Programming and Data Structures with Python Lab
Lab7. Object Oriented Bank in Python

Question1. Create a new class called **Account**.

1. Define a new class **Account** to represent a type of bank account.
2. When the class is instantiated you should provide the account number, the name of the account holder, an opening balance and the type of account (which can be a string representing 'current', 'deposit' or 'investment' etc.). This means that there must be an **__init__** method and you will need to store the data within the object.
3. Provide three instance methods for the Account: **deposit(amount)**, **withdraw(amount)** and **get_balance()**. The behaviour of these methods should be as expected, deposit will increase the balance, withdraw will decrease the balance and get_balance() returns the current balance.
4. Define a simple test application to verify the behaviour of your Account class.

It can be helpful to see how your class Account is expected to be used. For this reason a simple test application for the Account is given below:

```
acc1 = Account('123', 'John', 10.05, 'current')
acc2 = Account('345', 'John', 23.55, 'savings')
acc3 = Account('567', 'Phoebe', 12.45, 'investment')
print(acc1)
print(acc2)
print(acc3)
acc1.deposit(23.45)
acc1.withdraw(12.33)
print('balance:', acc1.get_balance())
```

The following output illustrates what the result of running this test application might look like:

```
Account[123] - John, current account = 10.05
Account[345] - John, savings account = 23.55
Account[567] - Phoebe, investment account = 12.45
balance: 21.17
```

The source code is given below for your reference as a starting point for all exercises

```
class Account:
    """" A class used to represent a type of account """

    def __init__(self, account_number, account_holder,
opening_balance, account_type):
        self.account_number = account_number
        self.account_holder = account_holder
        self.balance = opening_balance
        self.type = account_type

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

    def get_balance(self):
        return self.balance

    def __str__(self):
        return 'Account[' + self.account_number +'] - ' + \
               self.account_holder + ', ' + self.type + ' account = 
' + str(self.balance)


acc1 = Account('123', 'John', 10.05, 'current')
acc2 = Account('345', 'John', 23.55, 'savings')
acc3 = Account('567', 'Phoebe', 12.45, 'investment')

print(acc1)
print(acc2)
print(acc3)

acc1.deposit(23.45)
acc1.withdraw(12.33)
print('balance:', acc1.get_balance())
```

Question2. Keep track of number of instances of **Account**

- We want to allow the Account class to keep track of the number of instances of the class that have been created.
- Print out a message each time a new instance of the Account class is created.
- Print out the number of accounts created at the end of the previous test program.

For example add the following two statements to the end of the program:

```
print('Number of Account instances created:', 
Account.instance_count)
```

Question3. Create sub classes for **Account** class

The aim of these exercises is to extend the Account class you have been developing from the last two chapters by providing DepositAccount, CurrentAccount and InvestmentAccount subclasses.

Each of the classes should extend the Account class by:

- **CurrentAccount** adding an overdraft limit as well as redefining the withdraw method.
- **DepositAccount** by adding an interest rate.
- **InvestmentAccount** by adding an investment type attribute.

These features are discussed below:

The CurrentAccount class can have an overdraft_limit attribute. This can be set when an instance of a class is created and altered during the lifetime of the object. The overdraft limit should be included in the **__str__()** method used to convert the account into a string.

The CurrentAccount withdraw() method should verify that the balance never goes below the overdraft limit. If it does then the withdraw() method should not reduce the balance instead it should print out a warning message.

The DepositAccount should have an interest rate associated with it which is included when the account is converted to a string.

The InvestmentAccount will have a investment_type attribute which can hold a string such as ‘safe’ or ‘high risk’.

This also means that it is no longer necessary to pass the type of account as a parameter—it is implicit in the type of class being created.

For example, given this code snippet:

```
# CurrentAccount(account_number,account_holder,opening_balance,
# overdraft_limit)
acc1 = CurrentAccount('123', 'John', 10.05, 100.0)

# DepositAccount(account_number, account_holder, opening_balance,
# interest_rate)
acc2 = DepositAccount('345', 'John', 23.55, 0.5)

# InvestmentAccount(account_number, account_holder,opening_balance,
# investment_type)
acc3 = InvestmentAccount('567', 'Phoebe', 12.45, 'high risk')

acc1.deposit(23.45)
acc1.withdraw(12.33)
print('balance:', acc1.get_balance())

acc1.withdraw(300.00)
print('balance:', acc1.get_balance())
```

Then the output might be:

```
balance: 21.17
Withdrawal would exceed your overdraft limit
balance: 21.17
```

<u>**Question4.** Add Properties to **Account** class</u>

Convert the **balance** into a read only property, then verify that the following program functions correctly:

```
acc1 = CurrentAccount('123', 'John', 10.05, 100.0)
acc2 = DepositAccount('345', 'John', 23.55, 0.5)
acc3 = acc3 = InvestmentAccount('567', 'Phoebe', 12.45,'high risk')

print(acc1)
print(acc2)
print(acc3)

acc1.deposit(23.45)
acc1.withdraw(12.33)
print('balance:', acc1.balance)
print('Number of Account instances created:', Account.instance_count)

print('balance:', acc1.balance)
acc1.withdraw(300.00)
print('balance:', acc1.balance)
```

The output from this might be:

```
Creating new Account
Creating new Account
Creating new Account
Account[123] - John, current account = 10.05, overdraftlimit: -100.0
Account[345] - John, savings account = 23.55, interestrate: 0.5
Account[567] - Phoebe, investment account = 12.45, balance: 21.17

Number of Account instances created: 3
balance: 21.17
Withdrawal would exceed your overdraft limit
balance: 21.17
```

Question5. Add **Error Handling** routines

This exercise involves adding error handling support to the CurrentAccount class.

In the CurrentAccount class it should not be possible to withdraw or deposit a negative amount.

Define an exception/error class called **AmountError**. The AmountError should take the account involved and an error message as parameters.

Next update the deposit() and withdraw() methods on the Account and CurrentAccount class to raise an AmountError if the amount supplied is negative.

You should be able to test this using:

```
try:
      acc1.deposit(-1)
except AmountError as e:
      print(e)
```

This should result in the exception 'e' being printed out, for example:

```
AmountError (Cannot deposit negative amounts) on Account[123] -
John, current account = 21.17 overdraft limit: -100.0
```

Next modify the class such that if an attempt is made to withdraw money which will take the balance below the over draft limit threshold an Error is raised.

The Error should be a **BalanceError** that you define yourself. The BalanceError exception should hold information on the account that generated the error.

Test your code by creating instances of CurrentAccount and taking the balance below the overdraft limit.

Write code that will use try and except blocks to catch the exception you have defined.

You should be able to add the following to your test application:

```
try:
      print('balance:', acc1.balance)
      acc1.withdraw(300.00)
      print('balance:', acc1.balance)
except BalanceError as e:
      print('Handling Exception')
      print(e)
```

Question6. Package all classes into a separate module

The aim of this exercise is to create a module for the classes you have been developing.

You should move your Account, CurrentAccount, DepositAccount and BalanceError classes into a separate module (file) called **accounts**. Save this file into a new Python package called **fintech**.

Separate out the test application from this module so that you can import the classes from the package.

Your test application will now look like:

```
import fintech.accounts as accounts

acc1 = accounts.CurrentAccount('123', 'John', 10.05, 100.0)
acc2 = accounts.DepositAccount('345', 'John', 23.55, 0.5)
acc3 = accounts.InvestmentAccount('567', 'Phoebe', 12.45, 'high risk')

print(acc1)
print(acc2)
print(acc3)

acc1.deposit(23.45)
acc1.withdraw(12.33)
print('balance:', acc1.balance)

print('Number of Account instances created:',
accounts.Account.instance_count)

try:
      print('balance:', acc1.balance)
      acc1.withdraw(300.00)
      print('balance:', acc1.balance)
except accounts.BalanceError as e:
      print('Handling Exception')
      print(e)
```

You could of course also use **from accounts import *** to avoid prefixing the accounts related classes with accounts.

Question7. Convert Account as **Abstract Class**

The Account class of the project you have been working on throughout the last few chapters is currently a concrete class and is indeed instantiated in our test application.

Modify the Account class so that it is an Abstract Base Class which will force all concrete examples to be a subclass of Account.

The account creation code element might now look like:

```
acc1 = accounts.CurrentAccount('123', 'John', 10.05, 100.0)
acc2 = accounts.DepositAccount('345', 'John', 23.55, 0.5)
```

```
acc3 = accounts.InvestmentAccount('567', 'Phoebe', 12.45, 'risky')
```

Question8. Create History of Transactions using **Lists**

You should modify your Account class such that it is able to keep a history of transactions.
A Transaction is a record of a deposit or withdrawal along with an amount.

Note that the initial amount in an account can be treated as an initial deposit.

The history could be implemented as a list containing an ordered sequence to transactions. A Transaction itself could be defined by a class with an action (deposit or withdrawal) and an amount.

Each time a withdrawal or a deposit is made a new transaction record should be added to a transaction history list.

Now provide support for iterating through the transaction history of the account such that each deposit or withdrawal can be reviewed. You can do this by implementing the Iterable protocol—refer to the last chapter if you need to check how to do this. Note that it is the transaction history that we want to be able to iterate through—so you can use the history list as the basis of your iterable.

You should be able to run this following code at the end of your Accounts application:

```
for transaction in acc1:
      print(transaction)
```

Depending upon the exact set of transactions you have performed (deposits and withdrawals) you should get a list of those transactions being printed out:

```
Transaction[deposit: 10.05]
Transaction[deposit: 23.45]
Transaction[withdraw: 12.33]
```

Reference: This exercise has been copied from the book, ***"John Hunt, A Beginners Guide to Python3 Programming, Springer, 2019***"