ITECH1400 - Assignment 2 – Money Manager

Due Date: Friday of Week 11 at 5pm

This assignment will test your skills in designing and programming applications to specification and is worth 20% of your non-invigilated (type A) marks for this course. This is an **INDIVIDUAL ASSIGNMENT** – and while you may discuss it with your fellow students, you must not share designs or code or you will be in breach of the university plagiarism rules. This assignment should take you approximately 20 hours to complete.

Assignment Overview

You are tasked with creating an application that uses a GUI that simulates a ***simple money manager***. This tool is used to track all spending of an individual.

The assignment is broken up into the following main components:

1.) The ability to provide a login screen to prevent other people from viewing the information contained in the tool

2.) The ability to ***view the current balance funds***, add purchases made and deposits into the balance

3.) The ability to save the transactions to a file so that you can log in, add transactions to the application, log out and then return to find all of the transactions still there – i.e. the data is persistent.

4.) The ability to ***display a graph of your spending*** broken down by type – i.e. food, bills, rent.

5.) A ***Test Case*** that ensures that your ***simple money manager*** works as required.

Your submission should consist three Python scripts that implement the computer program (**moneymanager.py, main.py** and **testmoneymanager.py**).

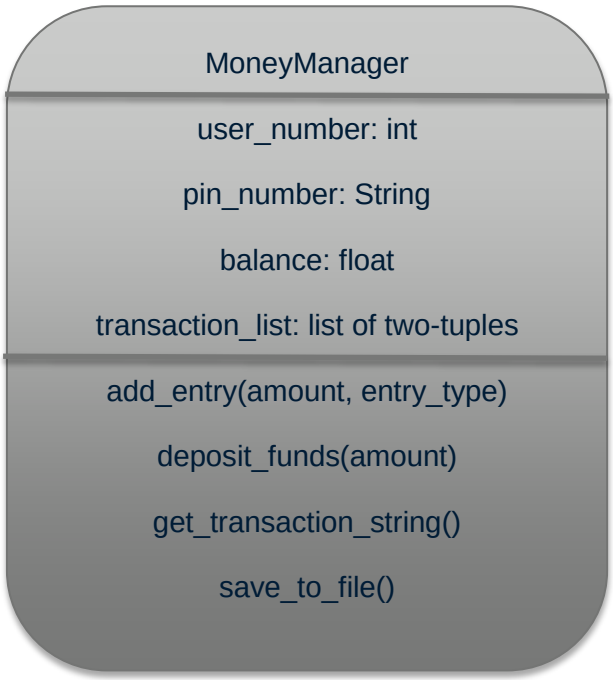
You are provided with a 'stub' of each of these files which contain all the function declarations and comments which describe the role of the function and how it can be put together, but you will have to write the code for vast majority of the functions yourself.

Your final submission should be a zipped archive (i.e. ‘zip file’) containing your completed Python scripts.

Assignment Part 1 Details

Class Design – Money Manager Tool

The following design for ***MoneyManager***



The following is the functionality of the class:

- add_entry – this adds a new entry into the money manager representing the spending of the user. This includes the amount spent and the type of item the money was spend on. It also removes the amount from the current balance. For the sake of simplicity the type of item the money was spent on can only be ***food, rent, bills, entertainment*** or ***other*** – any other type will raise an exception that should be handled
- Each transaction in the ***transaction_list*** is a tuple containing the word ***Deposit*** followed by an amount or ***EntryType*** followed by an amount
- The user of the tool cannot go into a negative balance so the user cannot spend money that is not available in their user account. So if they have $100 in their user account and then want to add an entry to spend $150, an exception will be raised with a suitable error message which is caught and displayed in the main.py file where the operation was attempted.
- All error messages, such as those from exceptions, should be displayed in a pop-up message box
- The get_transaction_string method should loop over all the transactions in the transaction_list creating a string version (with newline characters) of all the transactions associated with the user.
- The save_to_file function should save the user_number, pin_number, and balance in that order to a file called <user_number>.txt followed by the transaction list string generated from the get_transaction_string() method. The name of the user file is NOT '<user_number>.txt' - the name of the file is the ACTUAL USER NUMBER followed by ".txt", so for an user with user_number 123456 the name of the user file would be 123456.txt.

The Main Python Script

Our **main.py** script will contain all the main logic for our program. It will allow us to:

- Enter a user number via an Entry field by using the keyboard,
- Enter a PIN number via an Entry widget (we can use the keyboard OR a series of buttons to enter the PIN),
- Once we are logged in we must be able to:
    - See the balance of the user,
    - Deposit money for the user,
    - Add entries to represent spending of funds from our user (only up to the amount we have available),
    - Display a plot of spending
    - Log out of our user.

Every time a ***successful*** deposit or entry is made then a new transaction should be added to the user's transaction list. When we log out then the user file is overwritten with the new user details including our new balance and any transactions if any have been made.

The format of the user text file is as follows (each value on separate lines):

```
user_number
user_pin
currentBalance


123456
7890
800.00
```

After these first three lines we may have a number of transactions, each of which is specified as two lines. A *deposit* is indicated by the word **Deposit** on one line and then the amount on the next like. For example a deposit of $500 would look like this:

```
Deposit
500.00
```

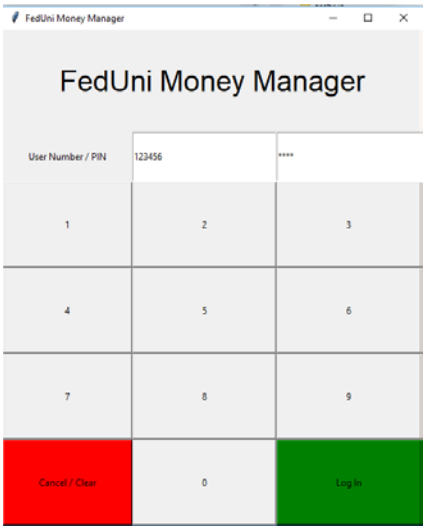
Similarly, a *entry* is also specified as two lines – first the word being the type of entry and then on the next line the amount, for example entries representing **rent** of $200 and **bills** of $250 would look like this:

```
Rent
200.00
Bills
250.00
```

You are provided with an example user file called **12345678.txt** – this file along with others will be used to mark your assessment, so you should make sure that your final submission can use users in this format successfully.

Login Screen

When the application is first launched, it should open a window that is "500x660" pixels in size (use the window object's geometry function to set this). Set the title of the window to "FedUni Money Manager" using the top-level window object's winfo_toplevel().title() function.



The window uses the GridManager layout manager for placing GUI elements (e.g. 'widgets'), it contains a Label that spans the top of the window saying "FedUni Money Manager" (font size is 28). On the next line is a label saying "User Number" and then an Entry widget for the user to type in their user number and an entry for the PIN number.

It then has a series of buttons from 0 through 9, along with a Log In button and a Clear/Cancel button.

Each time a number is pressed it is added to a string - for example, if the user pushed the 4 button then the 3 button then the 2 button and then the 1 button then the string should contain the text "4321". By using the show="*" attribute you can 'mask' the input so that anyone looking over your shoulder cannot see the exact pin number, they'll just see "****" instead. When the Clear/Cancel button is pressed, or when a user "logs out" then this PIN string should be reset to an empty string.

When the Log In button is pressed then the program should attempt to open the file with the user number followed by ".txt" - so in the example below, because the user number entered was "123456", the program will attempt to open the file "123456.txt".

If that file could not be opened then a messagebox should display a suitable error message such as "Invalid user number - please try again!". You will have to "try/catch" this risky functionality to avoid the program crashing - see the week 7 lecture materials if you need a recap.

If the user exists, then MoneyManager object should be created and the fields of the MoneyManager object should be set (e.g. user_number, pin_number, balance, and the transaction_list).

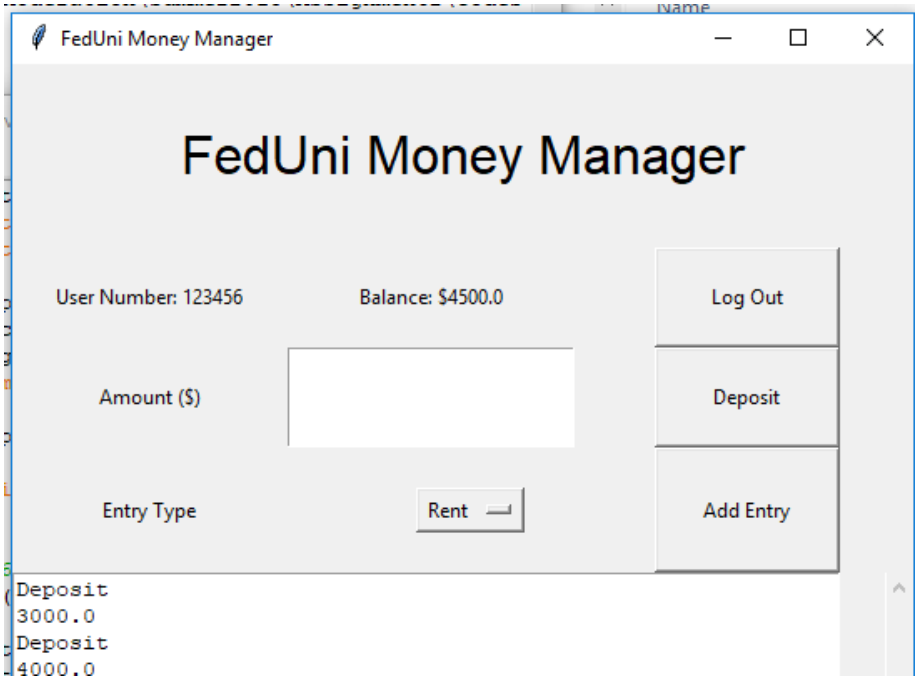
Because you don't know how many transactions are stored for this user, after reading the first three lines you will need to attempt to read two lines and if they exist create a tuple of the transaction (i.e. it's type and amount) and then add it to the MoneyManager object's transaction_list - for example you may do the following in a loop:

```
# Try to read a line, break if we've hit the end of the file
line = read_line_from_user_file()
if not line:
    break
else:
    # read another line, create a tuple from both lines and append the # tuple
    to the the MoneyManager object's transaction_list
```

The user screen has:
- A large "FedUni Money Manager" label at the top that spans 5 columns (font size is 22),
- A label with the user number followed by the actual user number displayed,

- A label with the current balance followed by the actual balance,
- A log out button which saves the user file (overwriting it) and causes all widgets to be removed from the screen and the **log in** screen displayed again,
- An "Amount" label followed by an amount **Entry** widget where you can type in how much to deposit or an entry added,
- Deposit and Withdraw buttons that deposit or withdraw funds using the MoneyManager classes methods to do so,
- A **Text** widget (i.e. multi-line text) that shows all the transactions associated with the user. The height of this widget is 10 lines and the width of the widget is 48 characters.
- To the right of the Text widget this is a scrollbar which can be used to scroll the Text widget (it is not really showing in the above screenshot because the Text widget does not have more than 10 lines worth of content), and finally
- At the bottom of the screen is a plot (histogram) of the spending of the user broken down by type

Below is the top section of the screen:



Money Manager Test Case

The final thing to do is to add a small test case consisting of seven unit tests that ensure that the **MoneyManager** class' **deposit_funds** and **add_entry** methods operate correctly.

You are provided with a **stub** of a **testmoneymanager.py** test case that already has the definitions of unit tests that you will write and a description of what they are testing. Your job is to write suitable assert statements into each of these unit tests so that they ensure the MoneyManager class' deposit and add entry functions do operate as they should. You should be able to accomplish each unit test in a maximum of two lines of code per unit test.

Two things to remember:

1.) The **setUp** method is automatically executed before each unit test, so the balance gets reset to **1000.0** before each test, and

2.) If your unit test fails (and the test is correct) then you should modify your **code** so that it passes the test, not the test so that it matches up with what your code is doing!

Development and Marking Tips

You will need to bind all the numerical log in buttons to the <Button-1> event and the same function, and then extract which button was pressed to help 'build up' the PIN number text. To extract which widget triggered the function use event.widget["text"].

You are provided with the complete function to remove all widgets from the top level window, the function to read a line from the user file which does NOT include the final "\n" newline character and most of the code to generate a graph and place it in your window - so you do not have to write these yourself.

The grid for the login screen has 3 columns and 6 rows, while the grid for the user screen has 5 columns and 5 rows. The fifth column on the user screen is the one holding the scrollbar.

Once you have your login screen working, it makes sense to set the user number variable and PIN number variables to '123456' and '7890' respectively so that they are already filled in when you launch the app and you can immediately click the "Log In" button to move to the user screen. Without this you will have to enter these values each time you run the program, which will be a nuisance.

When you are writing your code to save the money manager user object to file be aware that if the function fails before closing the file then the file will now be blank (i.e. it will not contain any user details). As such, it's probably best to keep a copy of the file close by so you can replace it if necessary.

The way this assignment is marked is a little different from assignment 1 where you could get marks for effort - that is, if something didn't quite work as it should you would still get marks. This was because you were designing the code and its logical flow. In this assignment you are given a precise specification of how the application should work, and the marking guide is more along the lines of "did you implement this feature as specified?, did you implement that feature as specified?".

The answer to these questions is largely a simple yes or no - if you did, you get the mark - and if you didn't then you don't get the mark. If your code doesn't run at all (i.e. it produces an error on startup) then it means your code cannot possibly implement the functionality, and your marks will be severely affected. Nobody wants this - so even if you do not implement the entire suite of functionality, please make sure that your code at least runs!

As a final reminder, please do not add functionality that is not described in this document. Rather than being rewarded with extra marks you will be penalised for not matching the project specification.

Submission and Marking Process

You must supply your program source code files and your document containing the first two section of the assignment as a single compressed archive called:

ITECH1400_Assignment_1_<YOUR-NAME>_<YOUR-STUDENT-ID>.zip

Obviously replace <YOUR-NAME> and <YOUR-STUDENT-ID) with your own personal details! You may supply your word processed documentation in either Microsoft Word or LibreOffice/OpenOffice formats only – no proprietary Mac specific formats, please.

Assignments will be marked on the basis of fulfilment of the requirements and the quality of the work. In addition to the marking criteria, marks may be deducted for failure to comply with the assignment requirements, including (but not limited to):

- Incomplete implementation(s), and
- Incomplete submissions (e.g. missing files), and
- Poor spelling and grammar.

Submit your assignment (all program source files plus your word processed document) Moodle.

The mark distribution for this assignment is explained on the next page – please look at it carefully and compare your submission to the marking guide.

Assignment 2 – FedUni Money Management Tool

Student name: **Student ID:**

Part	Assessment Criteria	Weight	Mark
1	Window is correct size with correct title	1	
2	Label displayed across the top in large font of login screen	1	
3	User number and pin at the top of the login screen	1	
4	All pin input is masked (ie uses * rather than numbers)	1	
5	Incorrect user number – error message	1	
6	Incorrect pin number – error message	1	
7	Cancel/clear – clears PIN only	1	
8	Login with valid user and PIN logs to user screen	1	
9	Label displayed across the top of user screen	1	
10	User number is displayed	1	
11	User balance is displayed	1	
12	Log out button exists and returns user to login screen	1	
13	Log out button saves user details to user file with any changes	1	
14	Labels and buttons exist as required – 1 mark for each up to 5 marks	5	
15	Deposit – able to deposit money into user and adds to the balance	3	
16	Deposit – illegal value provides suitable error message box	1	
17	Add Entry – able to add an entry with a type of Rent, Spending, etc	3	
18	Add Entry – with invalid type provides error message box	1	
19	Add Entry – with insufficient funds provides error message box	1	
20	Add Entry – with invalid number provides error message box	1	
21	Transactions – displayed in Multiline Text widget	1	
22	Transactions – new valid transactions are added	2	
23	Multiline Text- has a scrollbar that works	1	
24	Graph of spending exists and is correct	5	

25	Graph of spending is updated when the balance is changed	5	
26	Coding is commented sufficiently	5	
27	Functionality only as described within assignment sheet	3	
	Assignment total (out of 50 marks)		
	Contribution to grade (out of 20 marks)		

Comments: