

ITECH1400 - Assignment 2 – FedUni Banking

Due Date: 5pm, Friday of Week 11

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 banking interface** similar to an ATM / online banking using the Python 3 programming language.

The assignment is broken up into five main components:

- 1.) The ability to **provide an account number and a PIN** (Personal Identification Number) to sign into a bank account,
- 2.) The ability to **view the balance of the bank account** and to **deposit and withdraw virtual money into and out from the account**,
- 3.) The ability to **save transactions via file storage** so that you can log in, deposit some money and then log out – and when you log back in that money is still there, and finally
- 4.) The ability to **display a graph of projected earnings** on the bank account via the **compound interest** accrued over a variable amount of time.
- 5.) A **Test Case** that ensures your BankAccount's deposit and withdraw functionality operates correctly.



Your submission should consist of three Python scripts that implement this application as described in the following pages: **bankaccount.py**, **main.py** along with a **testbankaccount.py** which contains a small test case with a few simple unit tests than ensure that your bank accounts **deposit_funds** and **withdraw_funds** methods operate correctly.

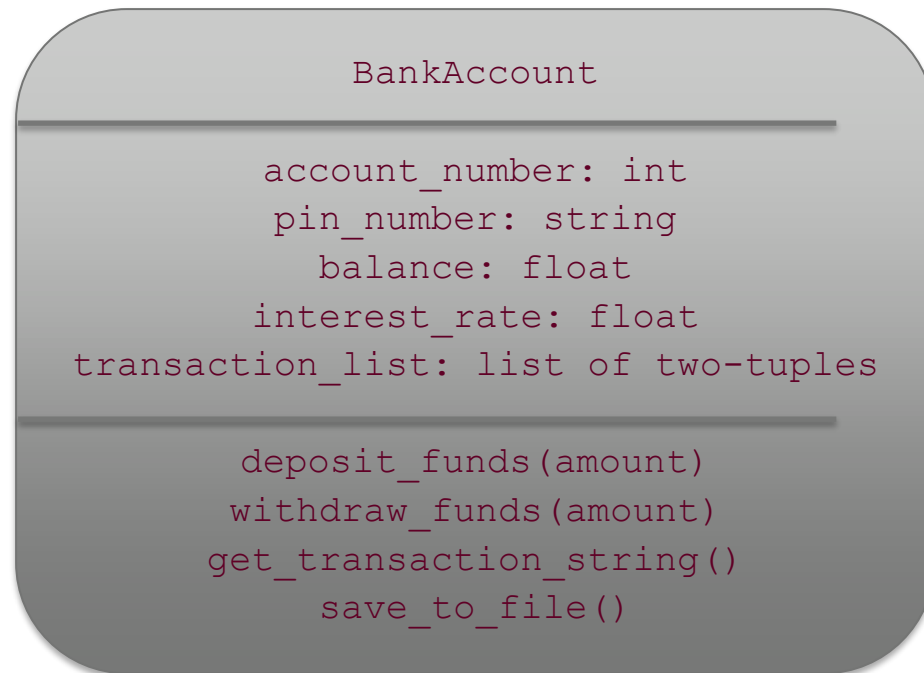
You are provided with a 'stub' of each of these files which contain all the function declarations and comments which describe the role of the function and how it can be put together, but you will have to write the code for vast majority of the functions yourself. You are also provided with a stub of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

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

There is no word processed component to this second assignment.

Bank Account Class Design

The design for a **BankAccount** object is laid out in the following class diagram:



As you might imagine, the **deposit_funds(amount)** function adds that money to the current **balance** of the account, and the **withdraw_funds(amount)** function removes (i.e. subtracts) money from the current **balance** of the account. Each transaction in the **transaction_list** is a **tuple** containing either the word *Deposit* or the word *Withdrawal* followed by an amount, for example: ("**Deposit**", 300.0) or ("**Withdrawal**", 100.0).

The bank accounts in our program do not have an overdraft facility so the user cannot withdraw money they do not have – that is, if they had \$200 in their account and they tried to withdraw more than \$200 then the operation should fail and the **withdraw_funds** function should raise an Exception 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 **messagebox**.

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 account.

The **save_to_file** function should save the **account_number**, **pin_number**, **balance**, and **interest_rate** in that order to a file called `<account_number>.txt` followed by the transaction list string generated from the **get_transaction_string()** method. The name of the account file is NOT '`<account_number>.txt`' - the name of the file is the *ACTUAL ACCOUNT NUMBER* followed by ".txt", so for an account with **account_number** 123456 the name of the account file would be **123456.txt**.

A full 'walk-through' video demonstrating the completed application and how it operates will be provided along with this assignment document.

Calculating Interest

To make it worthwhile for you to keep your money with a bank, the bank offers you an **interest rate** on your savings. Interest will be applied to the balance of an account once per month.

Let's do an example – suppose you had \$10,000 in a bank account and the bank paid you monthly interest at a rate of 3% per year. That would mean the bank pays you 3% of your balance divided by 12 (because there are 12 months in a year) per month. If we start our example on January and run it for a few months (and we don't deposit or withdraw any money throughout the year) then we end up with our bank balance changing like this:

Note: 3% divided by 12 is 0.25% per month – so we'll multiply our balance by 1.0025 to get the new balance.

Jan	Feb	Mar	Apr	May	Jun	Jul	Aug	Etc.
10,000.00	10,025.00	10,050.06	10,075.19	10,100.38	10,125.63	10,150.94	10,176.32	...

What's happening here is that the interest is **compounding** – which just means that we get that 0.25% applied not only to our *principle balance* (i.e. the \$10,000 we started with), but it also gets applied to the interest we earned. Although 3% interest is very low (but in line with the best rates you'd get in Australia at the moment because interest rates *are* very low), over time this compounding makes a serious difference!

Because FedUni Bank is the greatest bank of all time, it offers all accounts an interest rate of **33%**.

The Main Python Script

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

- Enter an account 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:
 - o See the balance of our account,
 - o Deposit funds into our account,
 - o Withdraw funds from our account (only up to the amount we have available),
 - o Display a plot of our projected interest over the next 12 months as outlined above, and finally
 - o Log out of our account.

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

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

```
account_number
account_pin
balance
interest_rate
```

For example, account number 123456 with PIN 7890 and a balance of \$800 with an interest rate of 33% would look like this:

```
123456
7890
800.00
0.33
```

After these first four 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 line. For example a deposit of \$500 would look like this:

```
Deposit
500.00
```

Similarly, a *withdrawal* is also specified as two lines – first the word **Withdrawal** and then on the next line the amount, for example a withdrawal of \$200 would look like this:

```
Withdrawal
200.00
```

You are provided with an example bank account 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 bank accounts in this format successfully.

You are also provided with a video demonstration of the completed assignment along with this document. Your application should match this user interface and function in the same way.

Login Screen

When the application is first launched, it should open a window that is "440x640" pixels in size (use the window object's **geometry** function to set this). Set the title of the window to "FedUni Banking" using the top-level window object's **wininfo_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 Banking" (font size is 32). On the next line is a label saying "Account Number" and then an **Entry** widget for the user to type in their account 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.

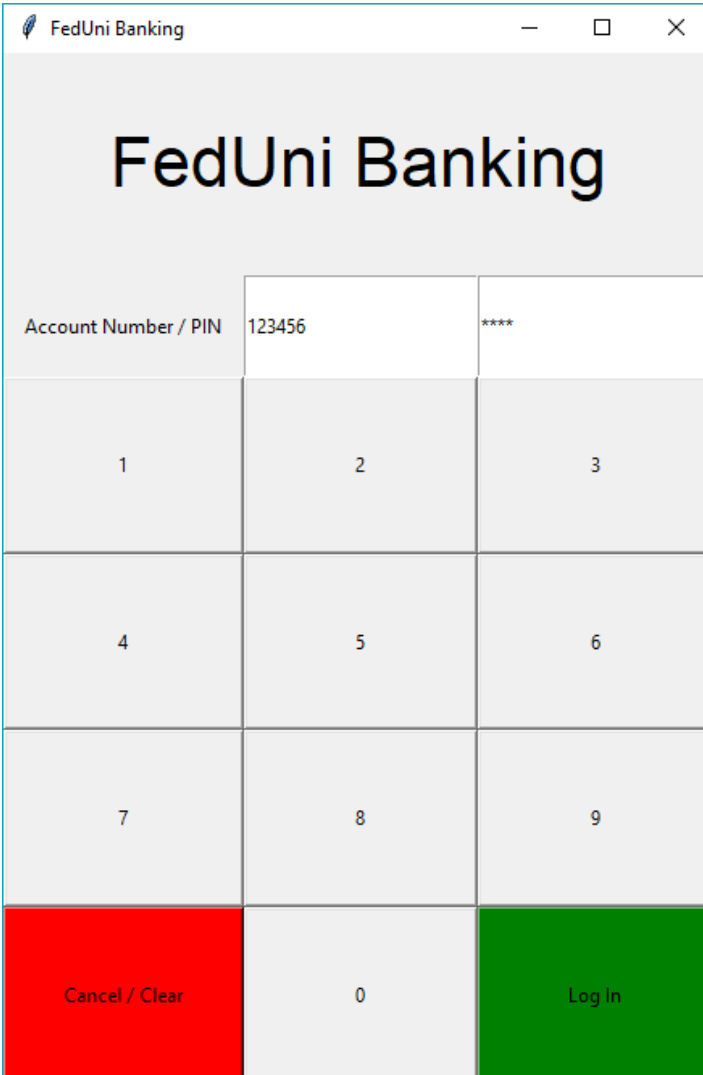
When the **Log In** button is pressed then the program should attempt to open the file with the account number followed by ".txt" - so in the example below, because the account number entered was "123456", the program will attempt to open the file "123456.txt".

If that file could not be opened then a messagebox should display a suitable error message such as *"Invalid account 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 account exists, then a `BankAccount` object should be created and the fields of the `BankAccount` object should be set (e.g. `account_number`, `pin_number`, `balance`, `interest_rate`, and the `transaction_list`).

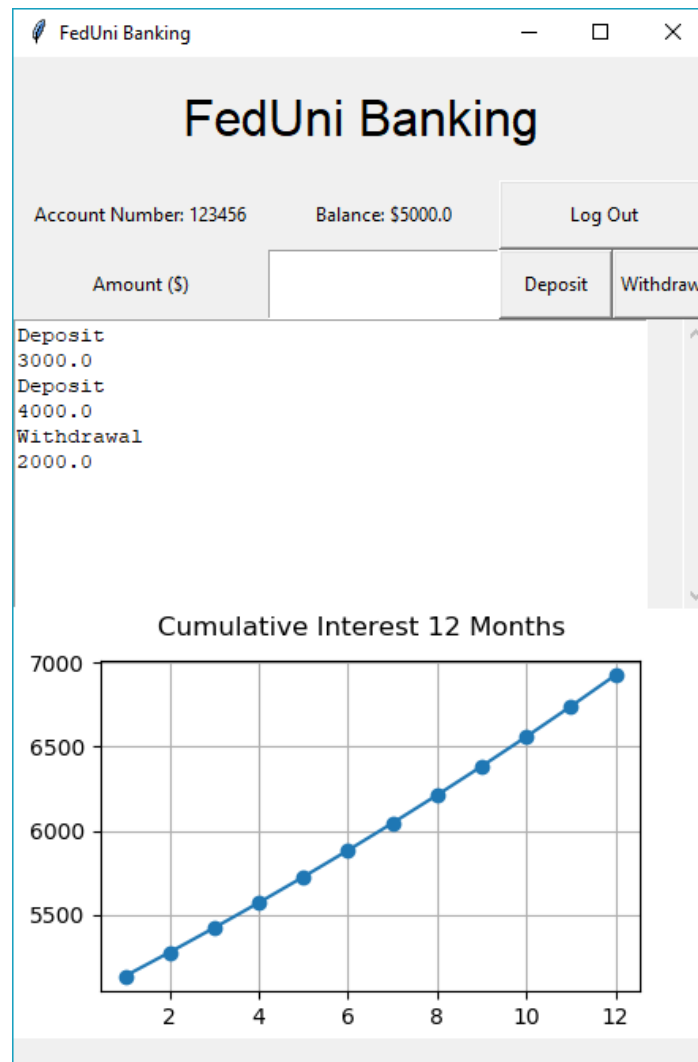
Because you don't know how many transactions are stored for this bank account, after reading the first four lines you will need to attempt to read two lines and if they exist create a tuple of the transaction (i.e. its type and amount) and then add it to the `BankAccount` object's `transaction_list` - 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_account_file()
if not line:
    break
else:
    # read another line, create a tuple from both lines and append the
    # tuple to the the BankAccount object's transaction_list
```



FedUni Banking		
Account Number / PIN	123456	****
1	2	3
4	5	6
7	8	9
Cancel / Clear	0	Log In

Account Screen



The account screen has:

- A large "FedUni Banking" label at the top that spans 5 columns (font size is 24),
- A label with the account number followed by the actual account number displayed,
- A label with the current balance followed by the actual balance,
- A log out button which saves the bank account 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 withdraw,
- Deposit and Withdraw buttons that deposit or withdraw funds using the BankAccount classes methods to do so,
- A **Text** widget (i.e. multi-line text) that shows all the transactions associated with the account. 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
- A small graph of the projected cumulative interest over the next 12 months.

Bank Account Test Case

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

You are provided with a **stub** of a **testbankaccount.py** test case that already has the definitions of the five 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 BankAccount classs' deposit and withdraw 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 account 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 account screen has 5 columns and 5 rows. The fifth column on the account screen is the one holding the scrollbar.

Once you have your login screen working, it makes sense to **set** the account 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 account 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 bank account 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 bank account 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 - so please stay on spec!

Submission and Marking Process

You must supply your program source code files as a single compressed archive called:

ITECH1400_Assignment_2_<YOUR-NAME>_<YOUR-STUDENT-ID>.zip

Obviously replace <YOUR-NAME> and <YOUR-STUDENT-ID> with your own personal details!

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

Submit your assignment to the Assignment 2 Upload location on Moodle before the deadline of Friday of week 11 at 5pm.

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 Banking

Student name:

Student ID:

Item	Assessment Criteria	Weight	Mark
1	Window is correct size of 440x640 with title of "FedUni Banking"	1	
2	"FedUni Banking" label is displayed in large font across top of login screen	1	
3	Account Number / PIN label exists near top left of login screen	1	
4	Clicking PIN entry buttons result in adding that number to pin entry	1	
5	All PIN number input is masked to be asterisks (i.e. **** not 7890)	1	
6	Incorrect account number results in suitable error message box	1	
7	Incorrect pin results in suitable error message box	1	
8	Cancel / Clear button clears PIN entry only	1	
9	Login button with valid account number and PIN logs in to account screen	1	
10	"FedUni Banking" label is displayed across top of account screen	1	
11	Account number is displayed on account number label	1	
12	Account balance is displayed on account balance label	1	
13	Log out button exists and returns user to login screen	1	
14	Log out button saves account details to account file with any changes made	1	
15	Amount label exists	1	
16	Amount entry exists	1	
17	Deposit button exists	1	
18	Withdraw button exists	1	
19	Clicking Deposit with legal value in amount entry adds to balance and add a suitable account transaction	2	
20	Clicking Deposit with illegal value results in suitable error message box	1	
21	Clicking Withdraw with legal value in amount entry subtracts from balance and adds a suitable account transaction	2	
22	Clicking Withdraw with illegal value results in suitable error message box	1	
23	Clicking Withdraw with insufficient funds results in suitable error message box	1	

24	Multiline Text widget showing account transactions exists	1	
25	Any new valid transaction made is displayed in the multiline Text widget	1	
26	Multiline Text widget has a scrollbar which can scroll the text	1	
27	Graph of interest exists and is correct for current balance showing next 12 months cumulative interest at bank account rate of 33% per annum	2	
28	Graph of interest is updated when balance changes	1	
29	BankAccount unit tests correctly test deposit / withdraw functionality	5	
	Assignment total (out of 36 marks)		
	Contribution to grade (out of 20 marks)		

Comments: