CA2 - Specification

CA2 - 29/11/2021
Class Group: TU856 / TU858

Worth: 30% of the overall mark for the module (The sum of CAs marks are capped at 30%)	**Submission:** 18/12/21 **Late submission penalty:** 4^d%, where d is the number of days. No submission allowed after 3 days.

Pairs: You can work on this project individually or in pairs. If you do it in a pair you have to email me (lucas.rizzo@tudublin.ie) by 06/12/2021 to let me know the pair names and which project you are solving.

Report (submissions without a report will not be marked)
Understanding of the code will be evaluated based on a report written to explain your system. It should contain one page describing the project classes and its methods, one paragraph as a user manual, one paragraph describing team members' individual contributions, and one paragraph describing the difficulties and more challenging parts.

Submission
Please submit your solution (Python files, any necessary external files, and report) via Brightspace. It is your responsibility to make sure you've uploaded the correct file as changes after the deadline won't be accepted.

Please make sure that both members of the pair submit the same solution on Brightspace.

Marking Scheme

Correct functionality (50%), Creativity (20%), Layout and use of comments (10%), Report (20%). More about marks at the end of the sheet.

Plagiarism

Plagiarism will result in a zero mark (0%). You should make yourself familiar with the plagiarism policy of Technological University Dublin. **You might be contacted by the lecturer to demo your code if the submission is believed to contain suspected plagiarism.**

Problems Description:

For this assignment you will have two choices of problems: a library management system and a bank management system.

Library Management System:

You are asked to develop an application to manage library services, such as borrowing and returns activities. You should be able to **borrow/return** a **book**, an **article** in a journal, or **digital media**. All the library information should be stored in **four external files**: `library.txt`, `items.txt`, `members.txt`, and `borrowing.txt`. It is up to you to define the structure of each file, but each member, item or transaction should have an unique ID.

Use Python classes to implement the library. Some of the functionality your system should provide includes:

- At least the following classes: Library, Items, Books, Articles, Digital Media, Members
- Books, Articles, Digital Media should be a **subclass** of Items.
- Each class should have an **__init__** and **__str__** methods. For each __str__ method, think what information each class should provide when you print their instances.
- Persistent memory: when you start your system it should read the files `library.txt`, `items.txt`, `members.txt`, and `borrowing.txt` in the same folder as the python code and create all the necessary instances.
- Provide a command line interface for the user to:
    - Add/edit/delete instances belonging to each class,
    - Members browse library items and select items to borrow.
    - Members returning borrowed items.
    - Make sure to update the external files after any information is modified.

Bank Management System:

You are asked to develop an application to manage bank services. An account is a general account class that contains **balance**, **deposit**, **transfer** (send money to another account in the bank) and **withdrawal** methods. Your bank should allow the creation of **two bank accounts types**. The two types of accounts should include:

- Savings accounts: these only allow one withdrawal or one transfer per month, and might also be opened by teenagers from 14 years old.
- Checking accounts: these are regular accounts that can be opened by customers who are 18 years or older. They can also have a negative balance to a specified credit limit.

All the bank information should be stored in three external files: `customers.txt`, `accounts.txt`, `accountsTransactions.txt`. It is up to you to define the structure of each file, but each customer, account or transaction should have an unique ID.

Use Python classes to implement the bank system. Some of the functionality your system should provide includes:

- At least the following classes: Account, Saving Account, Checking Account, Customer.

- Saving Account and Checking Account should be a subclass of Account.
- Each class should have an **__init__** and **__str__** methods. For each __str__ method, think what information each class should provide when you print their instances.
- Persistent memory: when you start your system it should read the files `customers.txt`, `accounts.txt`, and `accountsTransactions.txt` in the same folder as the python code and create all the necessary instances.
- Provide a command line interface for the user to:
    - Customer creating a new account.
    - Customer viewing the transactions performed in one of his accounts and the respective balance.
    - Customer performing the operations allowed by its account type.
    - Customer deletes his/her account.
    - Make sure to update the external files after any information is modified.

Hints and General Guidelines:

- Start with the easy parts. Define only your constructors and add new functionalities one by one.
- Once a method has been coded test it before moving on to the next part
- Make sure you include any relevant **error checking** and handle unexpected input
- Make sure to use **function annotations**
- Once you have defined a class you can use its instances as any other type. Remember to use **composition, aggregation and inheritance** if applicable.
- The easiest way to update external files is likely to write content to a new file and replace the old file with the new file. To remove a file use the ‘remove’ function from the ‘os’ module.
- Think about which data structure is more appropriate to the pieces of information you need to store: list, string, dictionary, tuple, etc.
- Think if each attribute needs to be **private or public**. Remember to add **get/set** methods for private attributes that are accessed outside the class.
- Add **docstrings** to all your classes and methods
- After implementing a method and testing, see if it can be improved. You don’t need to do an optimal solution in the first attempt. Most of the time you can reduce the number of lines and make an algorithm more elegant after an initial solution has been provided.

Sample Marking Schemes

Correct functionality (50%)

- Code compiles and runs with no problem.
- All required classes have been coded.
- Required functionalities work properly.
- Inheritance and composition/aggregation applied.
- Ability to handle files and deal with exception handling.
- Functions/methods are used correctly to break down the problem. They are all used for a single purpose or for a single task.
- Nice use of data structures such as dictionaries and sets.
- Command line menu is well implemented, and all its options work as expected

Creativity (20%)

- Additional features that were not required and that fit well in the system have been included. For example:
    - Think about what else a library/bank system might have. Perhaps different types of items, customers, members, etc.
    - Additional classes have been developed.
    - Operator overloading has been used for some operations.
    - Use of other concepts you self-study outside the content delivered in the course **(please add references for this, i.e. by adding a URL as a comment in your code)**

Layout and use of comments (10%)

Indentation and white space have been used appropriately. Code is easy to read, and naming conventions have been adopted. Code is well documented, and it is easy to understand.

Report (20%, submissions without a report will not be marked)

Understanding of the code will be evaluated based on a report written by you or your pair to explain your system. It should contain one page describing the project classes and its methods, one paragraph as a user manual, one paragraph describing team members' individual contributions, and one paragraph describing the difficulties and more challenging parts.

Rubric

	Expectations Far Surpassed (100%)	Expectations Exceeded (75%)	Expectations Met (50%)	Expectations Not Met (25%)	Not Done (0%)
Functionality	All requirements are met. All deliverables included in submission without deviation from requirements.	Requirements met. Deliverables included without major and few minor omissions or deviations from requirements.	Basic requirements met. Deliverables included without major omissions or deviations from requirements	Basic requirements not met or deliverables deviate substantially from requirements	Not convincingly attempted
Creativity	Additional features are well thought and surpass expectations. They add complexity to the system, make it more interesting, useful, and close to a real system for the project area.	Additional features are well thought. They add complexity to the system and make it more useful.	Additional features are well implemented and work as expected. Not much complexity, but a basic implementation.	Additional features are attempted but not well implemented and/or not working as expected.	Not convincingly attempted
Layout and use of comments	Layout and comments are flawless. Function annotations and docstrings have been used throughout the whole code. Naming conventions have been adopted. Comments are meaningful and make the reasoning more clear.	Layout and comments are well implemented. Function annotations and docstrings used throughout most part of the code.	Layout and comments are ok. The code is well organised and not difficult to understand.	Code is not well organised. The main scope is not linear and it is hard to follow. Few or no comments. The code is hard to understand and requires a lot of thinking to follow.	Not convincingly attempted
Report	Report is complete, well written and organised. It follows the appropriate length without being repetitive. Difficulties and challenges are well described and related to the solution delivered. User manual and class description are meaningful and help using and understanding the code.	Report is complete, well written and organised. Difficulties and challenges are well described. User manual and class description add relevant details.	Report contains all the sections. It describes all classes and adds values to the user. User manual is complete. Some challenges and difficulties are mentioned but could likely be expanded. Individual contributions are ok, but not detailed.	Report is incomplete. Significant parts are missing, such as classes documentation and/or user manual. Individual contributions are not well explained.	Not convincingly attempted