CA2 - Bank Accounts

Due 20 Dec 2021 by 23:59 **Points** 100 **Submitting** an external tool
Available until 20 Dec 2021 at 23:59

This assignment was locked 20 Dec 2021 at 23:59.

COMP517 Continuous Assessment Task 2

Assignment Number: 2 of 3

Deadline for submission: **15th December 2021, 23.59**

Submission Mode: Please provide your solutions electronically via Canvas, via the integrated CodeGrade system. **You should submit a single .py file.**

Purpose of the assessment: This assignment will test the ability of the students to work with Object Oriented Programming in Python.

Learning Outcomes Assessed:

LO1. Demonstrate knowledge of fundamental imperative programming concepts such as variables and assignment, conditional statements, loops and methods.

LO2. Be able to design and code applications in a suitable programming language.

LO3. Critical knowledge of concepts and principles of object-orientation such as objects and classes, encapsulation, object state, coupling, cohesion and modularity.

LO4. Critical awareness of important principles of software design and development, including appropriate naming of variables and classes, code layout, testing and debugging, and documentation.

Marking Criteria: Based on the marking descriptors of the University's Code of Practice on Assessment

Late submissions: For late submissions the standard penalties apply, according to UoL policy.

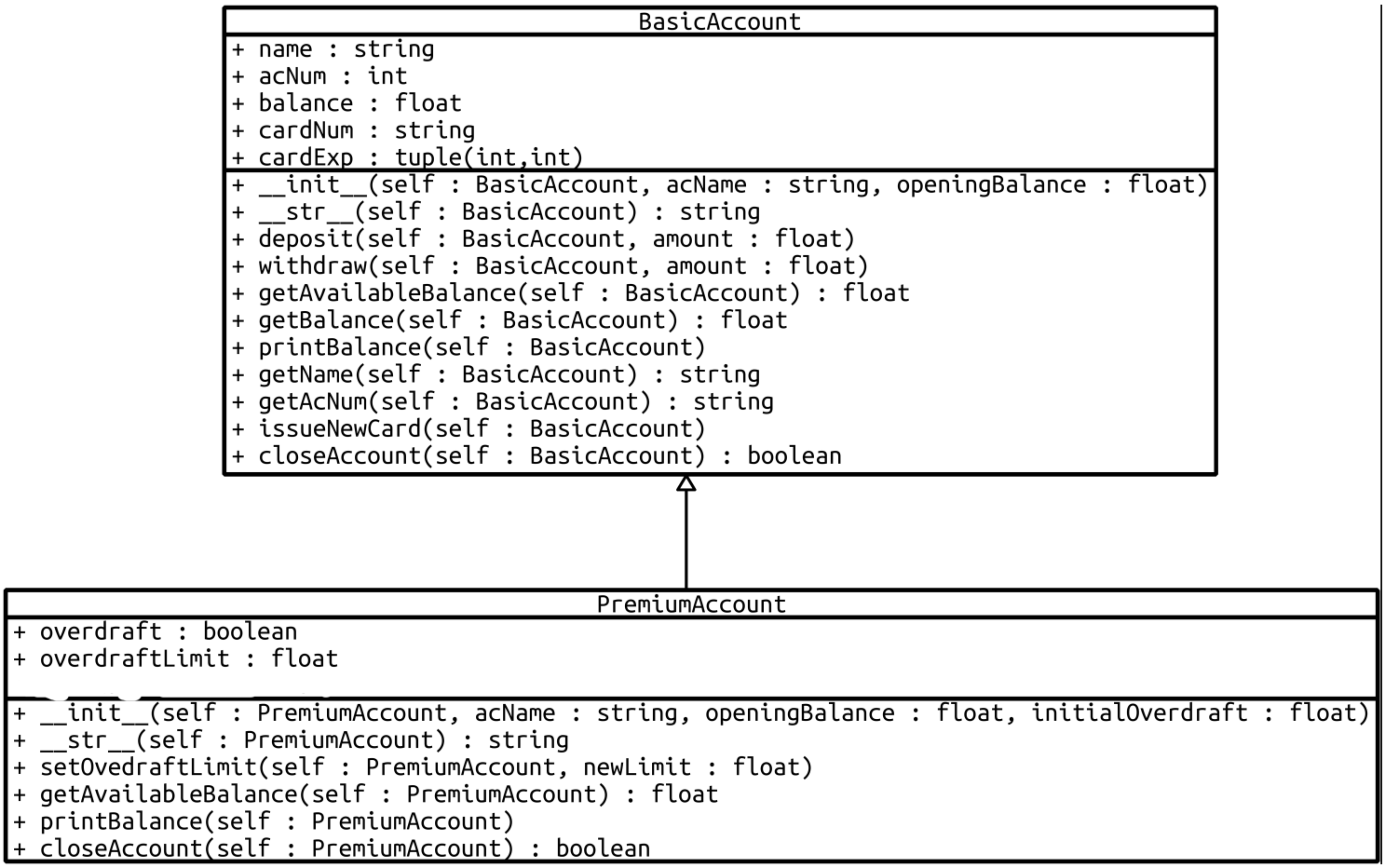
Plagiarism: Please provide individual answers and do not coordinate with other students to complete the assignment. Check the official UoL guidelines for plagiarism and academic integrity.
The submitted files will be checked using tools for plagiarism and collusion.

You may use the "Discussions" to ask clarification questions about the assignment, but please ***do not ask any questions that would give away the solutions or even parts of the solutions.***

Continuous Assessment Task 2 - Bank Accounts

In this assignment, you will design and implement two classes in Python: **BasicAccount** and **PremiumAccount.**

The classes need to be adherent to the following **UML Class diagram**, though you may add extra methods and variables:



To read the UML diagram, a line of **<varName> : <dataType>** means that **varName** is an object of that data type. For example, "name" is an object of type "string".

A line of **<methodSignature> : <dataType>** means that the method will return a particular data type. For example, the "getBalance" method will return a float data type. If for a method the data type is not specified, then
the method will not return anything.

Variables:

The variables of the classes are described as follows:
(note that these are not class variables, but instance variables. You'll have to decide whether to use class variables and how).

- **name** – the account holder's name.
- **acNum** – the number of the account. This should be “serial”, meaning that the first account to be created should have number 1, the second account to be created should have number 2, and so on.
- **balance** – The balance (in pounds) of the account.
- **cardNum** – The card number, which should be a string containing a 16-digit number (you should import and use the *random* module for this).
- **cardExp** - a tuple, where the first element is an integer corresponding to the month and the second element is 2-digit year. Eg: 03/23 represents March 2023. (you should import and use the *datetime* module for this).
- **overdraft** – a Boolean variable, which is True if the account has an overdraft, and False if does not.
- **overdraftLimit** – The amount that the account can go overdrawn by.

Methods:

The methods are as follows:

- **__init__(self, str, float)**

  Initialiser giving the account name and opening balance.
- **__init__(self, str, float, float)**
  initialiser giving the account name, opening balance, and overdraft limit (0 or above).
- **deposit(self, float)**
  Deposits the stated amount into the account, and adjusts the balance appropriately. Deposits must be a positive amount.
- **withdraw(self, float)**
  Withdraws the stated amount from the account, prints a message of “<Name> has withdrawn £<amount>. New balance is £<amount>”.
  If an invalid amount is requested, then the following message should be printed, and the method should then terminate: “Can not withdraw £<amount>".
  An amount is considered to be invalid if it is larger than the balance for (normal) accounts and if it is larger than the balance + overdraft limit for premium accounts.
- **getAvailableBalance(self)**
  Returns the total balance that is available in the account as a float. **It should also take into account any overdraft that is available.**

- **getBalance(self)**
  returns the balance of the account as a float. **If the account is overdrawn, then it should return a negative value.**

- **printBalance(self)**
  Should print to screen in a sensible way the balance of the account. If an overdraft is available, then this should also be printed and it should show how much overdraft is remaining.

- **getName(self)**
  Returns the name of the account holder as a string.

- **getAcNum(self)**
  Returns the account number as a string.

- **issueNewCard(self)**
  Creates a new card number, with the expiry date being 3 years to the month from now (e.g., if today is 1/12/21, then the expiry date would be (12/24)).

- **closeAccount(self)**
  To be called before deleting of the object instance. Returns any balance to the customer (*via the withdraw method*) and returns True.
  Returns False if the customer is in debt to the bank, and prints message "Can not close account due to customer being overdrawn by £<amount>".
  Clarification: You should not actually delete the account instance. This function will simply do the relevant "housekeeping" in the account, and return a Boolean value.

- **setOverdraftLimit(self, float)**
  Sets the overdraft limit to the stated amount

In addition to the above, you must also define suitable string representations that give the account name, available balance, and overdraft details (that is: you need to also implement the __str__ methods for each class).

Instructions and Marking Criteria

In your program you **should** follow the specified class, variable and method names and signatures, If you do not follow the specified function names and signatures, then the correctness of your program may not be able to be correctly determined; you may also lose marks for not following the brief.

You **should not** use any libraries or import any modules, besides the ones that are explicitly mentioned in the brief.

You **should** submit a single file, named **accounts.py**

You may write some test code in the main body, but **this must not run if your file is imported into another module**. If this is not ensured, the correctness of your program might not be able to be verified.

Further to the guidelines on CANVAS, the mark will be calculated with the following proportions:

Correctness against Brief and Robustness: 45%

Design Choices: 40%

Readability and layout: 15%