Lab 1 Requirement

Task Overview

In this lab, you will implement a complete bank account system using the principles of Object-Oriented Programming (OOP). You will apply encapsulation, inheritance, polymorphism, and abstraction in your code. This task is structured to reinforce your understanding of these concepts, with clear steps to follow.

Deliverables (80% Total)

Step 1: Base Class `Account` (30%)

- Create a base abstract class `Account` that models a generic bank account.
  - **Attributes**:
    - Protected attribute `_balance` (initial value is `0`)..
  - **Methods**:
    - `deposit(self, amount)`: Adds to the `_balance`. Ensure the deposit amount is positive. Raise a `ValueError` if it isn't.
    - `get_balance(self)`: Returns the current balance as a float.
  - **Abstract Methods**:
    - `withdraw(self, amount)`: An abstract method to be implemented by subclasses.

Step 2: Subclass `SavingsAccount` (25%)

- Create a subclass `SavingsAccount` inheriting from `Account`.
  - **Additional Attributes**:
    - `interest_rate`: A float representing the interest rate (default is `0.05` or 5%).
  - **Methods**:
    - `calculate_interest(self)`: Returns the interest based on the current balance (`interest = balance * interest_rate`).
    - Implement the `withdraw(self, amount)` method. This method should raise a `NotImplementedError` since withdrawals are not allowed from savings accounts.

Step 3: Subclass `CheckingAccount` (15%)

- Create a subclass `CheckingAccount` inheriting from `Account`.
  - **Methods**:

- Implement the `withdraw(self, amount)` method. Ensure that the withdrawal does not result in a negative balance. If the balance is insufficient, raise a `ValueError`.

Test Cases (20%)

Below are specific test cases that you must implement and ensure your code passes. Use these to verify your implementation.

Test Case 1: Deposit in Savings Account

1. Create a `SavingsAccount` with an initial balance of `1000`.
2. Deposit `500` into the account.
3. Call `get_balance()` and expect the balance to be `1500`.
4. Calculate interest using `calculate_interest()`. Expect the interest to be `1500 * 0.05 = 75`.

Expected Outcome:

```
print(savings.get_balance())  # Expected: 1500.0

print(savings.calculate_interest())  # Expected: 75.0
```

Test Case 2: Withdrawal from Checking Account

1. Create a `CheckingAccount` with an initial balance of `500`.
2. Deposit `200` into the account.
3. Withdraw `400` from the account.
4. Call `get_balance()` and expect the balance to be `300`.
5. Attempt to withdraw `500` and expect a `ValueError` due to insufficient balance.

Expected Outcome:

```
checking = CheckingAccount(500)

checking.deposit(200)

checking.withdraw(400)

print(checking.get_balance())  # Expected: 300.0

try:

    checking.withdraw(-500)
```

```
except ValueError as e:

# Expected: ValueError: Insufficient balance
```

Test Case 3: Prevent Withdrawal from Savings Account

1. Create a `SavingsAccount` with an initial balance of `1000`.
2. Attempt to withdraw `100` from the savings account and expect a `NotImplementedError` since withdrawals are not allowed.

Expected Outcome:

```
savings = SavingsAccount(1000)

try:

    savings.withdraw(100)

except NotImplementedError as e:

# Expected: NotImplementedError
```

Test Case 4: Invalid Deposit

1. Create a `CheckingAccount` with an initial balance of `500`.
2. Attempt to deposit `-100` and expect a `ValueError` due to the invalid deposit amount.

Expected Outcome:

```
checking = CheckingAccount(500)

try:

    checking.deposit(-100)

except ValueError as e:

# Expected: ValueError: Deposit amount must be positive
```

Grading Criteria

Code (80%): Focus on correct implementation of OOP principles.

Total Breakdown:

Step 1 (Base Class Account): 30%

Step 2 (Subclass SavingsAccount): 25%

Step 3 (Subclass CheckingAccount): 25%

Test Cases: 20%

Questions (20%): Based on concise and clear understanding of key OOP concepts and the implemented code.