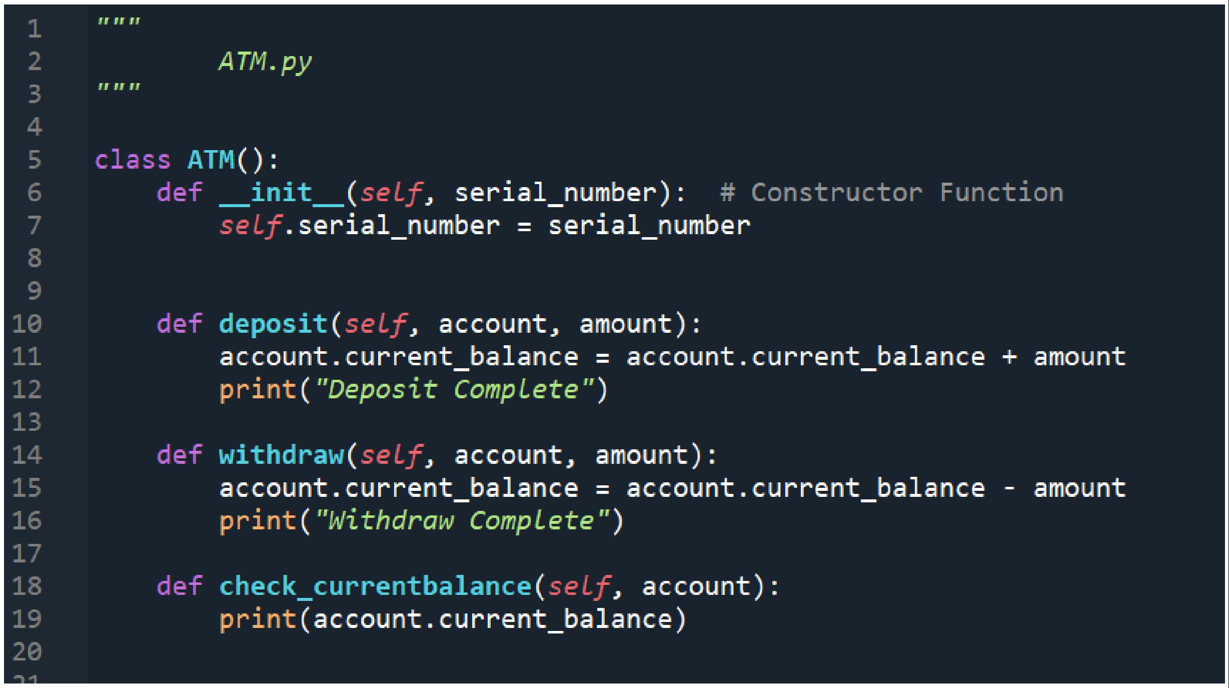
Introduction to Object-Oriented Programming - Activity 1	
San Juan, Edson Ray E.	09/14/2024
Course/Section: CPE21S4	Prof. Maria Rizette Sayo

6. Supplementary Activity:

Tasks

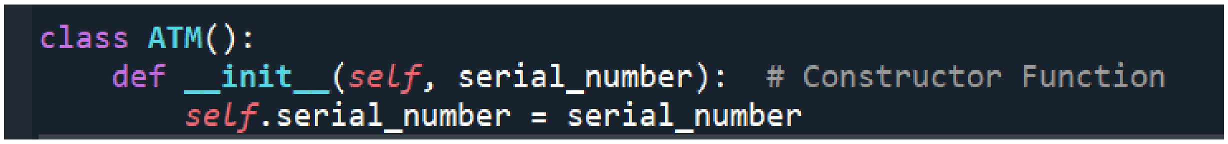
1. Modify the ATM.py program and add the constructor function.



```python
"""
        ATM.py
"""

class ATM():
    def __init__(self, serial_number):  # Constructor Function
        self.serial_number = serial_number


    def deposit(self, account, amount):
        account.current_balance = account.current_balance + amount
        print("Deposit Complete")

    def withdraw(self, account, amount):
        account.current_balance = account.current_balance - amount
        print("Withdraw Complete")

    def check_currentbalance(self, account):
        print(account.current_balance)

```

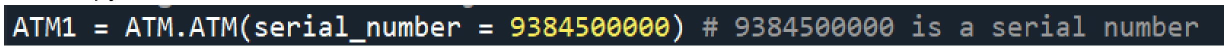
2. Modify the main.py program and initialize the ATM machine with any integer serial number combination and display the serial number at the end of the program.

Code:

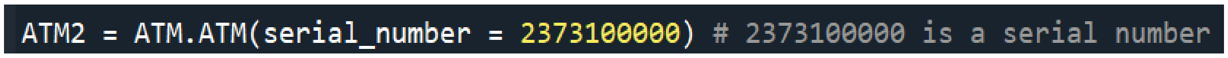


```python
class ATM():
    def __init__(self, serial_number):  # Constructor Function
        self.serial_number = serial_number
```

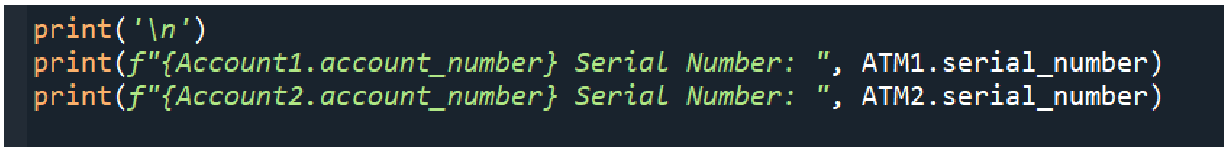
In main.py



```python
ATM1 = ATM.ATM(serial_number = 9384500000) # 9384500000 is a serial number
```

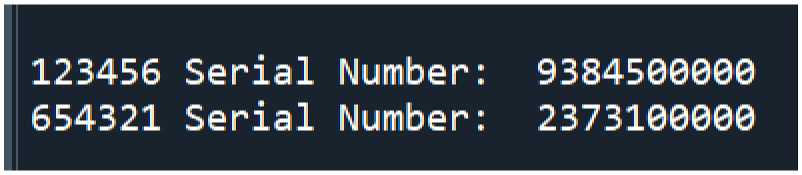


```python
ATM2 = ATM.ATM(serial_number = 2373100000) # 2373100000 is a serial number
```



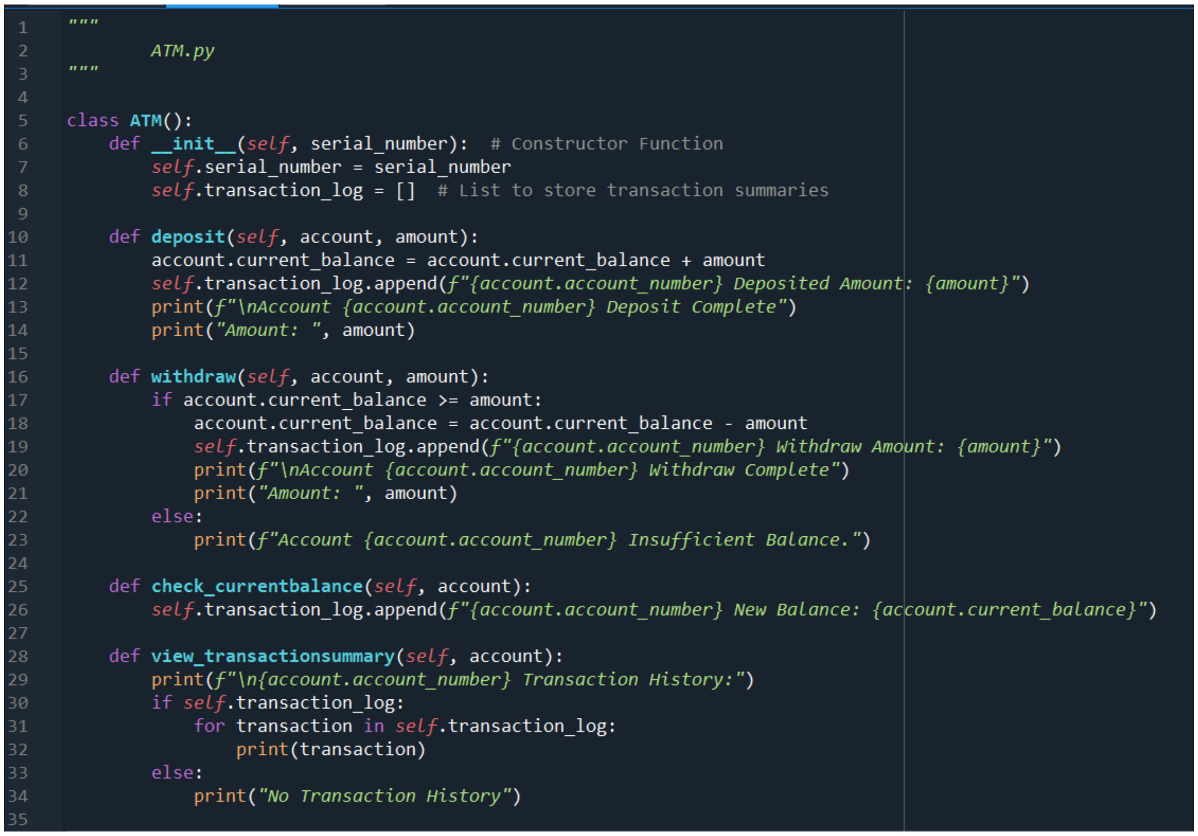
```python
print('\n')
print(f"{Account1.account_number} Serial Number: ", ATM1.serial_number)
print(f"{Account2.account_number} Serial Number: ", ATM2.serial_number)
```

Output:



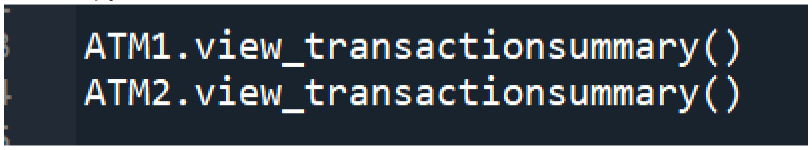
```
123456 Serial Number:  9384500000
654321 Serial Number:  2373100000
```

3. Modify the ATM.py program and add the view_transactionsummary() method. The method should display all the transaction made in the ATM object.



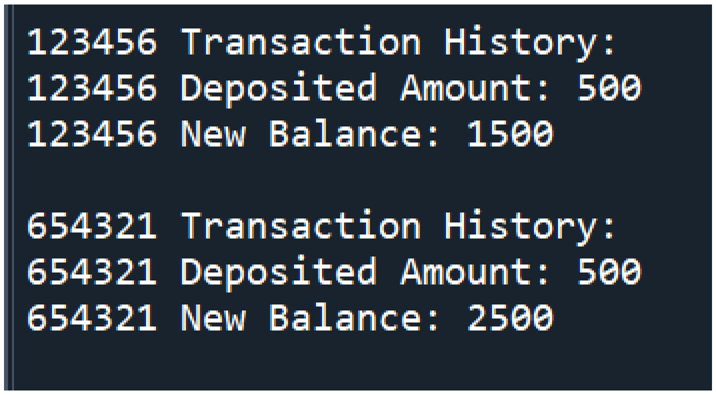
```python
"""
        ATM.py
"""

class ATM():
    def __init__(self, serial_number):  # Constructor Function
        self.serial_number = serial_number
        self.transaction_log = []  # List to store transaction summaries

    def deposit(self, account, amount):
        account.current_balance = account.current_balance + amount
        self.transaction_log.append(f"{account.account_number} Deposited Amount: {amount}")
        print(f"\nAccount {account.account_number} Deposit Complete")
        print("Amount: ", amount)

    def withdraw(self, account, amount):
        if account.current_balance >= amount:
            account.current_balance = account.current_balance - amount
            self.transaction_log.append(f"{account.account_number} Withdraw Amount: {amount}")
            print(f"\nAccount {account.account_number} Withdraw Complete")
            print("Amount: ", amount)
        else:
            print(f"Account {account.account_number} Insufficient Balance.")

    def check_currentbalance(self, account):
        self.transaction_log.append(f"{account.account_number} New Balance: {account.current_balance}")

    def view_transactionsummary(self, account):
        print(f"\n{account.account_number} Transaction History:")
        if self.transaction_log:
            for transaction in self.transaction_log:
                print(transaction)
        else:
            print("No Transaction History")
```

In main.py:



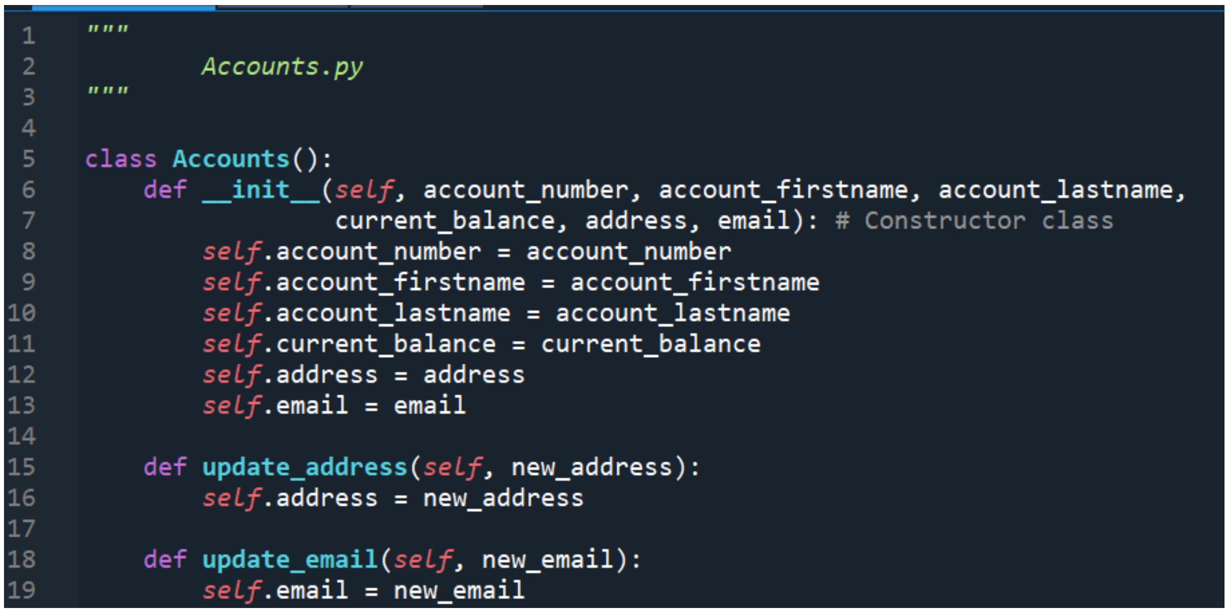
```python
ATM1.view_transactionsummary()
ATM2.view_transactionsummary()
```

Output:

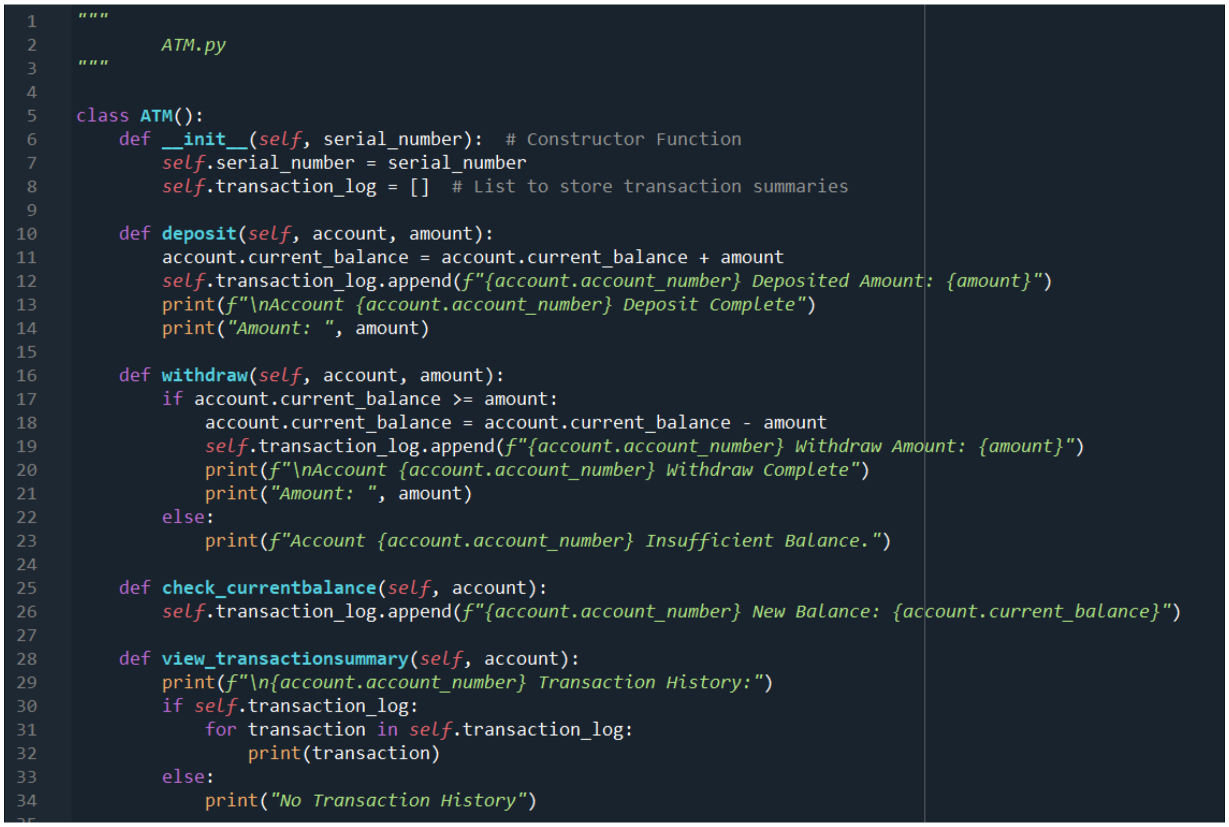


```
123456 Transaction History:
123456 Deposited Amount: 500
123456 New Balance: 1500

654321 Transaction History:
654321 Deposited Amount: 500
654321 New Balance: 2500
```

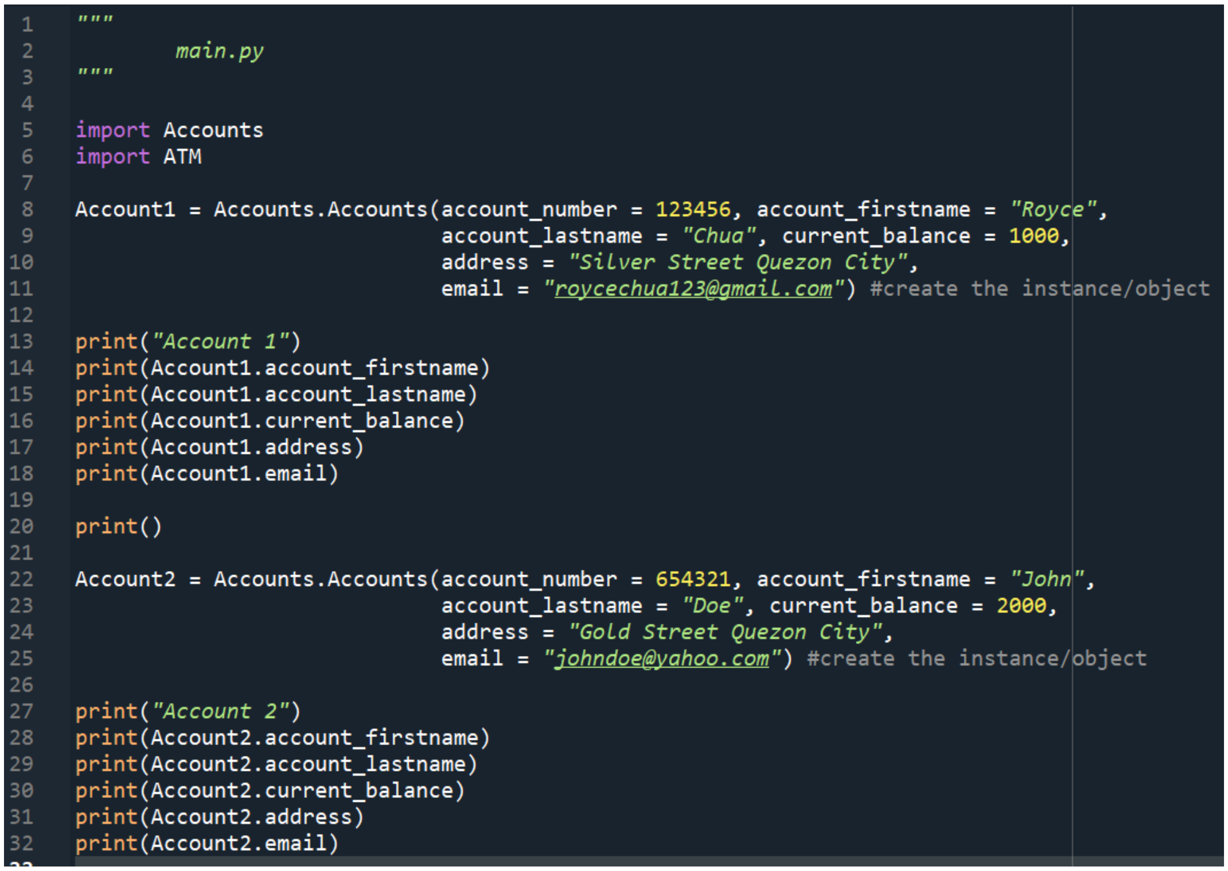
Overall Code + Output:



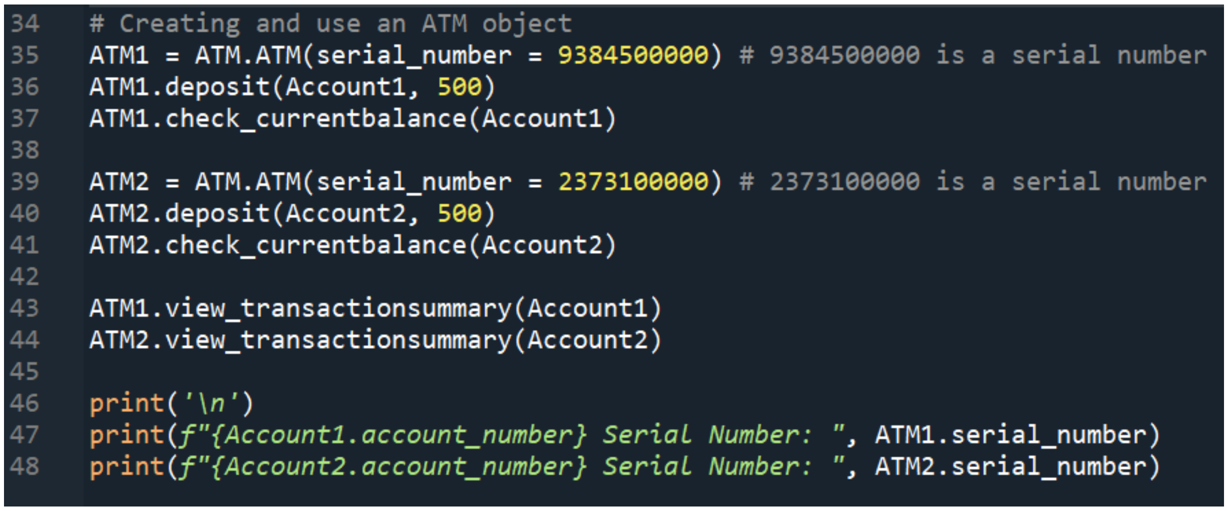
```python
"""
        Accounts.py
"""

class Accounts():
    def __init__(self, account_number, account_firstname, account_lastname,
                 current_balance, address, email): # Constructor class
        self.account_number = account_number
        self.account_firstname = account_firstname
        self.account_lastname = account_lastname
        self.current_balance = current_balance
        self.address = address
        self.email = email

    def update_address(self, new_address):
        self.address = new_address

    def update_email(self, new_email):
        self.email = new_email
```



```python
"""
        ATM.py
"""

class ATM():
    def __init__(self, serial_number):  # Constructor Function
        self.serial_number = serial_number
        self.transaction_log = []  # List to store transaction summaries

    def deposit(self, account, amount):
        account.current_balance = account.current_balance + amount
        self.transaction_log.append(f"{account.account_number} Deposited Amount: {amount}")
        print(f"\nAccount {account.account_number} Deposit Complete")
        print("Amount: ", amount)

    def withdraw(self, account, amount):
        if account.current_balance >= amount:
            account.current_balance = account.current_balance - amount
            self.transaction_log.append(f"{account.account_number} Withdraw Amount: {amount}")
            print(f"\nAccount {account.account_number} Withdraw Complete")
            print("Amount: ", amount)
        else:
            print(f"Account {account.account_number} Insufficient Balance.")

    def check_currentbalance(self, account):
        self.transaction_log.append(f"{account.account_number} New Balance: {account.current_balance}")

    def view_transactionsummary(self, account):
        print(f"\n{account.account_number} Transaction History:")
        if self.transaction_log:
            for transaction in self.transaction_log:
                print(transaction)
        else:
            print("No Transaction History")
```

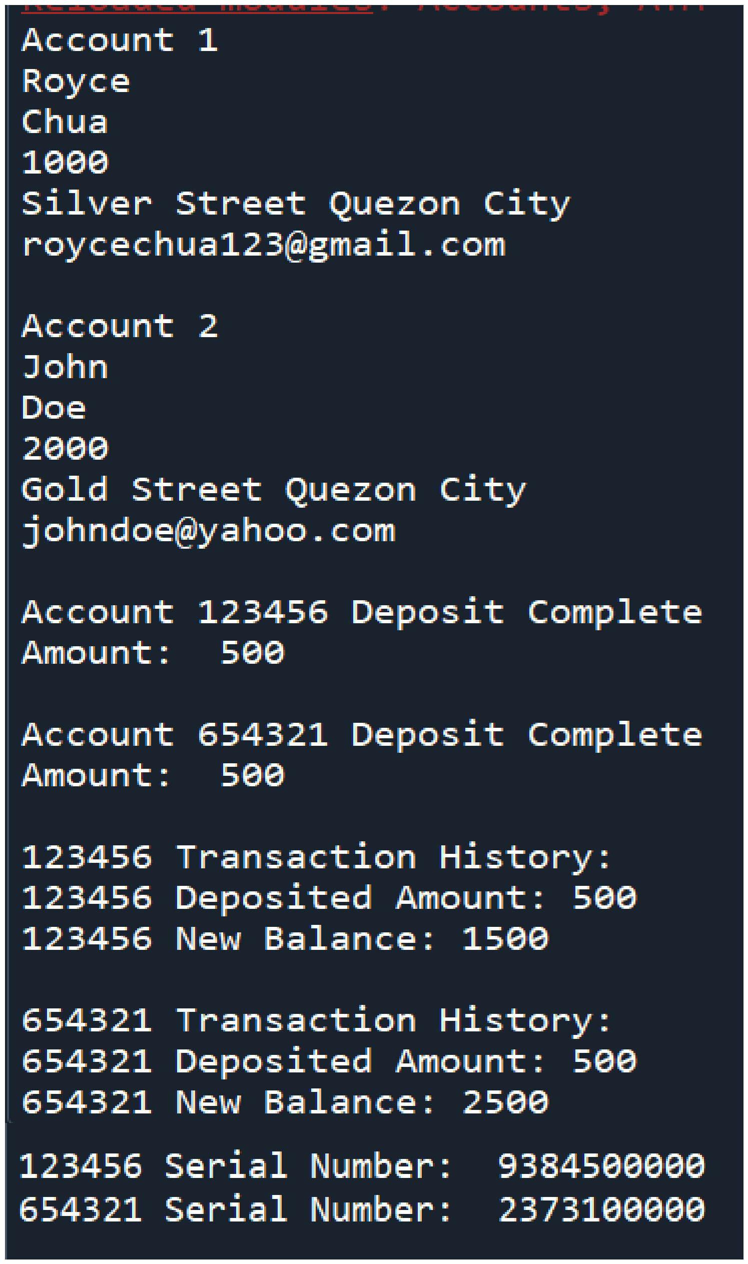


```python
"""
        main.py
"""

import Accounts
import ATM

Account1 = Accounts.Accounts(account_number = 123456, account_firstname = "Royce",
                             account_lastname = "Chua", current_balance = 1000,
                             address = "Silver Street Quezon City",
                             email = "roycechua123@gmail.com") #create the instance/object

print("Account 1")
print(Account1.account_firstname)
print(Account1.account_lastname)
print(Account1.current_balance)
print(Account1.address)
print(Account1.email)

print()

Account2 = Accounts.Accounts(account_number = 654321, account_firstname = "John",
                             account_lastname = "Doe", current_balance = 2000,
                             address = "Gold Street Quezon City",
                             email = "johndoe@yahoo.com") #create the instance/object

print("Account 2")
print(Account2.account_firstname)
print(Account2.account_lastname)
print(Account2.current_balance)
print(Account2.address)
print(Account2.email)
```



```python
# Creating and use an ATM object
ATM1 = ATM.ATM(serial_number = 9384500000) # 9384500000 is a serial number
ATM1.deposit(Account1, 500)
ATM1.check_currentbalance(Account1)

ATM2 = ATM.ATM(serial_number = 2373100000) # 2373100000 is a serial number
ATM2.deposit(Account2, 500)
ATM2.check_currentbalance(Account2)

ATM1.view_transactionsummary(Account1)
ATM2.view_transactionsummary(Account2)

print('\n')
print(f"{Account1.account_number} Serial Number: ", ATM1.serial_number)
print(f"{Account2.account_number} Serial Number: ", ATM2.serial_number)
```

Output:



```
Account 1
Royce
Chua
1000
Silver Street Quezon City
roycechua123@gmail.com

Account 2
John
Doe
2000
Gold Street Quezon City
johndoe@yahoo.com

Account 123456 Deposit Complete
Amount:  500

Account 654321 Deposit Complete
Amount:  500

123456 Transaction History:
123456 Deposited Amount: 500
123456 New Balance: 1500

654321 Transaction History:
654321 Deposited Amount: 500
654321 New Balance: 2500

123456 Serial Number:  9384500000
654321 Serial Number:  2373100000
```

Questions

1. What is a class in Object-Oriented Programming?
    - A class is a blueprint for creating objects (instances). Classes comprise of data for the object and the methods to operate on that data.

2. Why do you think classes are being implemented in certain programs while some are sequential(line-by-line)?
    - Because classes are easy to organize and structure code in a way that it can be reused, maintained, especially for complex systems. They contain data and methods, making it easier to model-real-world problems and manage large applications. In comparison, sequential (line by line) programming is often used in simpler programs where it does not need complex problem solving, and structure of object-oriented programming. Sequential programs are a straightforward/top-to-bottom flow, making them ideal for small tasks or scripts that do not require advanced functionality.

3. How is it that there are variables of the same name such account_firstname and account_lastname that exist but have different values?
    - Because they have different names, although they start with account_. Variables have their own memory location, and they can store different values independently of each other. These variable names are typically used to represent different uses of information related to the same concept.

4. Explain the constructor functions role in initializing the attributes of the class? When does the Constructor function execute or when is the constructor function called?

- The constructor function in a class is responsible for initializing the attributes of the class when an object (or instance) of that class is created. It sets up the initial state of the object by assigning values to its attributes. The constructor function (__init__(self, …..)) is called automatically when a new object is created from a class. This happens when you instantiate an object using the class name, followed by parentheses and any arguments required by the constructor

5. Explain the benefits of using Constructors over initializing the variables one by one in the main program?

- Constructors encapsulate the initialization logic within the class, hiding the details and making the code easier to manage. Some benefits are:
    - This approach ensures consistency by initializing all required attributes at once and prevents incomplete or incorrect object states.
    - It improves code readability and simplicity by allowing object creation in a single line, rather than setting each attribute manually.
    - Constructors also centralize initialization logic, so any changes need only be made in one place, ensuring all instances are updated accordingly.
- I concluded that constructors support dynamic input, making it flexible to initialize objects based

on varying values. Overall, constructors lead to cleaner, more maintainable code and reduce the risk of errors during object creation.

7. Conclusion:

I concluded that the code demonstrates good practices in object-oriented programming. The code showcases a well-structured and functional application of object-oriented principles. It effectively uses constructors for consistent initialization, containing logic within classes for better organization, and provides practical methods for managing and tracking financial transactions. The modular design and clear separation of concerns contribute to a maintainable and reusable codebase.

- Constructors in both the Accounts and ATM classes ensure consistent and complete initialization of objects, encapsulating the setup logic within the classes. This promotes cleaner, more maintainable code. The code's structure, with separate files for each class, enhances modularity and organization, making it easier to manage and extend.
- The ATM class provides essential functionality for handling deposits, withdrawals, and balance checks, while also maintaining a transaction log for better transparency. This allows the code to simulate real-world banking operations effectively. The approach of using reusable methods for different accounts adds flexibility and adaptability to the code.

Additionally, basic error handling, such as ensuring sufficient funds for withdrawals, makes the system more robust and prevents potential issues. Overall, the code showcases a well-structured application of object-oriented principles, ensuring maintainability, flexibility, and clarity in its design.