INFO1110 / COMP9001 Assignment 1

Money Tracker

Deadline: 23rd Sept 2019, 11:59pm AEST (Week 8, Monday)

Weighting: 10% of the final assessment mark

Overview

Brief Description

You will write a program that allows the user to manage their finances. The program will be able to record the user's incomes and expenses, display how their balance has changed, etc. It will also need to be able to handle regular incomes and expenses; for example, the user will be able to specify that they have a $100 income every Sunday, or that they spend $40.50 every Thursday.

Implementation details

Your program will be written in Python 3. The only modules you may import are `sys` and the `function.py` file which you will write yourself.

Submission

You will submit your code on the assignment page on Ed. To make a submission, you will need to press the "Mark" button. You may submit as many times as you wish without penalty - we will mark the **last** submission that you make. After each submission, the marking system will automatically check your code against the public test cases.

Please ensure you carefully follow the assignment specification. Your program output must exactly match the output shown in the examples.

Warning: Any attempts to deceive or disrupt the marking system will result in an immediate zero for the entire assignment. Negative marks can be assigned if you do not properly follow the assignment specifications, or your code is unnecessarily or deliberately obfuscated.

Help and feedback

You are encouraged to ask questions about the assignment during the Helpdesk and on the Ed discussion board; however, remember that **you should not be posting any assignment code publicly, as this would constitute academic dishonesty**.

The Program

Starting the Program

The program will be given 1 extra command line argument when it is run:

```
$ python3 tracker.py <filename>
```

This `<filename>` will specify a file with information about regular incomes and expenses; see the section on **Regular Transactions** for more information (it is recommended that you implement this feature last).

After handling this file, the program will ask the user for their starting balance, like so:

```
Starting balance: $
```

The user will then fill out this field with their initial balance, for example:

```
Starting balance: $4.11
```

- If the starting value cannot be converted to a float, the program should print `Error: Cannot convert to float!` and **quit immediately**.

  ```
  Starting balance: $cat
  Error: Cannot convert to float!
  ```

- If the starting value is negative or zero, the program should print `Error: Must start with positive balance!` and quit immediately.

  ```
  Starting balance: $-5
  Error: Must start with positive balance!
  ```

Once we have the regular payments and initial balance set up, we're good to go! The program should now continually ask for input, like so:

```
Enter command:
```

Depending on what the user enters, the program will record new transactions, show some statistics, etc. For example, if the user types `transaction`...

```
Enter command: transaction
```

...then the `transaction` operation (explained below) should execute. The program should continue asking for more inputs indefinitely, and execute the appropriate code each time.

Operations

`transaction`

Records a new income or expense transaction.

The program should ask the user how much money was involved in the transaction, like so:

```
Enter amount: $
```

The user should then be able to type in how much they earned or spent. Positive values represent income, and negative values represent expenses.

```
Enter amount: $5.95
```

```
Enter amount: $-2.35
```

This value should then be reflected in the user's balance; for example, if they originally had $5 but made a transaction of -$3, the user would now have $2.

However, if the value the user types in cannot be converted to a float, the program should print `Error: Cannot convert to float!` and ask for another command.

```
Enter amount: $cat
Error: Cannot convert to float!

Enter command:
```

`next`

Advances to the next day, making sure the user is not in debt.

If the user has strictly less than $0, the program should print `Oh no! You're in debt!` and quit immediately. Otherwise, it should print `Going to the next day...` and start a new day.

In this program, we start at **day 0** and increase the day number by 1 every time the `next` command is executed.

`status`

Displays a summary of how the day is going so far.

The program should print the following message:

```
Day <day number> (<weekday>)
Starting balance: $<starting balance>
Current balance: $<current balance>
```

Here,

- `<day number>` is the current day number of the program. As explained above, this starts at 0 and increases by 1 for every day that passes.
- `<weekday>` is which weekday it currently is, abbeviated to three letters; `Sun` for Sunday, `Mon` for Monday, etc. In this program, day 0 is always a Sunday, day 1 is a Monday, and so on. Day 6 would be a Saturday and day 7 is a Sunday again. (The `weekdays` list in the scaffold may be useful here.)
- `<starting balance>` is the amount of money we had at the **start** of the day, **before** any transactions (including regular transactions) were made. It should be displayed rounded to exactly 2 decimal places.
- `<current balance>` is the amount of money we **currently** have, **including** all the transactions we have made that day. It should be displayed rounded to exactly 2 decimal places.

If the current balance is greater than the starting balance, the program should also print `Nice work! You're in the black.`. Conversely, if the current balance is less than the starting balance, the program should print `Be careful! You're in the red.`. If the current balance is the same as the starting balance, no extra message needs to be printed.

Here are some examples:

```
Day 0 (Sun)
Starting balance: $5.00
Current balance: $6.00
Nice work! You're in the black.
```

```
Day 5 (Fri)
Starting balance: $55.00
Current balance: $54.50
Be careful! You're in the red.
```

```
Day 30 (Tue)
Starting balance: $120.00
Current balance: $120.00
```

`regular`

Displays a summary of the regular transactions.

Please see the section on "Regular Transactions" to see how these values will be specified.

The program should print `Regular Transactions:`; then, for every weekday, the program should show a summary of the regular transactions in the following format:

```
<weekday>: +$<regular income> -$<regular expense>
```

Here,

- `<weekday>` is the weekday abbreviated to three letters.
- `<regular income>` is the income the user regularly earns for that weekday. It should be displayed rounded to exactly 2 decimal places.
- `<regular expense>` is the expense the user regularly has for that weekday. It should be displayed rounded to exactly 2 decimal places.

Here is an example:

```
Regular Transactions:
Sun: +$5.00 -$4.00
Mon: +$3.00 -$2.00
Tue: +$7.35 -$1.00
Wed: +$0.00 -$0.00
Thu: +$14.90 -$5.00
Fri: +$6.00 -$3.00
Sat: +$6.00 -$5.00
```

`help`

Shows a list of availabile commands.

The program should print the following message:

```
The available commands are:
"transaction": Record a new income or expense
"next": Move on to the next day
"status": Show a summary of how you're doing today
"regular": Show a summary of your regular transactions
"help": Show this help message
"quit": Quit the program
```

`quit`

Quits.

The program should print `Bye!` and quit.

```
Enter command: quit
Bye!
```

Anything else

If any command other than those specified above is entered, the program should simply print the following message:

```
Command not found.
Use the "help" command for a list of available commands
```

The program should then continue prompting for commands.

Regular Transactions

Regular transactions are incomes and expenses that are automatically made on specific days of the week - for example, the user might always earn $100 on Sundays and spend $40.50 on Thursdays, which the program should automatically process. To implement these regular transactions, you need to follow these three steps:

Step 1: Write the function `process_file` in the `function.py` file. This function should take one **string** as an argument, open the file specified by this string and read the details of the regular transactions described in this file. You can assume that the file will always be in the following format, with exactly 14 lines and each line being convertible to a float without error:

```
<Sunday income>
<Sunday expense>
<Monday income>
<Monday expense>
...
<Friday income>
<Friday expense>
<Saturday income>
<Saturday expense>
```

All values in this file will be given as positive values; so if the user had a regular income of $100 on Sundays and a regular expense of $40.50 on Thursdays, our file would look like this:

```
100
0
0
0
0
0
0
0
0
40.50
0
0
0
0
```

The function should return a tuple of two lists, both of length 7; the first one being the regular **incomes** throughout the week, and the second one being the regular **expenses** throughout the week.

For example, if the string given to the function specified the file above, it should return `([100.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0], [0.0, 0.0, 0.0, 0.0, 40.5, 0.0, 0.0])`. Make sure your tuple contains lists of floats, not lists of strings.

However,

- If the file specified by the parameter **doesn't exist**, the function should raise a ValueError with the content of that error being the message `Error: String does not represent a valid file!`.

Step 2: At the start of your main code, use the `process_file` function to get the 2 lists containing the details of the regular incomes and expenses. The filename to be used as the parameter for this function will be specified as a command line argument when running the program.

```
$ python3 tracker.py <filename>
```

You will need to make sure you include the line `from function import process_file` at the top of your main code so that you can use the `process_file` function you wrote in part 1.

However,

- If the required command line argument is **not given**, the program should print `Error: Not enough command line arguments!` and quit immediately.
- If the `process_file` function **raises a ValueError**, the program should print `Error: File not found!` and quit immediately.

Step 3: Ensure your program automatically adds the regular income and subtracts the regular expenses for the appropriate weekday at the **start** of every day.

Examples

Regular Transaction Files

`no_regular_transactions.txt`:

```
0
0
0
0
0
0
0
0
0
0
0
0
0
0
```

`sample_regular_transactions.txt`:

```
5
4
3
2
7.35
1
0
0
14.9
5
6
3
6
5
```

Program Executions

Normal Case 1:

```
$ python3 tracker.py no_regular_transactions.txt
Starting balance: $100.50

Enter command: transaction
Enter amount: $10

Enter command: status
Day 0 (Sun)
Starting balance: $100.50
Current balance: $110.50
Nice work! You're in the black.

Enter command: next
Going to the next day...

Enter command: status
Day 1 (Mon)
Starting balance: $110.50
Current balance: $110.50

Enter command: regular
Regular Transactions:
Sun: +$0.00 -$0.00
Mon: +$0.00 -$0.00
Tue: +$0.00 -$0.00
Wed: +$0.00 -$0.00
Thu: +$0.00 -$0.00
Fri: +$0.00 -$0.00
Sat: +$0.00 -$0.00

Enter command: transaction
Enter amount: $-3

Enter command: transaction
Enter amount: $-10.2

Enter command: status
Day 1 (Mon)
Starting balance: $110.50
Current balance: $97.30
Be careful! You're in the red.

Enter command: quit
Bye!
```

Normal Case 2:

```
$ python3 tracker.py sample_regular_transactions.txt
Starting balance: $10.00

Enter command: next
Going to the next day...

Enter command: next
Going to the next day...

Enter command: next
Going to the next day...

Enter command: next
Going to the next day...

Enter command: status
Day 4 (Thu)
Starting balance: $18.35
Current balance: $28.25
Nice work! You're in the black.

Enter command: transaction
Enter amount: $100

Enter command: quit
Bye!
```

Invalid Command:

```
$ python3 tracker.py no_regular_transactions.txt
Starting balance: $5

Enter command: STATUS
Command not found.
Use the "help" command for a list of available commands

Enter command: exit
Command not found.
Use the "help" command for a list of available commands

Enter command: quit
Bye!
```

Going into Debt:

```
$ python3 tracker.py no_regular_transactions.txt
Starting balance: $5

Enter command: transaction
Enter amount: $-10

Enter command: status
Day 0 (Sun)
Starting balance: $5.00
Current balance: $-5.00
Be careful! You're in the red.

Enter command: transaction
Enter amount: $6

Enter command: status
Day 0 (Sun)
Starting balance: $5.00
Current balance: $1.00
Be careful! You're in the red.

Enter command: next
Going to the next day...

Enter command: transaction
Enter amount: $-10

Enter command: status
Day 1 (Mon)
Starting balance: $1.00
Current balance: $-9.00
Be careful! You're in the red.

Enter command: next
Oh no! You're in debt!
```

Non-existent File

```
$ python3 tracker.py nonexistent.txt
Error: File not found!
```

Unable to Interpret Transaction as Float

```
$ python3 tracker.py no_regular_transactions.txt
Starting balance: $10.50

Enter command: transaction
Enter amount: $cat
Error: Cannot convert to float!

Enter command: quit
Bye!
```

Hints and Extra Information

- You should always print an empty line directly before asking the user for a command.
- Commands are case-sensitive. For example, `quit` is valid, but `QUIT` is invalid.
- To print a string containing single quotes, enclose it in double quotes, and vice versa (or put a backslash before it).
- Use the modulo operator (`%`) to your advantage. It can help you keep track of weekdays!
- For the purposes of this assignment, you can treat all monetary values as floats - none of the test cases will cause floating point errors.

Marking

- 7 marks will be given for passing the automated test cases. There are a total of 28 tests, and you will receive 0.5 marks for every 2 that you pass (quarter marks will **not** be given).
- 3 marks will be given for a manual inspection of code. You should ensure that your code uses meaningful variable names, good structure, and helpful and concise comments to clarify the intentions of the code.

Late Submissions

Late submissions have a hard cap of 25% of the total possible marks per day late. For example, if your submission is 1 hour late, you can only receive a maximum of 75% for this assessment. If your submission is 48 hours late, you can only receive a maximum of 50% for this assessment.

Do not make any submissions after the deadline unless you want it to be counted as late. Remember, we will be marking the last submission you make.

Academic declaration

By submitting this assignment, you declare the following:

I declare that I have read and understood the University of Sydney Student Plagiarism: Coursework Policy and Procedure, and except where specifically acknowledged, the work contained in this assignment/project is my own work, and has not been copied from other sources or been previously submitted for award or assessment.

I understand that failure to comply with the Student Plagiarism: Coursework Policy and Procedure can lead to severe penalties as outlined under Chapter 8 of the University of Sydney By-Law 1999 (as amended). These penalties may be imposed in cases where any significant portion of my submitted work has been copied without proper acknowledgment from other sources, including published works, the Internet, existing programs, the work of other students, or work previously submitted for other awards or assessments.

I realise that I may be asked to identify those portions of the work contributed by me and required to demonstrate my knowledge of the relevant material by answering oral questions or by undertaking supplementary work, either written or in the laboratory, in order to arrive at the final assessment mark.

I acknowledge that the School of Computer Science, in assessing this assignment, may reproduce it entirely, may provide a copy to another member of faculty, and/or communicate a copy of this assignment to a plagiarism checking service or in-house computer program, and that a copy of the assignment may be maintained by the service or the School of Computer Science for the purpose of future plagiarism checking.