Zaitoon Ashraf IT Park

Artificial Intelligence & Data Science Course

Project Deadline 03rd January 2025

Mandatory Project:

Banking System Assignment (Functional Programming)

Objective

In this assignment, you will develop a simple banking system using **functional programming** concepts in Python. You will implement several features of a banking system, such as creating an account, depositing money, withdrawing money, checking the balance, and printing a transaction statement. The system will be implemented using functions and basic data structures like lists and dictionaries.

Learning Outcomes

By completing this assignment, you will:

1. Understand how to implement a basic banking system using functions.
2. Practice using dictionaries and lists to store and manage data.
3. Learn how to break a complex problem into smaller, manageable functions.
4. Work with Python's built-in functions and error handling mechanisms.
5. Reading and writing the data into text files

Assignment Details

You will build a **banking system** that performs the following tasks:

1. **Create an account** for a user with a name and initial balance.
2. **Deposit money** into the user's account.
3. **Withdraw money** from the user's account.
4. **Check balance** to view the current amount in the account.
5. **Print a transaction statement** showing all deposits and withdrawals.

You will use a dictionary to store account details, including the balance and a list to track transactions. Each action (deposit, withdrawal) will be represented as a separate function.

Step-by-Step Breakdown

Step 1: Create an Account

When a user opens an account, we will store their details in a **dictionary**. The dictionary will contain:

- The **account holder's name**.
- The **balance** (default is 0).
- A **list of transactions** (which starts as empty).

Data structure:

```
account = {
    "name": "John Doe",
    "balance": 0.0,
    "transactions": []
}
```

Step 2: Deposit Money

The **deposit function** will add money to the user's account balance. Each deposit will be recorded as a **transaction** (deposit) in the transaction document via file read write to store transactions permanently.

Function requirements:

- The deposit amount must be positive.
- Add the deposit amount to the current balance.
- Record the deposit in the transaction in file

Step 3: Withdraw Money

The **withdraw function** allows the user to withdraw money from the account.

Function requirements:

- The withdrawal amount must be positive.
- The user must have enough balance to withdraw the money.
- If the withdrawal is successful, subtract the withdrawal amount from the balance and record the transaction.
- If the balance is insufficient, print an error message.

Step 4: Check Balance

The **check_balance function** simply returns the current balance from the dictionary.

Step 5: Print Statement

The **print_statement function** will display all transactions made on the account (deposits and withdrawals) and the balance after each transaction.

Functional Breakdown

Let's break down the key **functions** you need to implement:

1. **Create Account**
    - This will initialize the account with a name, balance, and an empty transaction list.
2. **Deposit Function**
    - Takes the current account and the amount to deposit as inputs.
    - Updates the balance and records the deposit in the transaction list.
3. **Withdraw Function**
    - Takes the current account and the amount to withdraw.
    - Checks if the balance is sufficient and updates the balance if the withdrawal is successful.
    - Records the withdrawal in the transaction list or prints an error message if the withdrawal is not possible.
4. **Check Balance**
    - Simply returns the balance from the account.
5. **Print Statement**
    - Prints all transactions, showing the type of transaction (Deposit or Withdrawal) and the balance after each transaction.

Functional Programming Approach

Instead of using classes and objects, we will rely on **functions** and **data structures** like dictionaries and lists. Each function will take the current account data as input, modify it if necessary, and return the updated account data.

Example:

Let's imagine an account for a user named **John Doe**.

1. **Create Account**
   We initialize the account with a name and a starting balance of 0.
2. **Deposit Money**
   John deposits $500 into his account.
3. **Withdraw Money**
   John withdraws $200 from his account.
4. **Check Balance**
   John checks his balance.
5. **Print Statement**
   The statement shows all deposits and withdrawals made.

Example Workflow

1. **Create an Account**:
   You initialize an account for John Doe with a balance of $0.

   ```
   Account for John Doe created with balance $0.0.
   ```

2. **Deposit $500**:
   John deposits $500 into his account. The balance is now $500.

   ```
   John deposited $500. New balance: $500.0.
   ```

3. **Withdraw $200**:
   John withdraws $200. The balance is now $300.

   ```
   John withdrew $200. New balance: $300.0.
   ```

4. **Check Balance**:
   John checks his balance and sees $300.0.

5. **Print Statement**:
   The statement shows:

   ```
   Account statement for John Doe:
   - Deposit: $500. New Balance: $500.0
   - Withdrawal: $200. New Balance: $300.0
   ```

Additional Considerations

- **Error Handling**:
  You need to handle common errors such as:
  - Attempting to deposit a negative amount.
  - Trying to withdraw more money than the account balance.
  - Printing an empty statement if no transactions have occurred.
- **Edge Cases**:
  Test your functions with edge cases, such as:
  - Depositing a negative value.
  - Withdrawing more than the current balance.
  - Ensuring the account balance cannot go negative.

Submission Requirements

1. Submit a Python file that implements the following:
   - A function to create an account.
   - A function to deposit money into the account.
   - A function to withdraw money from the account.
   - A function to check the balance.
   - A function to print the statement.
2. Your code should handle basic errors and provide clear feedback to the user.
3. You are expected to test your program with different scenarios (e.g., depositing negative amounts, withdrawing more than the balance, etc.).
4. Include comments in your code to explain what each function does.

Grading Criteria

- **Correctness (50%)**: Does the program perform the expected operations (deposit, withdraw, balance check, print statement)?
- **Code Quality (30%)**: Is the code well-organized and easy to understand?
- **Edge Case Handling (10%)**: Does the program handle common edge cases and errors (e.g., insufficient balance)?
- **Efficiency (10%)**: Is the code simple and efficient in solving the problem?

Additional Tips

- Make sure you **test your code** thoroughly. Try different inputs to ensure the program behaves as expected.
- Break down the problem into smaller parts and write each function separately.
- Always test your functions with different scenarios before moving to the next function.

Good luck, and enjoy building your banking system!
