAccount(3, "user3", 8500)
bank.add_account(acc1)
bank.add_account(acc2)
bank.add_account(acc3)
bank.list_accounts()
```

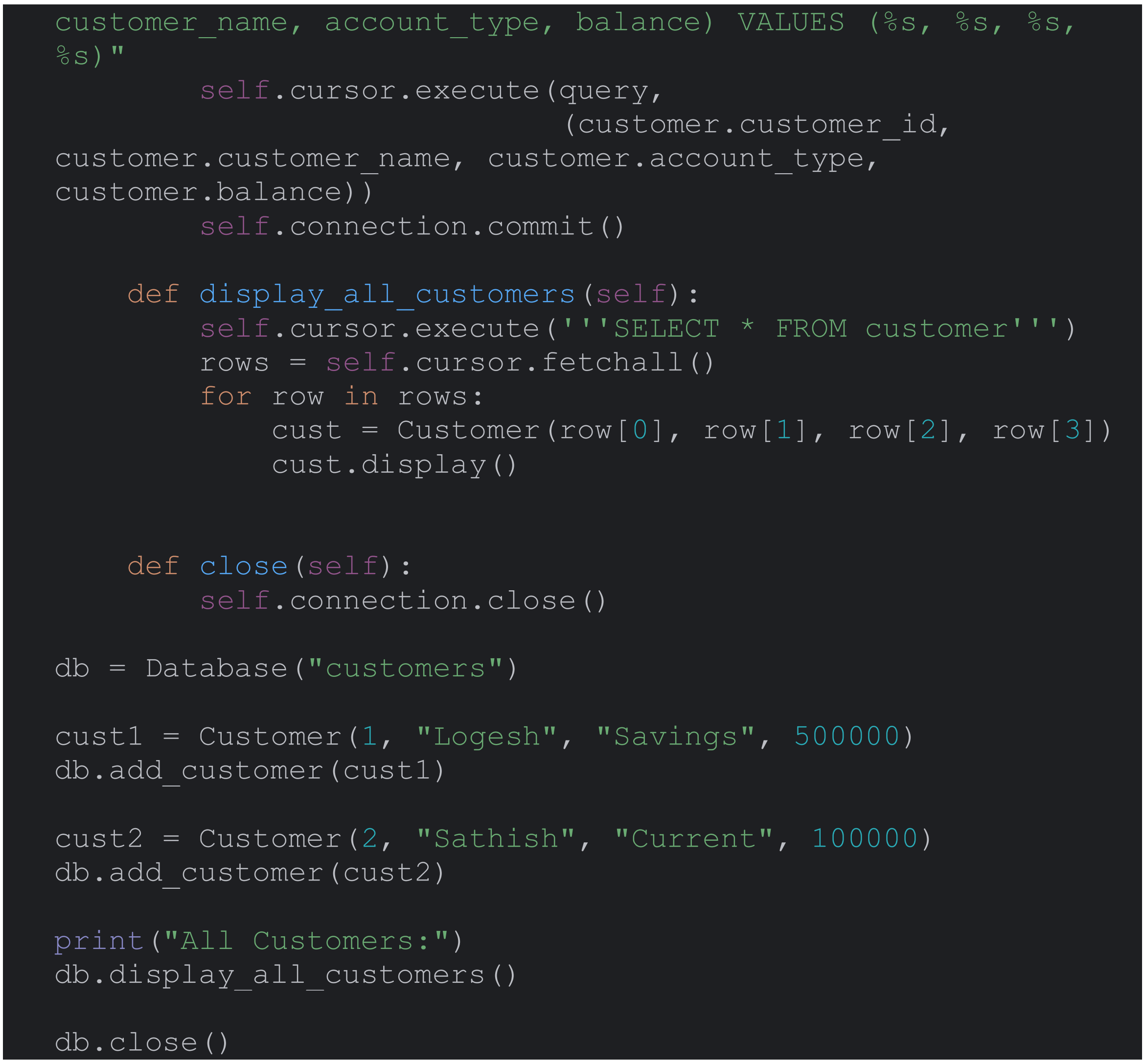


```
Account Number: 1
Customer Name: user1
Account Balance: 10000
Account Number: 2
Customer Name: user2
Account Balance: 27000
Account Number: 3
Customer Name: user3
Account Balance: 8500
```

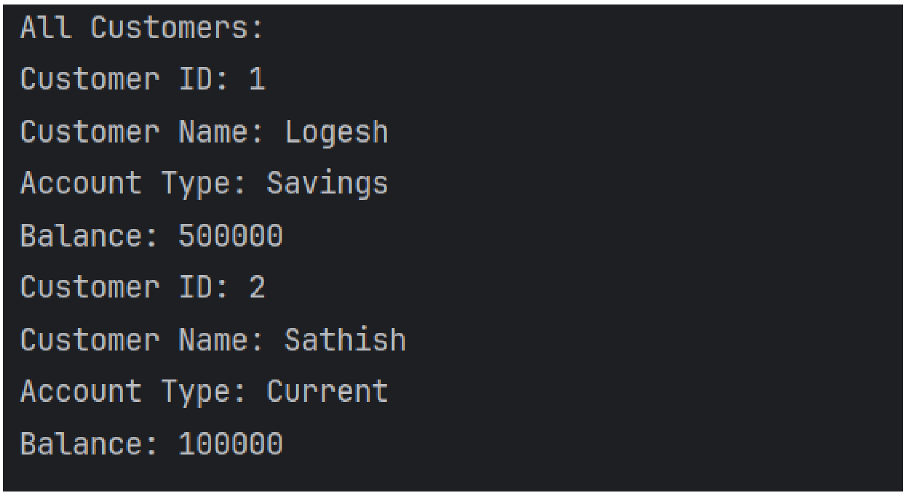
Task 14: Database Connectivity.

1. Create a **'Customer'** class as mentioned above task.

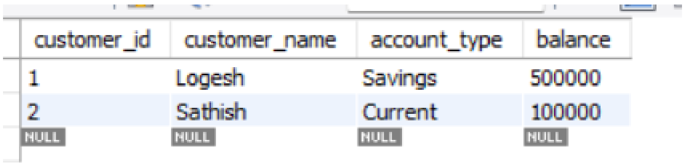
```
2. import mysql.connector
   class Customer:
       def __init__(self, customer_id, customer_name,
   account_type, balance):
           self.customer_id = customer_id
           self.customer_name = customer_name
           self.account_type = account_type
           self.balance = balance
       def display(self):
           print("Customer ID:", self.customer_id)
           print("Customer Name:", self.customer_name)
           print("Account Type:", self.account_type)
           print("Balance:", self.balance)


   class Database:
       def __init__(self,db_name):
           self.connection = mysql.connector.connect(

   host="localhost",user="root",password="root",port="3306",dat
   abase=db_name
           )
           self.cursor = self.connection.cursor()
           self.cursor.execute('''CREATE TABLE IF NOT EXISTS
   customer
                                (customer_id int PRIMARY KEY,
                                customer_name text,
                                account_type text,
                                balance int)''')
           self.connection.commit()

       def add_customer(self, customer):
           query="INSERT INTO customer(customer_id,
```



```
customer_name, account_type, balance) VALUES (%s, %s, %s,
%s)"
        self.cursor.execute(query,
                            (customer.customer_id,
customer.customer_name, customer.account_type,
customer.balance))
        self.connection.commit()

    def display_all_customers(self):
        self.cursor.execute('''SELECT * FROM customer''')
        rows = self.cursor.fetchall()
        for row in rows:
            cust = Customer(row[0], row[1], row[2], row[3])
            cust.display()


    def close(self):
        self.connection.close()

db = Database("customers")

cust1 = Customer(1, "Logesh", "Savings", 500000)
db.add_customer(cust1)

cust2 = Customer(2, "Sathish", "Current", 100000)
db.add_customer(cust2)

print("All Customers:")
db.display_all_customers()

db.close()
```



```
All Customers:
Customer ID: 1
Customer Name: Logesh
Account Type: Savings
Balance: 500000
Customer ID: 2
Customer Name: Sathish
Account Type: Current
Balance: 100000
```



customer_id	customer_name	account_type	balance
1	Logesh	Savings	500000
2	Sathish	Current	100000
NULL	NULL	NULL	NULL

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

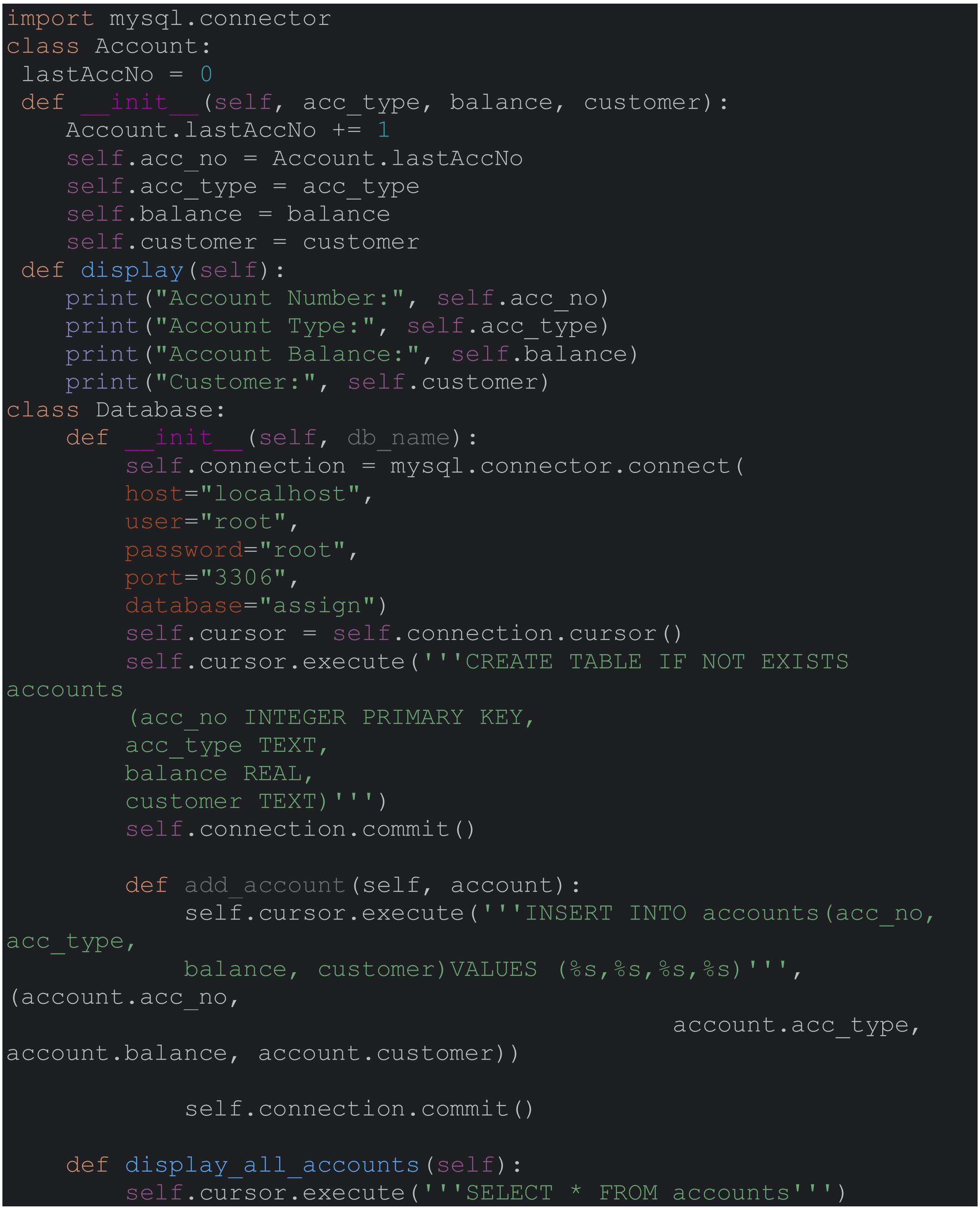
Account Number (a unique identifier).

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

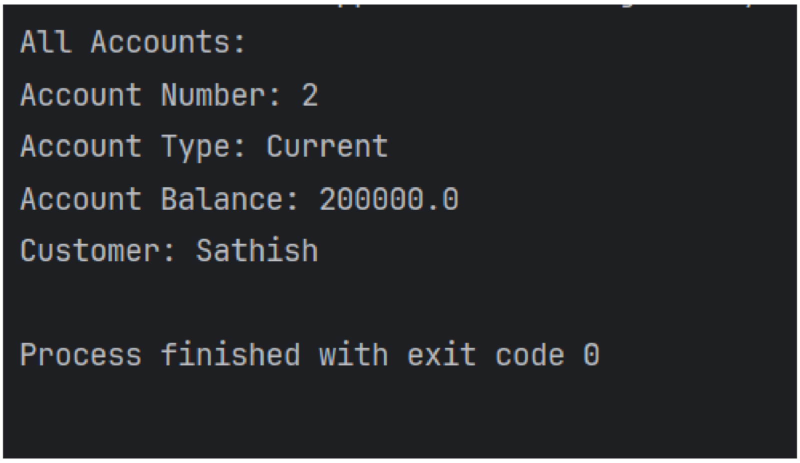
Account Balance

Customer (the customer who owns the account)

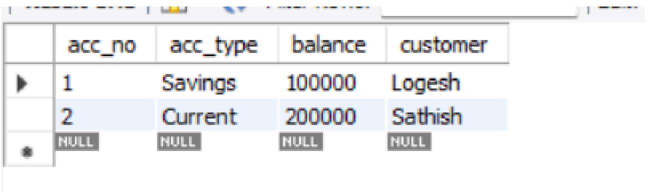
lastAccNo



```
import mysql.connector
class Account:
 lastAccNo = 0
 def __init__(self, acc_type, balance, customer):
    Account.lastAccNo += 1
    self.acc_no = Account.lastAccNo
    self.acc_type = acc_type
    self.balance = balance
    self.customer = customer
 def display(self):
    print("Account Number:", self.acc_no)
    print("Account Type:", self.acc_type)
    print("Account Balance:", self.balance)
    print("Customer:", self.customer)
class Database:
    def __init__(self, db_name):
        self.connection = mysql.connector.connect(
        host="localhost",
        user="root",
        password="root",
        port="3306",
        database="assign")
        self.cursor = self.connection.cursor()
        self.cursor.execute('''CREATE TABLE IF NOT EXISTS
accounts
        (acc_no INTEGER PRIMARY KEY,
        acc_type TEXT,
        balance REAL,
        customer TEXT)''')
        self.connection.commit()

        def add_account(self, account):
            self.cursor.execute('''INSERT INTO accounts(acc_no,
acc_type,
            balance, customer)VALUES (%s,%s,%s,%s)''',
(account.acc_no,
                                                        account.acc_type,
account.balance, account.customer))

            self.connection.commit()

    def display_all_accounts(self):
        self.cursor.execute('''SELECT * FROM accounts''')
        rows = self.cursor.fetchall()
        for row in rows:
            acc = Account(row[1], row[2], row[3])
            acc.acc_no = row[0]
        acc.display()

    def close(self):
        self.connection.close()


db = Database("assign")
acc1 = Account("Savings", 100000, "Logesh")
db.add_account(acc1)
acc2 = Account("Current", 200000, "Sathish")
db.add_account1(acc2)
print("All Accounts:")
db.display_all_accounts()
db.close()
```



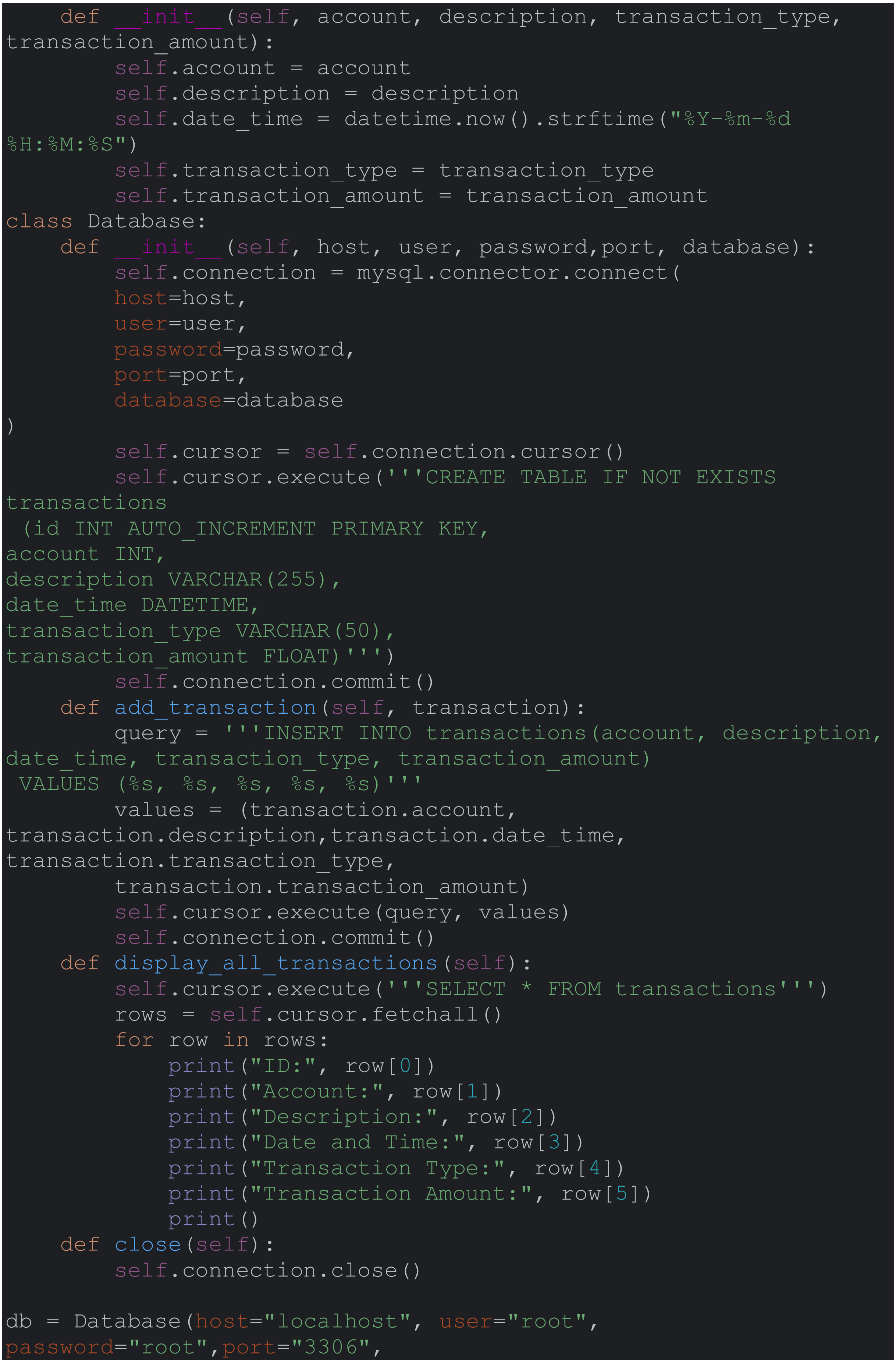
```
All Accounts:
Account Number: 2
Account Type: Current
Account Balance: 200000.0
Customer: Sathish

Process finished with exit code 0
```



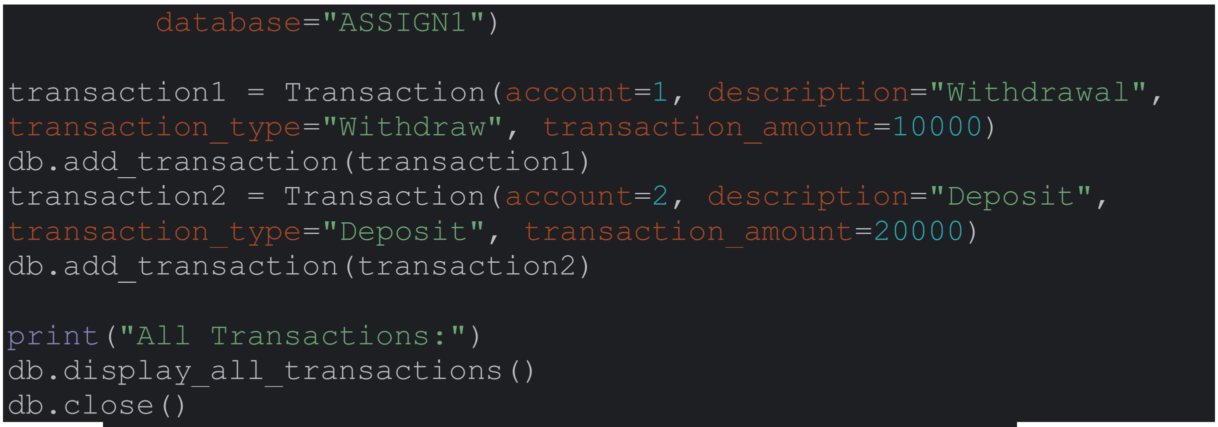
acc_no	acc_type	balance	customer
1	Savings	100000	Logesh
2	Current	200000	Sathish
NULL	NULL	NULL	NULL

- 3. Create a class **‘TRANSACTION’** that include following attributes Account
- Description
- Date and Time
- TransactionType(Withdraw, Deposit, Transfer)
- TransactionAmount

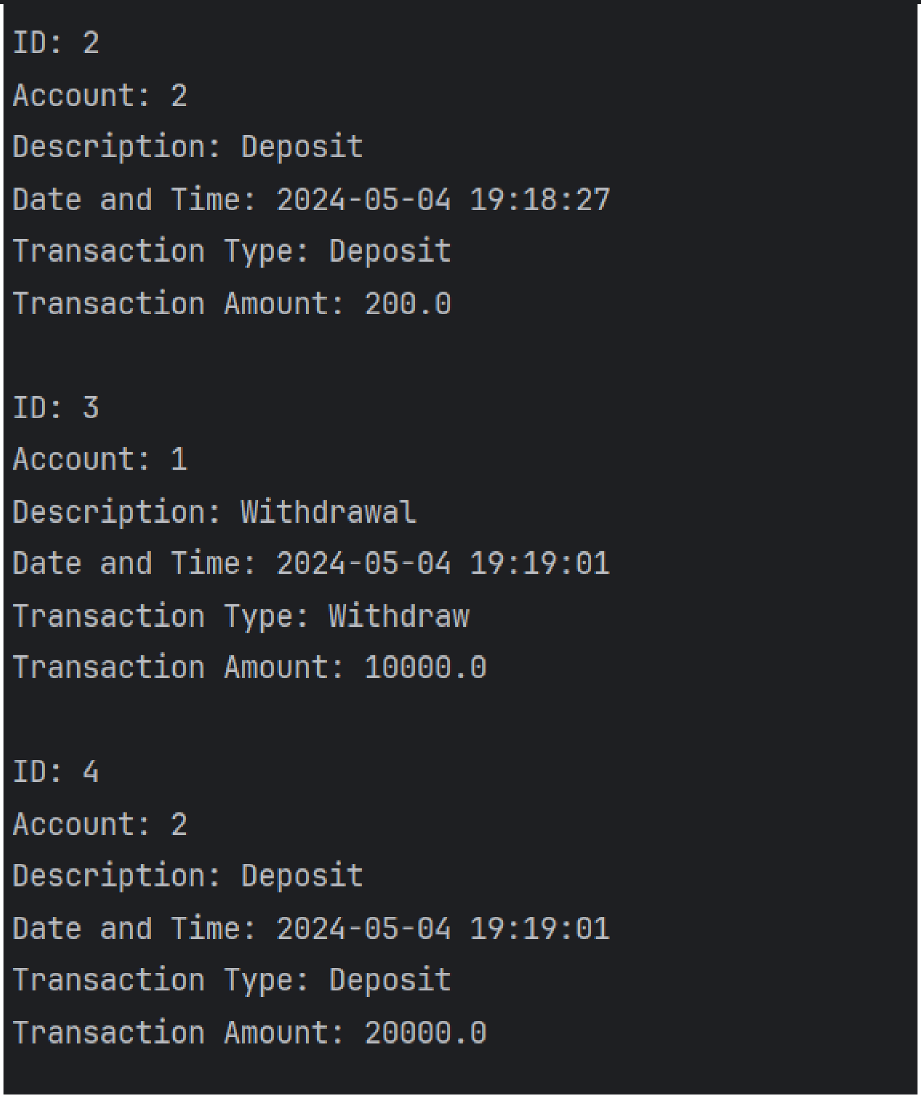
```
import mysql.connector
from datetime import datetime
class Transaction:
```



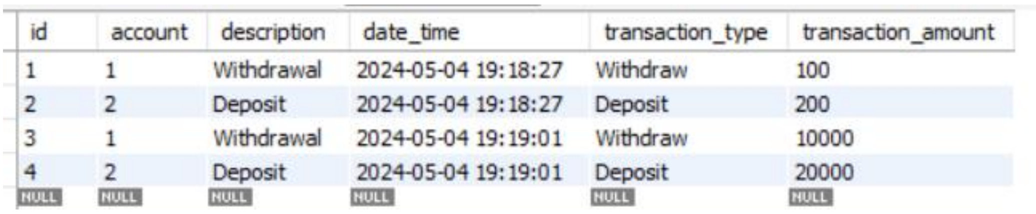
```
    def __init__(self, account, description, transaction_type,
transaction_amount):
        self.account = account
        self.description = description
        self.date_time = datetime.now().strftime("%Y-%m-%d
%H:%M:%S")
        self.transaction_type = transaction_type
        self.transaction_amount = transaction_amount
class Database:
    def __init__(self, host, user, password,port, database):
        self.connection = mysql.connector.connect(
        host=host,
        user=user,
        password=password,
        port=port,
        database=database
)
        self.cursor = self.connection.cursor()
        self.cursor.execute('''CREATE TABLE IF NOT EXISTS
transactions
 (id INT AUTO_INCREMENT PRIMARY KEY,
account INT,
description VARCHAR(255),
date_time DATETIME,
transaction_type VARCHAR(50),
transaction_amount FLOAT)''')
        self.connection.commit()
    def add_transaction(self, transaction):
        query = '''INSERT INTO transactions(account, description,
date_time, transaction_type, transaction_amount)
 VALUES (%s, %s, %s, %s, %s)'''
        values = (transaction.account,
transaction.description,transaction.date_time,
transaction.transaction_type,
        transaction.transaction_amount)
        self.cursor.execute(query, values)
        self.connection.commit()
    def display_all_transactions(self):
        self.cursor.execute('''SELECT * FROM transactions''')
        rows = self.cursor.fetchall()
        for row in rows:
            print("ID:", row[0])
            print("Account:", row[1])
            print("Description:", row[2])
            print("Date and Time:", row[3])
            print("Transaction Type:", row[4])
            print("Transaction Amount:", row[5])
            print()
    def close(self):
        self.connection.close()

db = Database(host="localhost", user="root",
password="root",port="3306",
```



```
        database="ASSIGN1")

transaction1 = Transaction(account=1, description="Withdrawal",
transaction_type="Withdraw", transaction_amount=10000)
db.add_transaction(transaction1)
transaction2 = Transaction(account=2, description="Deposit",
transaction_type="Deposit", transaction_amount=20000)
db.add_transaction(transaction2)

print("All Transactions:")
db.display_all_transactions()
db.close()
```



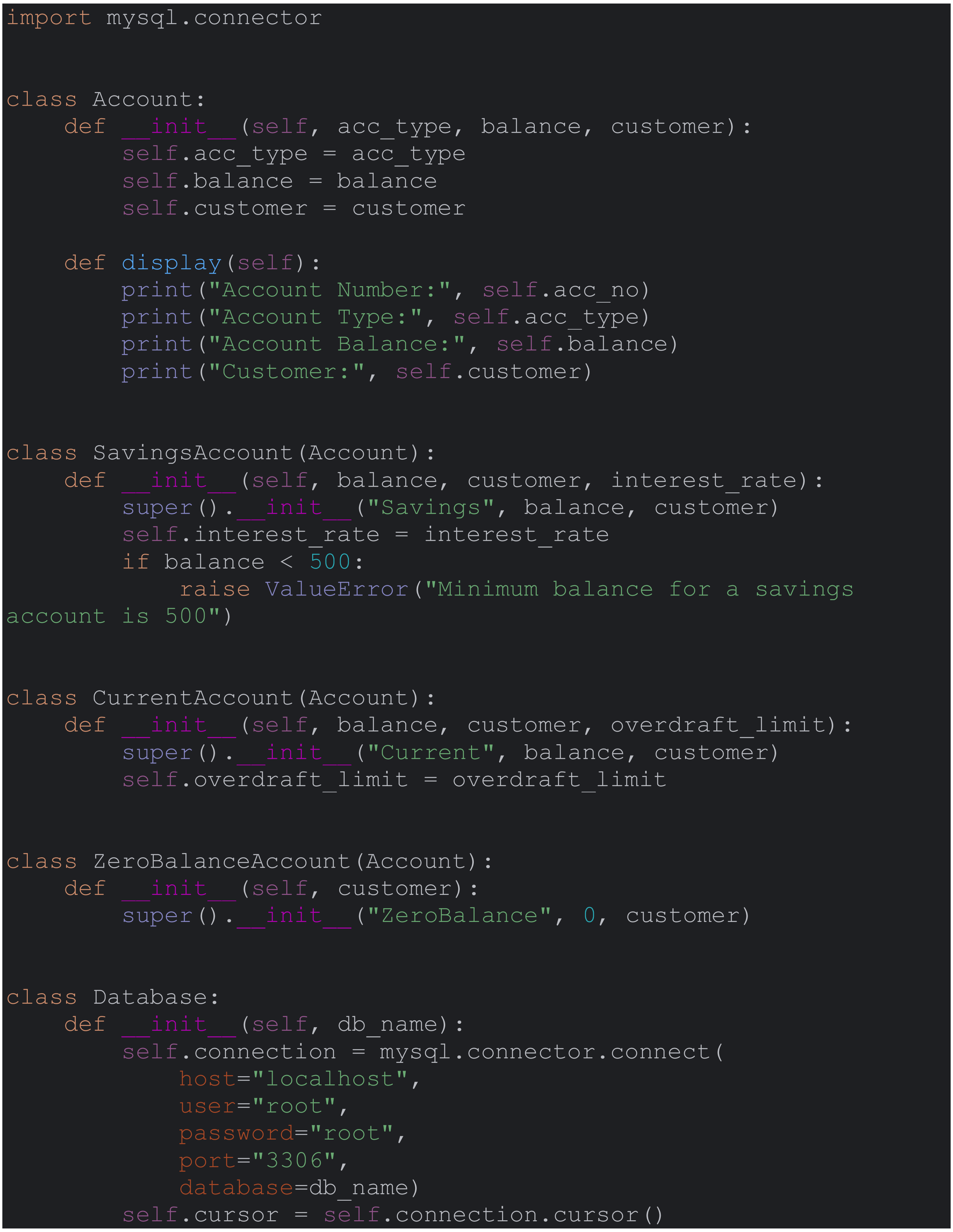
```
ID: 2
Account: 2
Description: Deposit
Date and Time: 2024-05-04 19:18:27
Transaction Type: Deposit
Transaction Amount: 200.0

ID: 3
Account: 1
Description: Withdrawal
Date and Time: 2024-05-04 19:19:01
Transaction Type: Withdraw
Transaction Amount: 10000.0

ID: 4
Account: 2
Description: Deposit
Date and Time: 2024-05-04 19:19:01
Transaction Type: Deposit
Transaction Amount: 20000.0
```



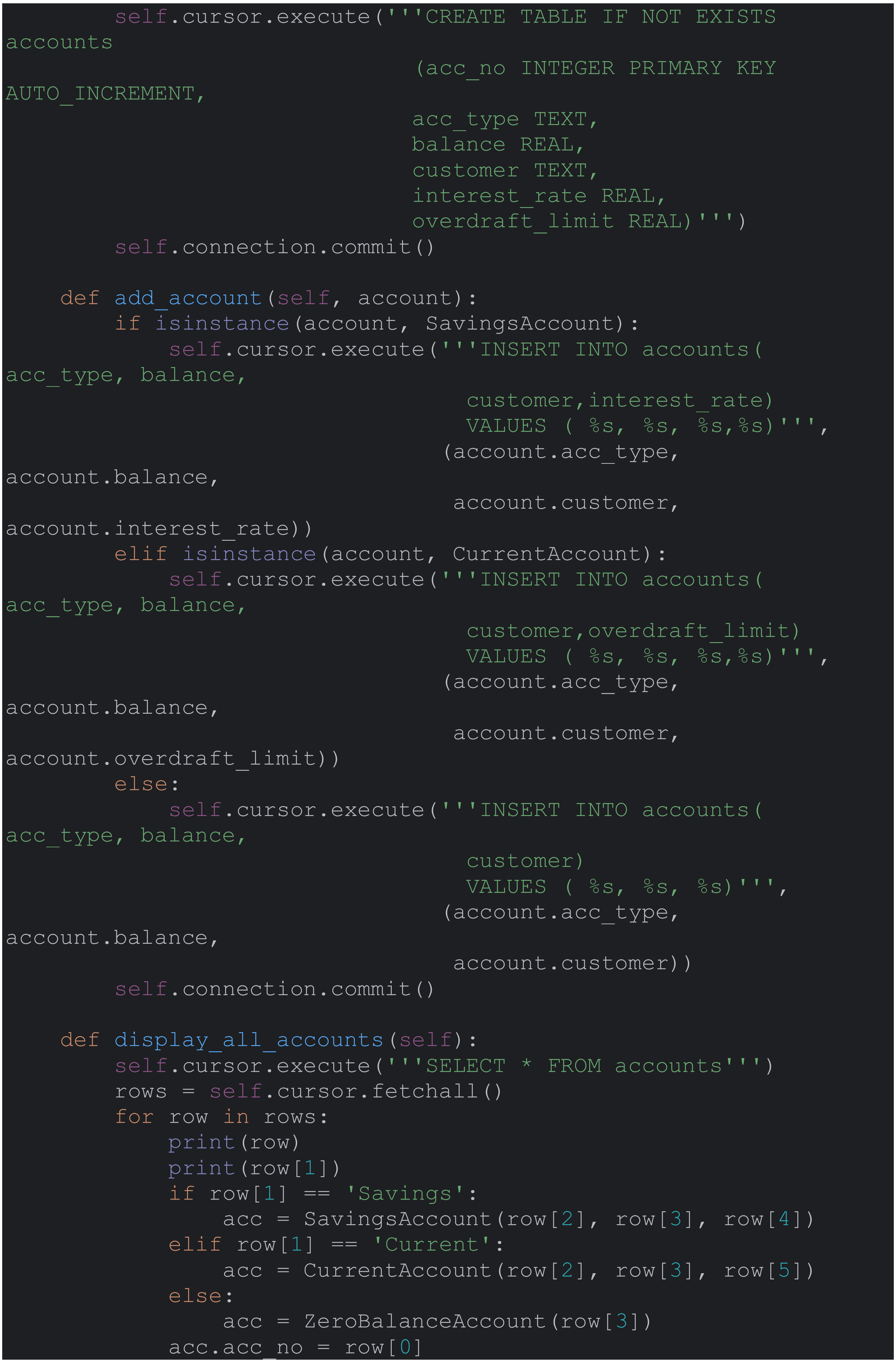
id	account	description	date_time	transaction_type	transaction_amount
1	1	Withdrawal	2024-05-04 19:18:27	Withdraw	100
2	2	Deposit	2024-05-04 19:18:27	Deposit	200
3	1	Withdrawal	2024-05-04 19:19:01	Withdraw	10000
4	2	Deposit	2024-05-04 19:19:01	Deposit	20000
NULL	NULL	NULL	NULL	NULL	NULL

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.

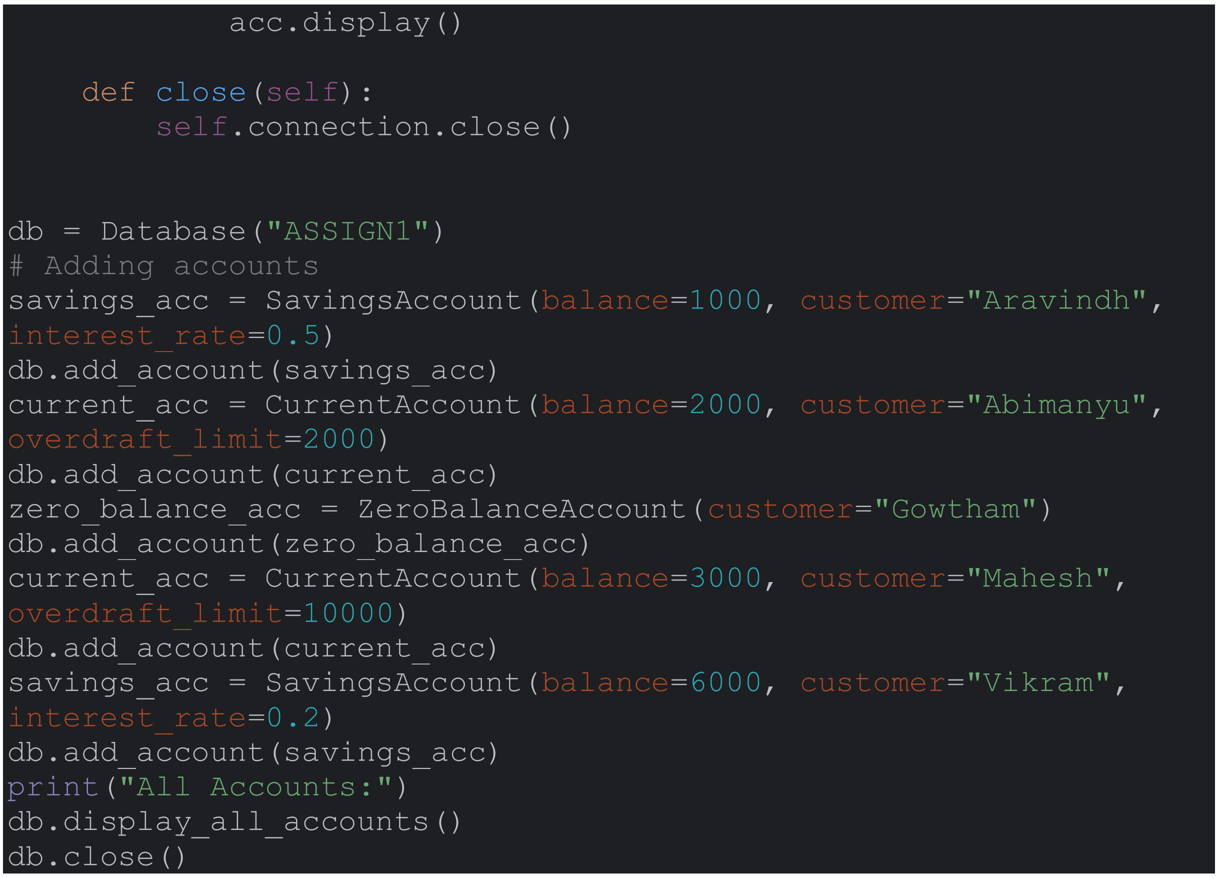
- **CurrentAccount:** A Current account that includes an additional attribute for overdraftLimit(credit limit).
- **ZeroBalanceAccount**: ZeroBalanceAccount can be created with Zero balance.



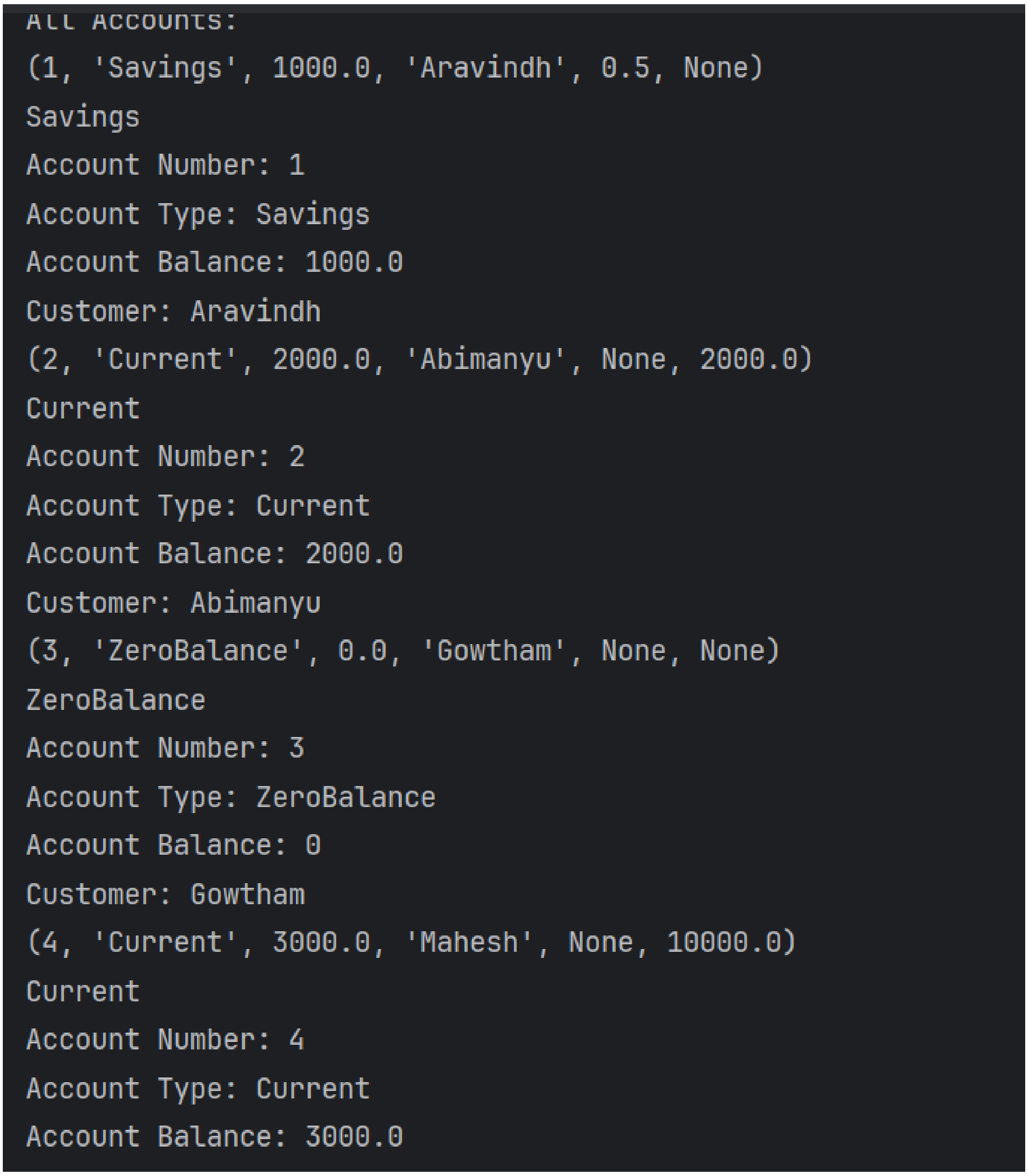
```
import mysql.connector


class Account:
    def __init__(self, acc_type, balance, customer):
        self.acc_type = acc_type
        self.balance = balance
        self.customer = customer

    def display(self):
        print("Account Number:", self.acc_no)
        print("Account Type:", self.acc_type)
        print("Account Balance:", self.balance)
        print("Customer:", self.customer)


class SavingsAccount(Account):
    def __init__(self, balance, customer, interest_rate):
        super().__init__("Savings", balance, customer)
        self.interest_rate = interest_rate
        if balance < 500:
            raise ValueError("Minimum balance for a savings
account is 500")


class CurrentAccount(Account):
    def __init__(self, balance, customer, overdraft_limit):
        super().__init__("Current", balance, customer)
        self.overdraft_limit = overdraft_limit


class ZeroBalanceAccount(Account):
    def __init__(self, customer):
        super().__init__("ZeroBalance", 0, customer)


class Database:
    def __init__(self, db_name):
        self.connection = mysql.connector.connect(
            host="localhost",
            user="root",
            password="root",
            port="3306",
            database=db_name)
        self.cursor = self.connection.cursor()
```



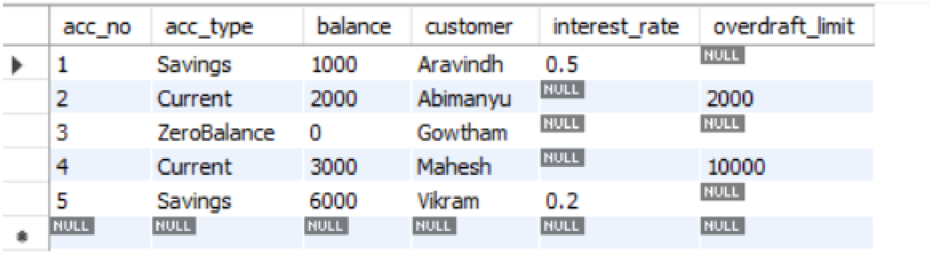
```
        self.cursor.execute('''CREATE TABLE IF NOT EXISTS
accounts
                              (acc_no INTEGER PRIMARY KEY
AUTO_INCREMENT,
                              acc_type TEXT,
                              balance REAL,
                              customer TEXT,
                              interest_rate REAL,
                              overdraft_limit REAL)''')
        self.connection.commit()

    def add_account(self, account):
        if isinstance(account, SavingsAccount):
            self.cursor.execute('''INSERT INTO accounts(
acc_type, balance,
                                  customer,interest_rate)
                                  VALUES ( %s, %s, %s,%s)''',
                                (account.acc_type,
account.balance,
                                 account.customer,
account.interest_rate))
        elif isinstance(account, CurrentAccount):
            self.cursor.execute('''INSERT INTO accounts(
acc_type, balance,
                                  customer,overdraft_limit)
                                  VALUES ( %s, %s, %s,%s)''',
                                (account.acc_type,
account.balance,
                                 account.customer,
account.overdraft_limit))
        else:
            self.cursor.execute('''INSERT INTO accounts(
acc_type, balance,
                                  customer)
                                  VALUES ( %s, %s, %s)''',
                                (account.acc_type,
account.balance,
                                 account.customer))
        self.connection.commit()

    def display_all_accounts(self):
        self.cursor.execute('''SELECT * FROM accounts''')
        rows = self.cursor.fetchall()
        for row in rows:
            print(row)
            print(row[1])
            if row[1] == 'Savings':
                acc = SavingsAccount(row[2], row[3], row[4])
            elif row[1] == 'Current':
                acc = CurrentAccount(row[2], row[3], row[5])
            else:
                acc = ZeroBalanceAccount(row[3])
            acc.acc_no = row[0]
```



```
            acc.display()

    def close(self):
        self.connection.close()


db = Database("ASSIGN1")
# Adding accounts
savings_acc = SavingsAccount(balance=1000, customer="Aravindh", interest_rate=0.5)
db.add_account(savings_acc)
current_acc = CurrentAccount(balance=2000, customer="Abimanyu", overdraft_limit=2000)
db.add_account(current_acc)
zero_balance_acc = ZeroBalanceAccount(customer="Gowtham")
db.add_account(zero_balance_acc)
current_acc = CurrentAccount(balance=3000, customer="Mahesh", overdraft_limit=10000)
db.add_account(current_acc)
savings_acc = SavingsAccount(balance=6000, customer="Vikram", interest_rate=0.2)
db.add_account(savings_acc)
print("All Accounts:")
db.display_all_accounts()
db.close()
```



```
All Accounts:
(1, 'Savings', 1000.0, 'Aravindh', 0.5, None)
Savings
Account Number: 1
Account Type: Savings
Account Balance: 1000.0
Customer: Aravindh
(2, 'Current', 2000.0, 'Abimanyu', None, 2000.0)
Current
Account Number: 2
Account Type: Current
Account Balance: 2000.0
Customer: Abimanyu
(3, 'ZeroBalance', 0.0, 'Gowtham', None, None)
ZeroBalance
Account Number: 3
Account Type: ZeroBalance
Account Balance: 0
Customer: Gowtham
(4, 'Current', 3000.0, 'Mahesh', None, 10000.0)
Current
Account Number: 4
Account Type: Current
Account Balance: 3000.0
```



acc_no	acc_type	balance	customer	interest_rate	overdraft_limit
1	Savings	1000	Aravindh	0.5	NULL
2	Current	2000	Abimanyu	NULL	2000
3	ZeroBalance	0	Gowtham	NULL	NULL
4	Current	3000	Mahesh	NULL	10000
5	Savings	6000	Vikram	0.2	NULL
NULL	NULL	NULL	NULL	NULL	NULL

- 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.
- **getTransations(account_number: long, FromDate:Date, ToDate: Date):** Should return the list of transaction between two dates.

- **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.

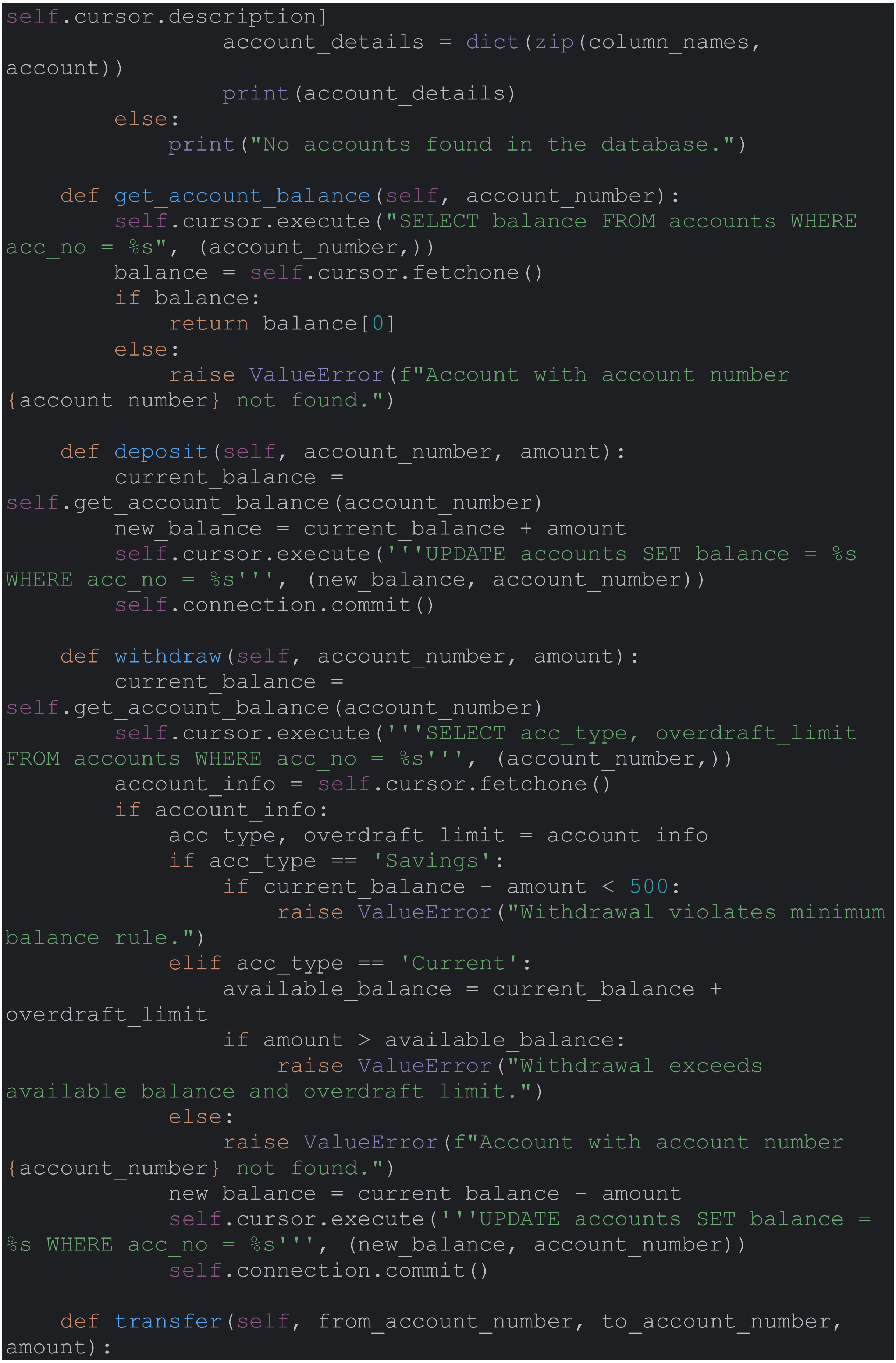
- **calculateInterest():** the calculate_interest() method to calculate interest based on the balance and interest rate.

  - Attributes ○ accountList: List of **Accounts** to store any account objects.
  - ○ transactionList: List of **Transaction** to store transaction objects.
  - ○ branchName and branchAddress as String objects
  -
- **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<Account> 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. ○ A savings account should maintain a minimum balance and checking if the withdrawal violates the minimum balance rule.

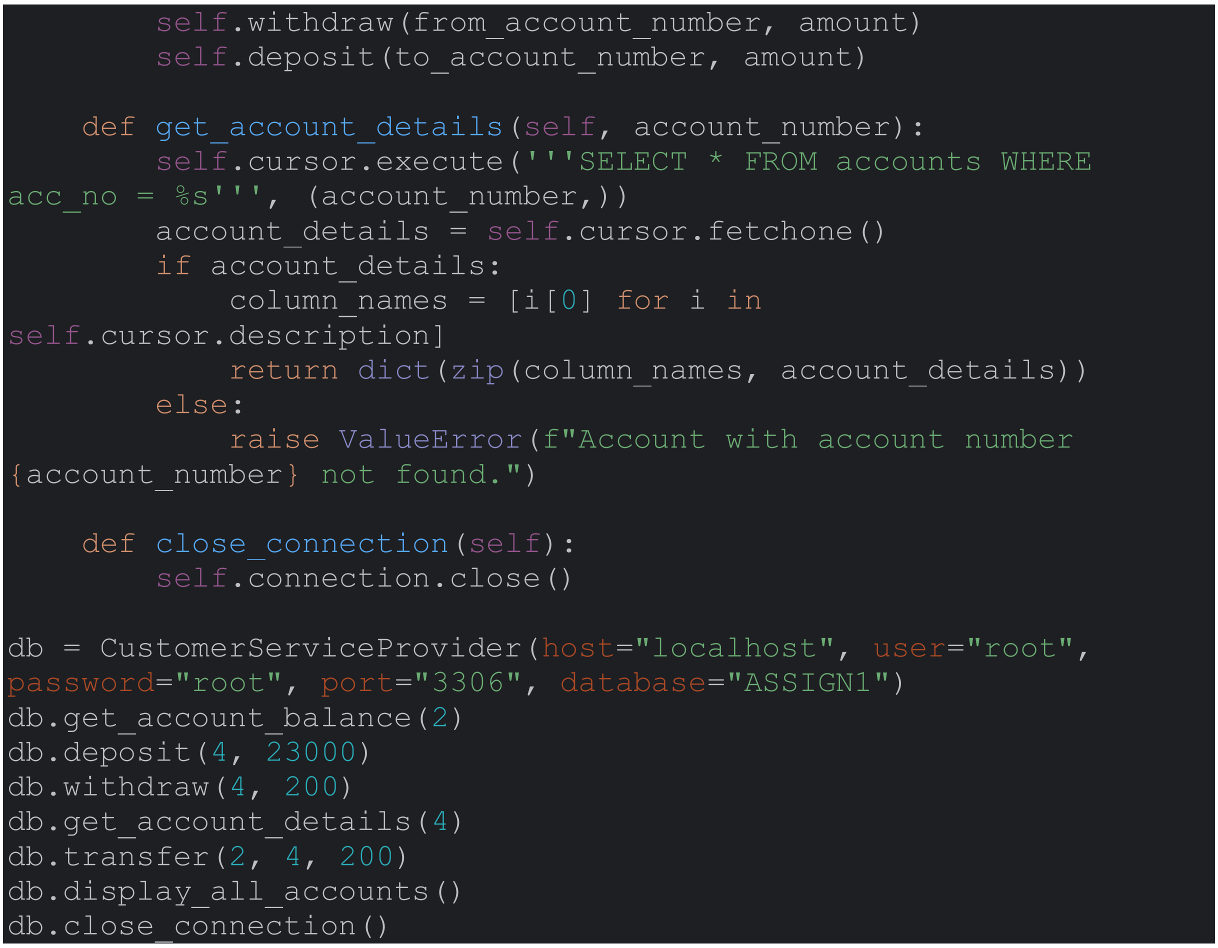
- ○ Current account customers are allowed withdraw overdraftLimit and available account balance. withdraw limit can exceed the available balance and should not exceed the overdraft limit.
-

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

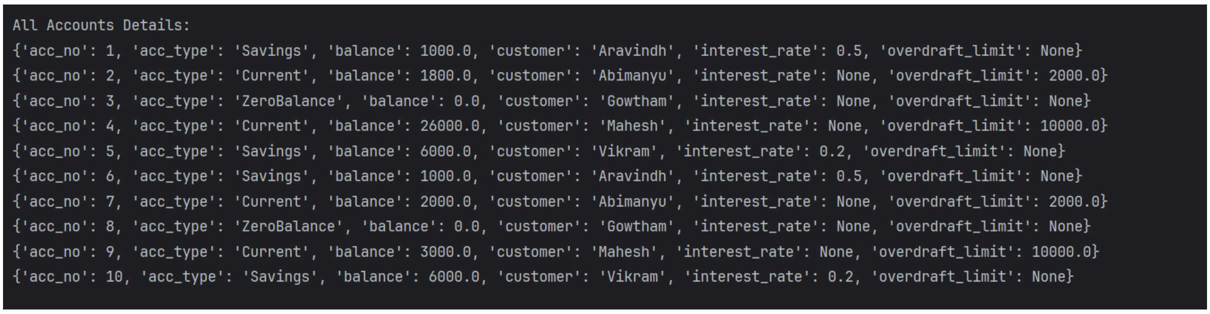
```
import mysql.connector
from abc import ABC, abstractmethod

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

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

    @abstractmethod
    def get_account_details(self, account_number):
        pass

class CustomerServiceProvider(ICustomerServiceProvider):
    def __init__(self, host, user, password, port, database):
        self.connection = mysql.connector.connect(
            host=host,
            user=user,
            password=password,
            port=port,
            database=database
        )
        self.cursor = self.connection.cursor()

    def display_all_accounts(self):
        self.cursor.execute('''SELECT * FROM accounts''')
        all_accounts = self.cursor.fetchall()
        if all_accounts:
            print("All Accounts Details:")
            for account in all_accounts:
                column_names = [i[0] for i in
```



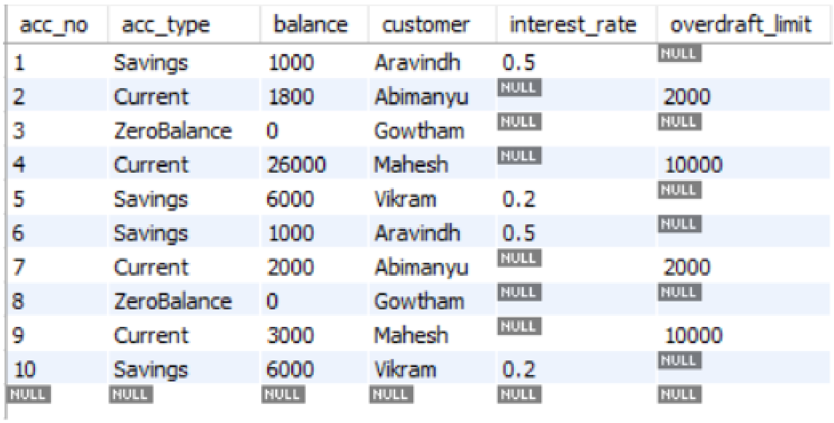
```
self.cursor.description]
                account_details = dict(zip(column_names,
account))
                print(account_details)
        else:
            print("No accounts found in the database.")

    def get_account_balance(self, account_number):
        self.cursor.execute("SELECT balance FROM accounts WHERE
acc_no = %s", (account_number,))
        balance = self.cursor.fetchone()
        if balance:
            return balance[0]
        else:
            raise ValueError(f"Account with account number
{account_number} not found.")

    def deposit(self, account_number, amount):
        current_balance =
self.get_account_balance(account_number)
        new_balance = current_balance + amount
        self.cursor.execute('''UPDATE accounts SET balance = %s
WHERE acc_no = %s''', (new_balance, account_number))
        self.connection.commit()

    def withdraw(self, account_number, amount):
        current_balance =
self.get_account_balance(account_number)
        self.cursor.execute('''SELECT acc_type, overdraft_limit
FROM accounts WHERE acc_no = %s''', (account_number,))
        account_info = self.cursor.fetchone()
        if account_info:
            acc_type, overdraft_limit = account_info
            if acc_type == 'Savings':
                if current_balance - amount < 500:
                    raise ValueError("Withdrawal violates minimum
balance rule.")
            elif acc_type == 'Current':
                available_balance = current_balance +
overdraft_limit
                if amount > available_balance:
                    raise ValueError("Withdrawal exceeds
available balance and overdraft limit.")
            else:
                raise ValueError(f"Account with account number
{account_number} not found.")
            new_balance = current_balance - amount
            self.cursor.execute('''UPDATE accounts SET balance =
%s WHERE acc_no = %s''', (new_balance, account_number))
            self.connection.commit()

    def transfer(self, from_account_number, to_account_number,
amount):
```



```
        self.withdraw(from_account_number, amount)
        self.deposit(to_account_number, amount)

    def get_account_details(self, account_number):
        self.cursor.execute('''SELECT * FROM accounts WHERE 
acc_no = %s''', (account_number,))
        account_details = self.cursor.fetchone()
        if account_details:
            column_names = [i[0] for i in 
self.cursor.description]
            return dict(zip(column_names, account_details))
        else:
            raise ValueError(f"Account with account number 
{account_number} not found.")

    def close_connection(self):
        self.connection.close()

db = CustomerServiceProvider(host="localhost", user="root", 
password="root", port="3306", database="ASSIGN1")
db.get_account_balance(2)
db.deposit(4, 23000)
db.withdraw(4, 200)
db.get_account_details(4)
db.transfer(2, 4, 200)
db.display_all_accounts()
db.close_connection()
```



```
All Accounts Details:
{'acc_no': 1, 'acc_type': 'Savings', 'balance': 1000.0, 'customer': 'Aravindh', 'interest_rate': 0.5, 'overdraft_limit': None}
{'acc_no': 2, 'acc_type': 'Current', 'balance': 1800.0, 'customer': 'Abimanyu', 'interest_rate': None, 'overdraft_limit': 2000.0}
{'acc_no': 3, 'acc_type': 'ZeroBalance', 'balance': 0.0, 'customer': 'Gowtham', 'interest_rate': None, 'overdraft_limit': None}
{'acc_no': 4, 'acc_type': 'Current', 'balance': 26000.0, 'customer': 'Mahesh', 'interest_rate': None, 'overdraft_limit': 10000.0}
{'acc_no': 5, 'acc_type': 'Savings', 'balance': 6000.0, 'customer': 'Vikram', 'interest_rate': 0.2, 'overdraft_limit': None}
{'acc_no': 6, 'acc_type': 'Savings', 'balance': 1000.0, 'customer': 'Aravindh', 'interest_rate': 0.5, 'overdraft_limit': None}
{'acc_no': 7, 'acc_type': 'Current', 'balance': 2000.0, 'customer': 'Abimanyu', 'interest_rate': None, 'overdraft_limit': 2000.0}
{'acc_no': 8, 'acc_type': 'ZeroBalance', 'balance': 0.0, 'customer': 'Gowtham', 'interest_rate': None, 'overdraft_limit': None}
{'acc_no': 9, 'acc_type': 'Current', 'balance': 3000.0, 'customer': 'Mahesh', 'interest_rate': None, 'overdraft_limit': 10000.0}
{'acc_no': 10, 'acc_type': 'Savings', 'balance': 6000.0, 'customer': 'Vikram', 'interest_rate': 0.2, 'overdraft_limit': None}
```



acc_no	acc_type	balance	customer	interest_rate	overdraft_limit
1	Savings	1000	Aravindh	0.5	NULL
2	Current	1800	Abimanyu	NULL	2000
3	ZeroBalance	0	Gowtham	NULL	NULL
4	Current	26000	Mahesh	NULL	10000
5	Savings	6000	Vikram	0.2	NULL
6	Savings	1000	Aravindh	0.5	NULL
7	Current	2000	Abimanyu	NULL	2000
8	ZeroBalance	0	Gowtham	NULL	NULL
9	Current	3000	Mahesh	NULL	10000
10	Savings	6000	Vikram	0.2	NULL
NULL	NULL	NULL	NULL	NULL	NULL

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

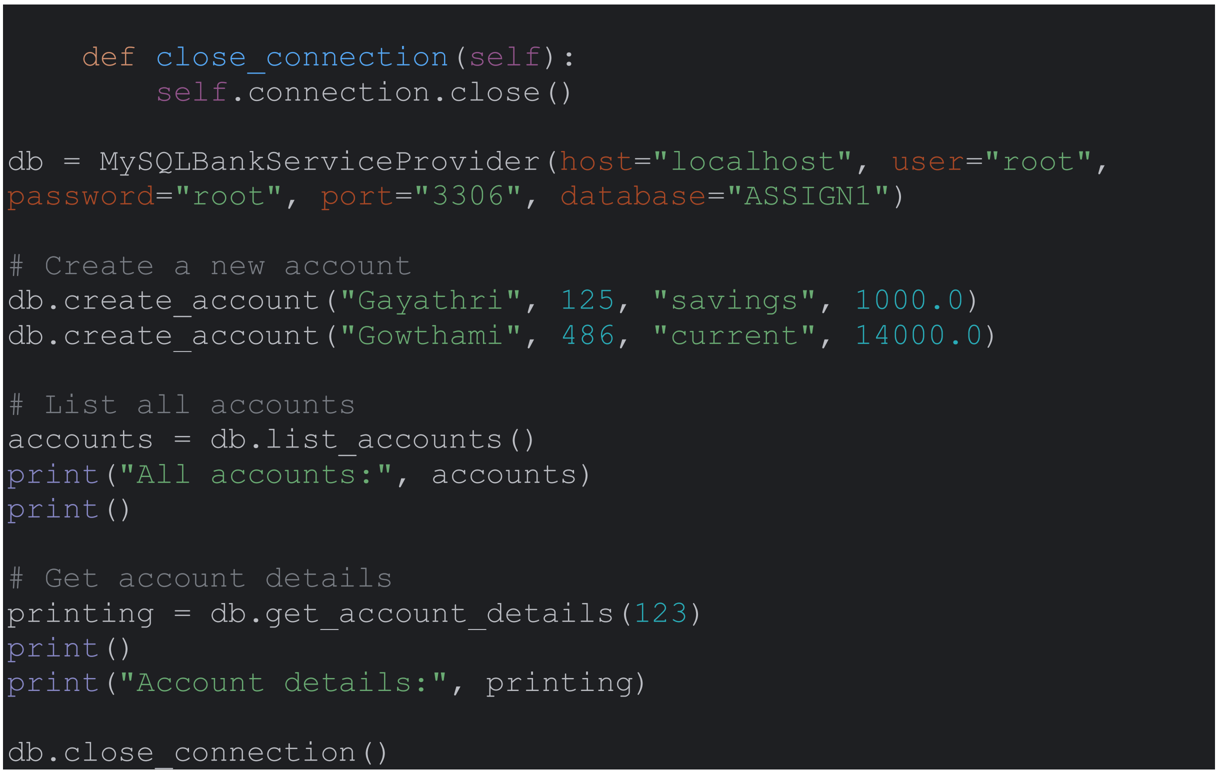
create_account(Customer customer, long accNo, String accType, float balance): Create a new bank account for the given customer with the initial balance.

listAccounts(): Array of BankAccount: List all accounts in the bank.(List[Account] accountsList)

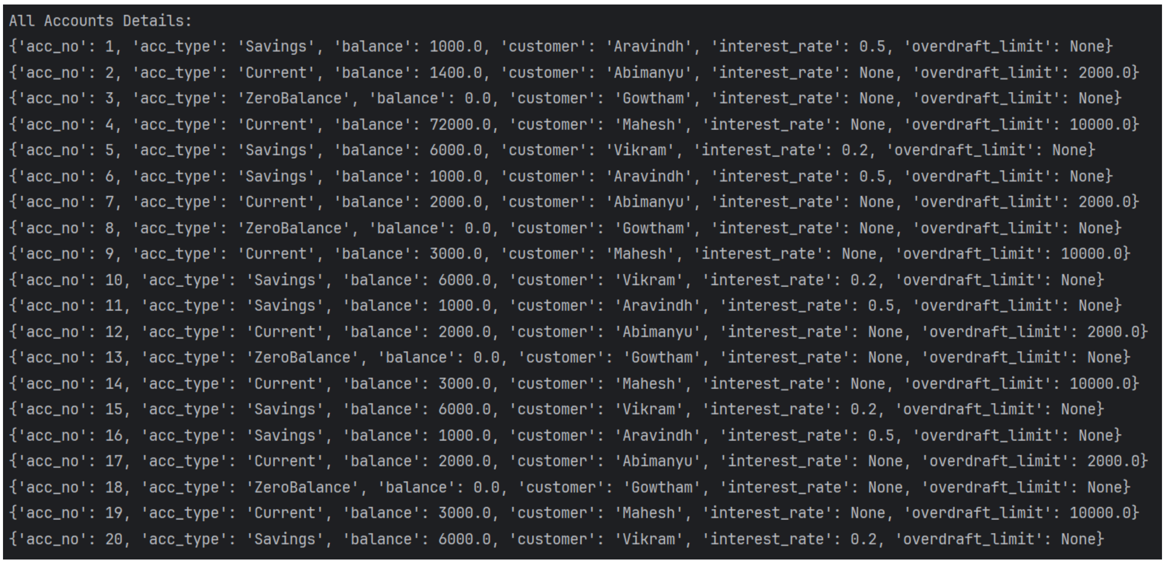
getAccountDetails(account_number: long): Should return the account and customer details.

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

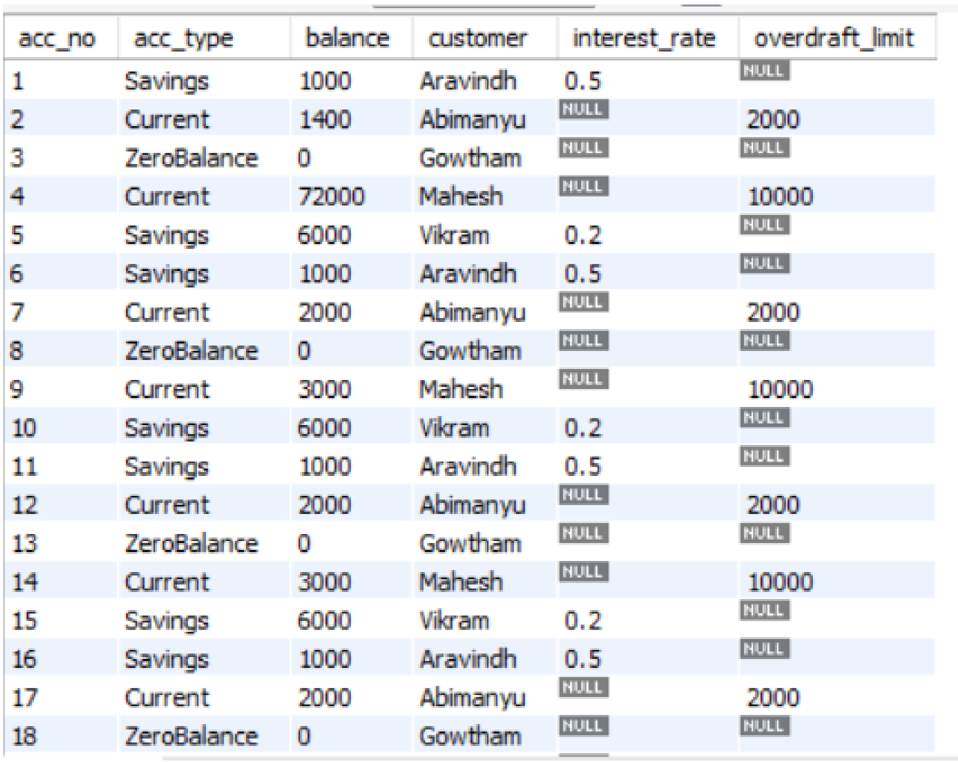
```
import mysql.connector
from abc import ABC, abstractmethod

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

    @abstractmethod
    def list_accounts(self):
        pass

    @abstractmethod
    def get_account_details(self, account_number):
        pass

class MySQLBankServiceProvider(IBankServiceProvider):
    def __init__(self, host, user, password, port, database):
        self.connection = mysql.connector.connect(
            host=host,
            user=user,
            password=password,
            port=port,
            database=database
        )
        self.cursor = self.connection.cursor()

    def create_account(self, customer, acc_no, acc_type,
balance):
        query = "INSERT INTO accounts (customer,acc_no, acc_type,
balance) VALUES (%s, %s, %s,%s)"
        values = (customer, acc_no, acc_type, balance)
        self.cursor.execute(query, values)
        self.connection.commit()

    def list_accounts(self):
        self.cursor.execute("SELECT * FROM accounts")
        accounts = self.cursor.fetchall()
        return accounts

    def get_account_details(self, account_number):
        query = "SELECT * FROM accounts WHERE acc_no = %s"
        self.cursor.execute(query, (account_number,))
        account_details = self.cursor.fetchone()
        return account_details
```



```
    def close_connection(self):
        self.connection.close()

db = MySQLBankServiceProvider(host="localhost", user="root",
password="root", port="3306", database="ASSIGN1")

# Create a new account
db.create_account("Gayathri", 125, "savings", 1000.0)
db.create_account("Gowthami", 486, "current", 14000.0)

# List all accounts
accounts = db.list_accounts()
print("All accounts:", accounts)
print()

# Get account details
printing = db.get_account_details(123)
print()
print("Account details:", printing)

db.close_connection()
```

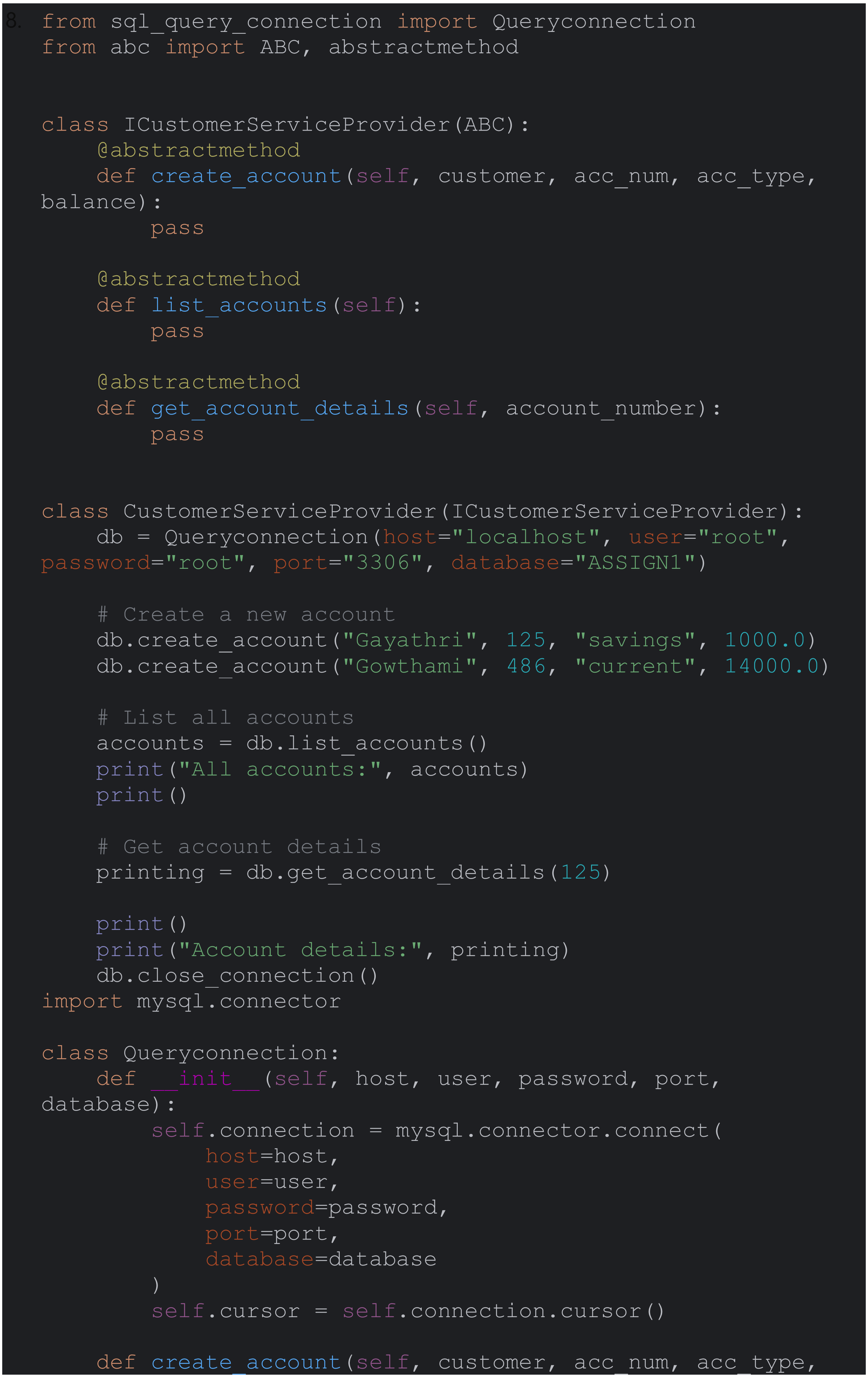


```
All Accounts Details:
{'acc_no': 1, 'acc_type': 'Savings', 'balance': 1000.0, 'customer': 'Aravindh', 'interest_rate': 0.5, 'overdraft_limit': None}
{'acc_no': 2, 'acc_type': 'Current', 'balance': 1400.0, 'customer': 'Abimanyu', 'interest_rate': None, 'overdraft_limit': 2000.0}
{'acc_no': 3, 'acc_type': 'ZeroBalance', 'balance': 0.0, 'customer': 'Gowtham', 'interest_rate': None, 'overdraft_limit': None}
{'acc_no': 4, 'acc_type': 'Current', 'balance': 72000.0, 'customer': 'Mahesh', 'interest_rate': None, 'overdraft_limit': 10000.0}
{'acc_no': 5, 'acc_type': 'Savings', 'balance': 6000.0, 'customer': 'Vikram', 'interest_rate': 0.2, 'overdraft_limit': None}
{'acc_no': 6, 'acc_type': 'Savings', 'balance': 1000.0, 'customer': 'Aravindh', 'interest_rate': 0.5, 'overdraft_limit': None}
{'acc_no': 7, 'acc_type': 'Current', 'balance': 2000.0, 'customer': 'Abimanyu', 'interest_rate': None, 'overdraft_limit': 2000.0}
{'acc_no': 8, 'acc_type': 'ZeroBalance', 'balance': 0.0, 'customer': 'Gowtham', 'interest_rate': None, 'overdraft_limit': None}
{'acc_no': 9, 'acc_type': 'Current', 'balance': 3000.0, 'customer': 'Mahesh', 'interest_rate': None, 'overdraft_limit': 10000.0}
{'acc_no': 10, 'acc_type': 'Savings', 'balance': 6000.0, 'customer': 'Vikram', 'interest_rate': 0.2, 'overdraft_limit': None}
{'acc_no': 11, 'acc_type': 'Savings', 'balance': 1000.0, 'customer': 'Aravindh', 'interest_rate': 0.5, 'overdraft_limit': None}
{'acc_no': 12, 'acc_type': 'Current', 'balance': 2000.0, 'customer': 'Abimanyu', 'interest_rate': None, 'overdraft_limit': 2000.0}
{'acc_no': 13, 'acc_type': 'ZeroBalance', 'balance': 0.0, 'customer': 'Gowtham', 'interest_rate': None, 'overdraft_limit': None}
{'acc_no': 14, 'acc_type': 'Current', 'balance': 3000.0, 'customer': 'Mahesh', 'interest_rate': None, 'overdraft_limit': 10000.0}
{'acc_no': 15, 'acc_type': 'Savings', 'balance': 6000.0, 'customer': 'Vikram', 'interest_rate': 0.2, 'overdraft_limit': None}
{'acc_no': 16, 'acc_type': 'Savings', 'balance': 1000.0, 'customer': 'Aravindh', 'interest_rate': 0.5, 'overdraft_limit': None}
{'acc_no': 17, 'acc_type': 'Current', 'balance': 2000.0, 'customer': 'Abimanyu', 'interest_rate': None, 'overdraft_limit': 2000.0}
{'acc_no': 18, 'acc_type': 'ZeroBalance', 'balance': 0.0, 'customer': 'Gowtham', 'interest_rate': None, 'overdraft_limit': None}
{'acc_no': 19, 'acc_type': 'Current', 'balance': 3000.0, 'customer': 'Mahesh', 'interest_rate': None, 'overdraft_limit': 10000.0}
{'acc_no': 20, 'acc_type': 'Savings', 'balance': 6000.0, 'customer': 'Vikram', 'interest_rate': 0.2, 'overdraft_limit': None}
```



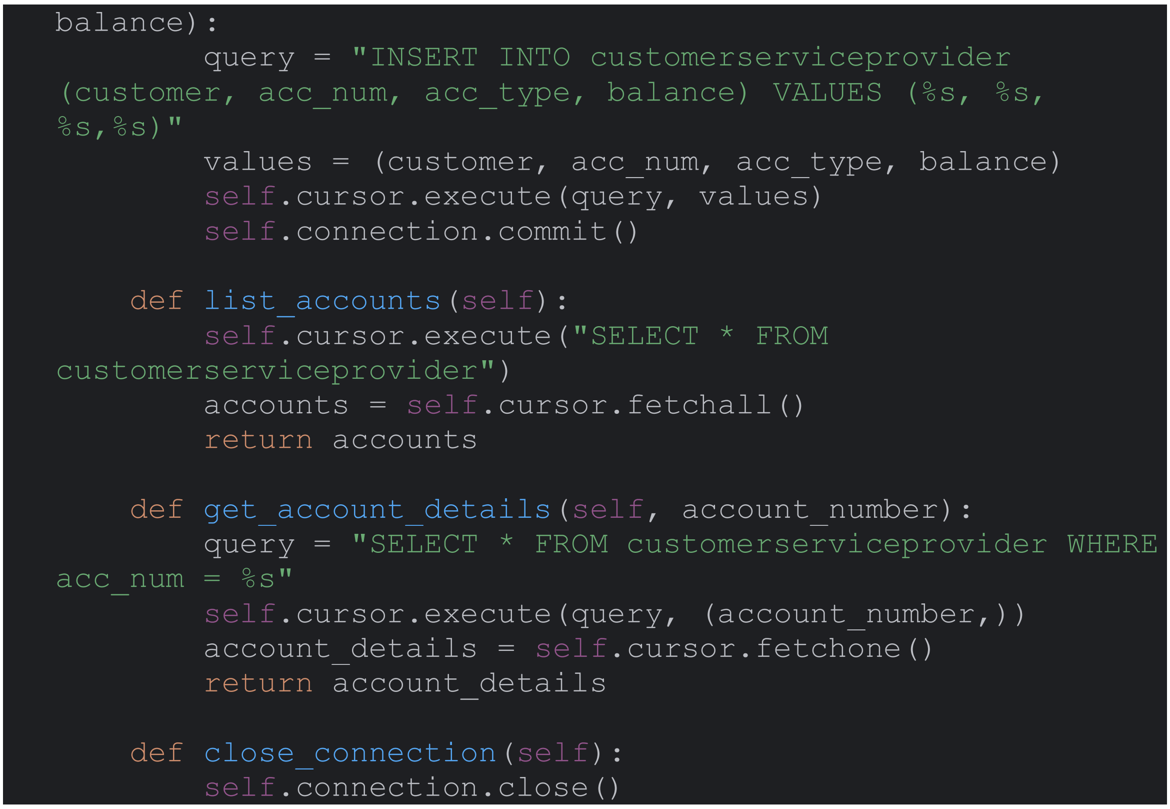
acc_no	acc_type	balance	customer	interest_rate	overdraft_limit
1	Savings	1000	Aravindh	0.5	NULL
2	Current	1400	Abimanyu	NULL	2000
3	ZeroBalance	0	Gowtham	NULL	NULL
4	Current	72000	Mahesh	NULL	10000
5	Savings	6000	Vikram	0.2	NULL
6	Savings	1000	Aravindh	0.5	NULL
7	Current	2000	Abimanyu	NULL	2000
8	ZeroBalance	0	Gowtham	NULL	NULL
9	Current	3000	Mahesh	NULL	10000
10	Savings	6000	Vikram	0.2	NULL
11	Savings	1000	Aravindh	0.5	NULL
12	Current	2000	Abimanyu	NULL	2000
13	ZeroBalance	0	Gowtham	NULL	NULL
14	Current	3000	Mahesh	NULL	10000
15	Savings	6000	Vikram	0.2	NULL
16	Savings	1000	Aravindh	0.5	NULL
17	Current	2000	Abimanyu	NULL	2000
18	ZeroBalance	0	Gowtham	NULL	NULL

7. Create **CustomerServiceProviderImpl** class which implements I**CustomerServiceProvider** provide all implementation methods. These methods do not interact with database directly.

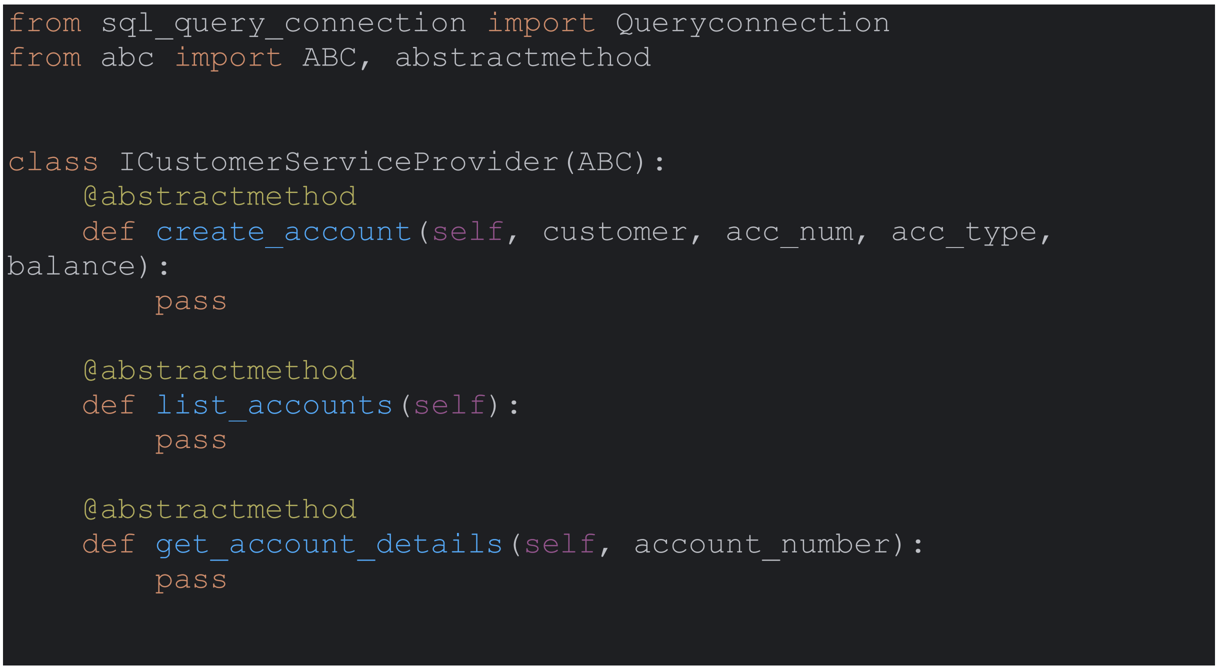


8.

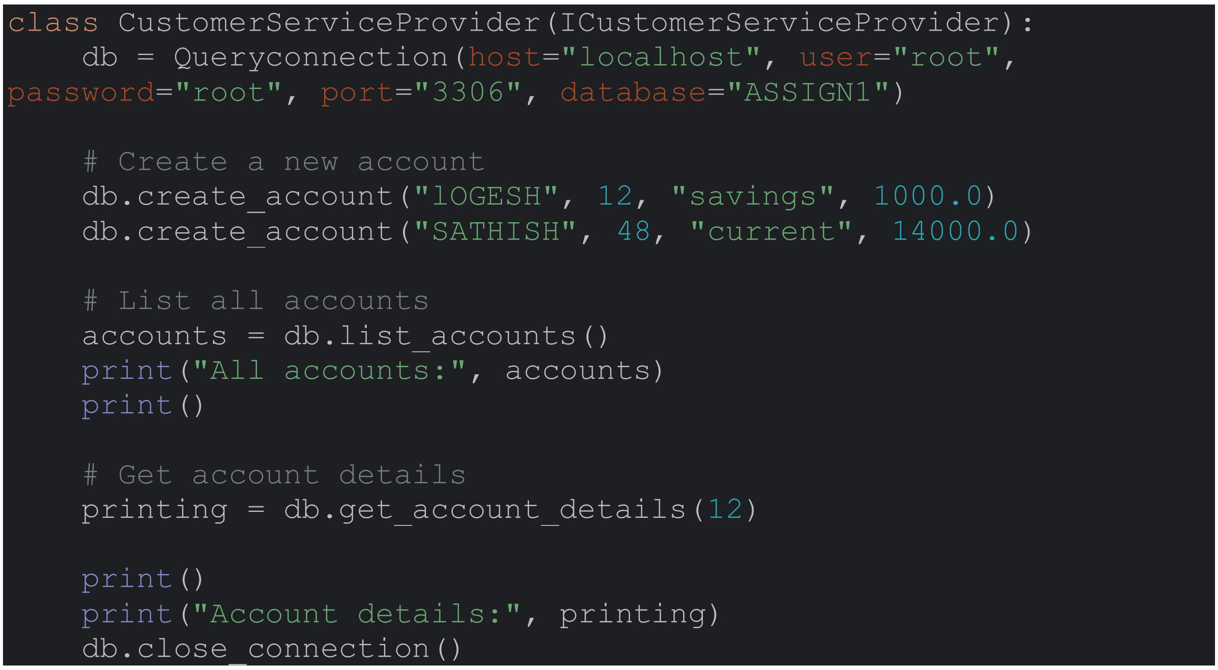
```
from sql_query_connection import Queryconnection
from abc import ABC, abstractmethod


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

    @abstractmethod
    def list_accounts(self):
        pass

    @abstractmethod
    def get_account_details(self, account_number):
        pass


class CustomerServiceProvider(ICustomerServiceProvider):
    db = Queryconnection(host="localhost", user="root",
password="root", port="3306", database="ASSIGN1")

    # Create a new account
    db.create_account("Gayathri", 125, "savings", 1000.0)
    db.create_account("Gowthami", 486, "current", 14000.0)

    # List all accounts
    accounts = db.list_accounts()
    print("All accounts:", accounts)
    print()

    # Get account details
    printing = db.get_account_details(125)

    print()
    print("Account details:", printing)
    db.close_connection()
import mysql.connector

class Queryconnection:
    def __init__(self, host, user, password, port,
database):
        self.connection = mysql.connector.connect(
            host=host,
            user=user,
            password=password,
            port=port,
            database=database
        )
        self.cursor = self.connection.cursor()

    def create_account(self, customer, acc_num, acc_type,
```



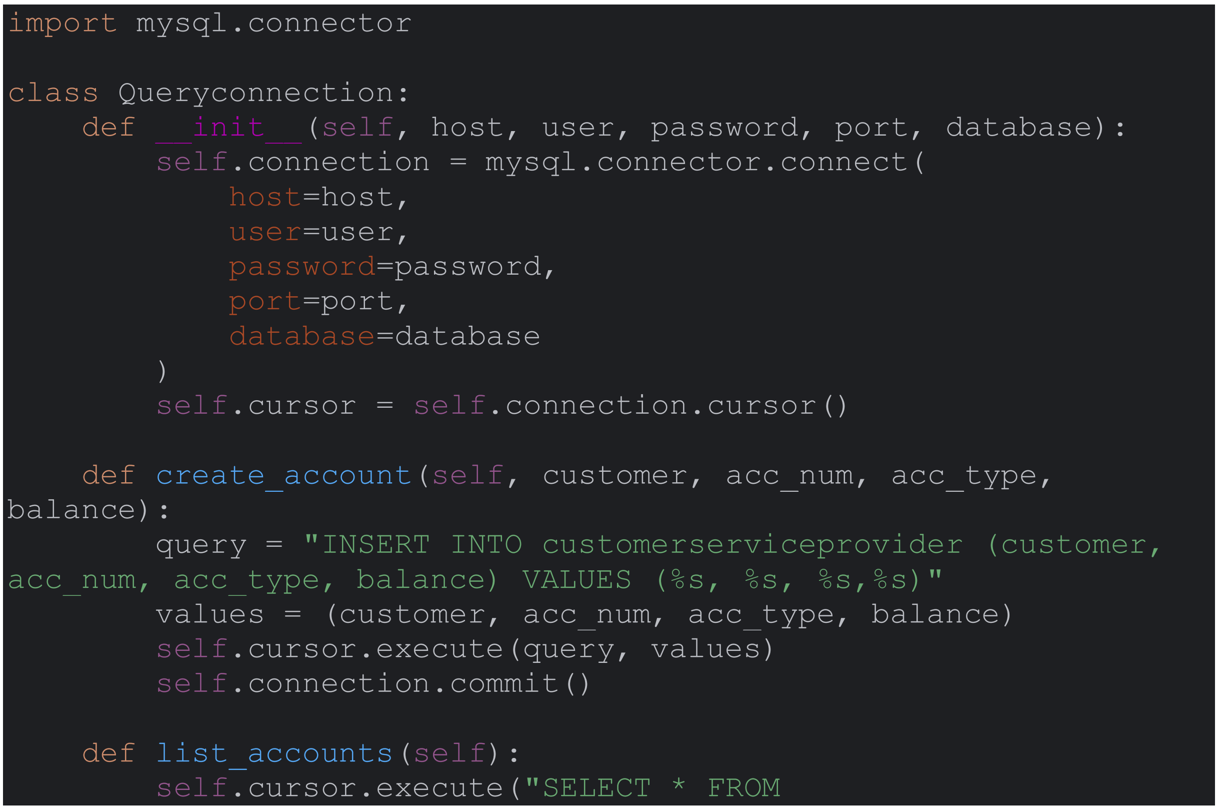
```
balance):
        query = "INSERT INTO customerserviceprovider
(customer, acc_num, acc_type, balance) VALUES (%s, %s,
%s,%s)"
        values = (customer, acc_num, acc_type, balance)
        self.cursor.execute(query, values)
        self.connection.commit()

    def list_accounts(self):
        self.cursor.execute("SELECT * FROM
customerserviceprovider")
        accounts = self.cursor.fetchall()
        return accounts

    def get_account_details(self, account_number):
        query = "SELECT * FROM customerserviceprovider WHERE
acc_num = %s"
        self.cursor.execute(query, (account_number,))
        account_details = self.cursor.fetchone()
        return account_details

    def close_connection(self):
        self.connection.close()
```



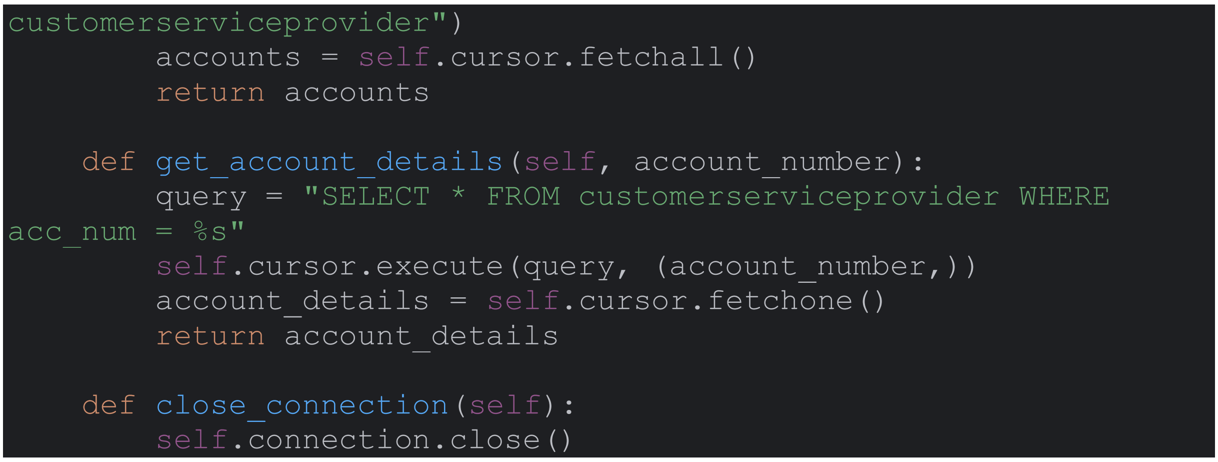
```
from sql_query_connection import Queryconnection
from abc import ABC, abstractmethod


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

    @abstractmethod
    def list_accounts(self):
        pass

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



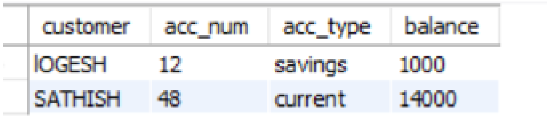
```
class CustomerServiceProvider(ICustomerServiceProvider):
    db = Queryconnection(host="localhost", user="root",
password="root", port="3306", database="ASSIGN1")

    # Create a new account
    db.create_account("lOGESH", 12, "savings", 1000.0)
    db.create_account("SATHISH", 48, "current", 14000.0)

    # List all accounts
    accounts = db.list_accounts()
    print("All accounts:", accounts)
    print()

    # Get account details
    printing = db.get_account_details(12)

    print()
    print("Account details:", printing)
    db.close_connection()
```



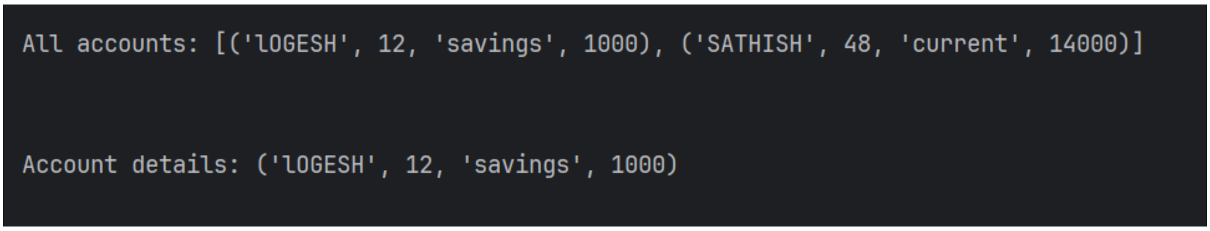
```
import mysql.connector

class Queryconnection:
    def __init__(self, host, user, password, port, database):
        self.connection = mysql.connector.connect(
            host=host,
            user=user,
            password=password,
            port=port,
            database=database
        )
        self.cursor = self.connection.cursor()

    def create_account(self, customer, acc_num, acc_type,
balance):
        query = "INSERT INTO customerserviceprovider (customer,
acc_num, acc_type, balance) VALUES (%s, %s, %s,%s)"
        values = (customer, acc_num, acc_type, balance)
        self.cursor.execute(query, values)
        self.connection.commit()

    def list_accounts(self):
        self.cursor.execute("SELECT * FROM
```



```
customerserviceprovider")
        accounts = self.cursor.fetchall()
        return accounts

    def get_account_details(self, account_number):
        query = "SELECT * FROM customerserviceprovider WHERE
acc_num = %s"
        self.cursor.execute(query, (account_number,))
        account_details = self.cursor.fetchone()
        return account_details

    def close_connection(self):
        self.connection.close()
```



customer	acc_num	acc_type	balance
lOGESH	12	savings	1000
SATHISH	48	current	14000



```
All accounts: [('lOGESH', 12, 'savings', 1000), ('SATHISH', 48, 'current', 14000)]


Account details: ('lOGESH', 12, 'savings', 1000)
```