Python Exam: Expense Management System

You must submit the completed program as per instructions provided on Canvas.

You CANNOT:

- Work, collaborate or share code with someone else on this exam.
- Search for keywords leading to solutions on the exam. For example, do not Google "loading a .csv file in Python".
- Discuss the exam with other students. Other students have not yet taken the exam, and you CANNOT provide academically dishonest assistance to them.

You CAN:

- Reference any material from the course or recitation. This includes videos, slides, code samples, homework assignments, quizzes, recitation recordings and coding exercises.
- Reference any online Python documentation.
- Use the Canvas search bar for quick syntax and content searches.

Use of Ed Discussion/Office Hours as it Relates to the Exam

- You CAN ask any question on Ed Discussion or during office hours that directly relates to the exam logistics and/or technical errors.
- You CANNOT ask a question specifically related to code or "how to do something". For example, you cannot ask "How should I go about writing the code for this function?". You also cannot post code and ask "Can you help me understand why my code isn't working?"

The Assignment

This Python exam will involve implementing a *system for managing expenses*. You will download the skeleton of the program, then implement the functions/methods. The design of the program has been set up for you.

In this system, users will be able to add and deduct expenses, update expenses, sort expenses, and export filtered expenses to a file. The program will initially load a collection of expenses

from 2 different .txt files (in the same format) and store them in a dictionary.

Steps for Completing the Exam

1. Complete all of the required methods
    a. Implement all of the methods defined in ***ExpensesLoader.py***, ***ExpensesManager.py***, and ***Expense.py***
    b. Docstrings have already been provided
    c. Add comments to your code
    d. You can create any number of helper methods (with docstrings).
    e. The *main* function in ***expenses.py*** has already been implemented for you. DO NOT CHANGE THE CODE IN MAIN.
2. Test your code by running (and passing) all of the provided test cases in the given ***expenses_test.py***.
    a. **Write additional test cases as noted** and make sure they pass as expected. Your test cases should be distinct.
3. Make sure your entire program and the unit testing file run without errors!
    a. ***expenses.py*** is the main program file. Run this file to run your entire program.

Required Methods

Below you will find explanations of the methods that need to be written in ***ExpensesLoader.py***, ***ExpensesManager.py***, and ***Expense.py***. We are expecting to see these methods with these names and signatures exactly. Do not change the names of these methods as we will be running automated tests against each individual method. You will fail the autograded tests if you change the method names or signatures. You will also fail each test unless you remove the `raise NotImplementedError` line in each method.

Be sure to add comments to your code.

In ***ExpensesLoader.py***:

def import_expenses(self, expenses, file):

- Reads data from the given file and stores the expenses in the given expenses dictionary, where the expense type is the key and the value is an Expense object with the parameters expense type and total amount for that expense type.
- The same expense type may appear multiple times in the given file, so add all the amounts for the same expense types.
- Ignore expenses with missing amounts. If a line contains both an expense type and an

expense amount, they will be separated by a colon (:).
- You can assume that if they exist, expense types are one-word strings and the amounts are numerical and can be casted to a float data type.
- Strip any whitespace before or after the expense types and amounts.
- Blank lines should be ignored.
- Expenses are case sensitive. "coffee" is a different expense from "Coffee"
- This method will be called twice in the *main* function in ***expenses.py*** with the same dictionary but different files.
- This method doesn't return anything. Rather, it updates the given expenses dictionary based on the expenses in the given file.
- For example, after loading the expenses from the file, the expenses dictionary should look like this: {'food': Expense('food', 5.00), 'coffee': Expense('coffee', 12.40), 'rent': Expense('rent', 825.00), 'clothes': Expense('clothes', 45.00), 'entertainment': Expense('entertainment', 135.62), 'music': Expense('music', 324.00), family': Expense('family', 32.45)}

In ***ExpensesManager.py***:

def get_expense(self, expenses, expense_type):

- Returns the Expense object for the given expense type in the given expenses dictionary.
- Prints a friendly message and returns `None` if the expense type doesn't exist.
- (Note: Printing a friendly message means that the program should not raise an error or otherwise terminate. Simply tell the user that the requested expense type does not exist and continue the program.
- Also note that `None` is a specific keyword in Python of `NoneType`. You should not return a string "None" from this method.)

def add_expense(self, expenses, expense_type, value):

- If the expense_type already exists in the given expenses dictionary, add the value to the associated Expense object amount.
- Otherwise, create a new entry in the expenses dictionary with the expense_type as the key and the value as an Expense object with the given expense_type and value parameters.
- Prints the updated Expense object.
- This method doesn't return anything.

def deduct_expense(self, expenses, expense_type, value):

- From the given expenses dictionary, get the Expense object associated with the given expense_type and deduct the given value from the amount.
- Raises a `RuntimeError` if the given value is greater than the existing amount of the Expense object. Note: You are not supposed to use try/except to catch the `RuntimeError` you raised. We expect the method to raise a `RuntimeError` if the value is greater than the existing total of the expense type.
- Prints a friendly message if the expense type doesn't exist from the given expenses dictionary. (Note: Printing a friendly message means that the program should not raise an error or otherwise terminate. Simply tell the user that the requested expense type does not exist and continue the program.)
- Print the updated Expense object if `RuntimeError` is not raised.
- This method doesn't return anything.

def update_expense(self, expenses, expense_type, value):

- From the given expenses dictionary, update the Expense object associated with the given expense_type and use the given value.
- Prints a friendly message if the expense type doesn't exist. Note: Printing a friendly message means that the program should not raise an error or otherwise terminate. Simply tell the user that the requested expense type does not exist and continue the program.
- Prints the updated Expense object if it exists.
- This method doesn't return anything.

def sort_expenses(self, expenses, sorting):

- Converts the key:value pairs in the given expenses dictionary to a list of tuples containing the expense type and amount of the Expense object (Expense.expense_type, Expense.amount) and sorts based on the given sorting argument.
- If the sorting argument is the string 'expense_type', sorts the list of tuples based on the expense type (e.g. 'rent') in ascending alphabetical order of the expense_type, e.g. sorted results: ("coffee", 5.0), ("food", 5000.0), ("rent", 1000.0)
- Otherwise, if the sorting argument is 'amount', sorts the list of tuples based on the total expense amount (e.g. 825.0) in descending order of the expense amount, e.g. sorted results: ("food", 5000.0), ("rent", 1000.0), ("coffee", 5.0)
- Returns the list of sorted tuples. (Note: If the given sorting argument is not an acceptable value (e.g. 'expense_type' or 'amount'), this method does nothing except print a friendly message and return `None`.)

def export_expenses(self, expenses, expense_types, file):

- Exports the Expense objects associated with the given expense_types from the given expenses dictionary to the given file.
- Do not append to the file. If the function is called again and the given file already exists, make sure it overwrites what was previously in the file instead of appending to it.
- Iterates over the given expenses dictionary, filters based on the given expense types (a list of strings), and exports to a file. Skips any expense type in the given list of expense types that doesn't exist.
- If the expenses argument is the dictionary {"food": Expense("food", 5000.00), "rent": Expense("rent", 1000.00), "coffee": Expense("coffee", 5.00), "clothes": Expense("clothes", 58.92)} and the expense_types argument is the list of strings ['coffee', 'clothes', 'rent'], exports a file containing:
```
coffee: 5.00
clothes: 58.92
rent: 1000.00
```
- If the expenses argument is the dictionary {"food": Expense("food", 5000.00), "rent": Expense("rent", 1000.00), "coffee": Expense("coffee", 5.00), "clothes": Expense("clothes", 58.92)} and the expense_types argument is the list of strings ['coffee', 'clothes', 'sports'], exports a file containing:
```
coffee: 5.00
clothes: 58.92
```
- Note, the specified expense type 'sports' does not exist in the expenses dictionary, so it is ignored.
- If an item is duplicated in the given expense types, don't worry about it, just export the data as is. You should not deduplicate the expense types.
- This method doesn't return anything.

In ***Expense.py***:

def add_amount(self, amount):

- Adds the given amount to the total amount of the expense.
- This method doesn't return anything.

def deduct_amount(self, amount):

- Deducts the given amount from the total amount of the expense.
- This method doesn't return anything.

We have provided you with the following methods. DO NOT MODIFY these:

def __init__(self, expense_type, amount):

- Initialize expense_type with given expense_type and amount

def __str__(self):

- Returns string representation of expense, aimed at the user. Called by str(object) and the built-in functions format() and print() to compute an "informal" or nicely printable string representation. Returns expense type and amount, rounded to 2 decimal places.

def __repr__(self):

- Returns string representation of expense, aimed at the programmer. Typically used for debugging, to provide an information-rich and unambiguous string representation. Returns expense type and amount, rounded to 2 decimal places.

Unit Testing

To test your code, **we have provided you with a SUBSET of the unit tests for this assignment in *expenses_test.py***. When we grade, we will run additional tests against your program. **Passing the pre-submission tests does not guarantee that you will pass the post-submission tests.**

Expected Output

We've provided the 'template_behaviour.txt' file to show the expected behavior of the expense management program while it's running. For example, in the scenario below, entering "1" will allow the user to get the information for a particular expense. Entering "coffee" will show 12.40, the total for that expense. You are not required to round any numbers.

```
Welcome to the expense management system!  What would you like to do?
1: Get expense info
2: Add an expense
3: Deduct an expense
4: Update an expense
5: Sort expenses
6: Export expenses
0: Exit the system

1
Expense type? coffee
coffee: 12.40
```

For another example, in the scenario below, entering "2" will allow the user to add an amount to an existing expense. Entering "coffee" and 1.32 will add to that expense, and show 13.72, the new total for that expense.

```
Welcome to the expense management system!  What would you like to do?
1: Get expense info
2: Add an expense
3: Deduct an expense
4: Update an expense
5: Sort expenses
6: Export expenses
0: Exit the system

2
Expense type? coffee
Amount to add? 1.32
Expense: 13.72
```

What to Submit

Your submission should include the following files:

1. ***expenses.py (DO NOT MODIFY)***
2. ***ExpensesLoader.py***, ***ExpensesManager.py***, and ***Expense.py*** with all the required implementation.
3. ***expenses_test.py***: the unit testing file
4. ***expenses.txt*** and ***expenses_2.txt***: the .txt files to be read by your program
    a. It is important that you DO NOT edit these files. If you do, you could fail the automated testing.

b. DO NOT change the spacing or remove any blank lines.
c. DO NOT copy/paste the text from these files into other files.

Evaluation

1. Did you implement the individual methods correctly in ***ExpensesLoader.py***, ***ExpensesManager.py***, and ***Expense.py***? – *65 points*
    a. Does your program successfully load and parse the .txt files and store all of the expenses in a dictionary database?
    b. Can you get expense information from the system?
    c. Can you add expenses to the system?
    d. Can you update and deduct from expenses?
    e. Can you sort expenses by expense type and amount?
    f. Can you export expenses to a file?
    g. Does your program raise a `RuntimeError` if you try to deduct an invalid amount?
2. Unit Testing – *30 points*
    a. Does your program pass all of the provided unit tests in ***expenses_test.py***?
    b. Did you write the additional required test cases for each function in ***expenses_test.py***? Did you test both typical examples **and edge cases**?
3. Coding Style – *5 points*
    a. Appropriate naming of variables
    b. Naming of helper methods (with docstrings)
    c. Clear comments in your code