# Homework 5 : Banking System

HW deadline as per Canvas.

This homework deals with the following topics:

- Dictionaries
- File I/O
- Unit testing
- Raising a Runtime Error (Reference the provided "Raising a RuntimeError" reading)

## The Assignment

This assignment will involve implementing a bank program that manages bank accounts and allows for deposits, withdrawals, purchases, sorting bank accounts based on different criteria, and exporting bank statements.

The program will initially load a list of accounts from a .txt file, and deposits and withdrawals from additional .csv files. Then it will parse and combine all of the data and store it in a dictionary.

## Steps

1. Write your program and implement the required functions
    a. Create your program in ***bank_accounts.py***
    b. Implement all of the functions defined in "Functions to implement" below
        i. Be sure to add docstrings to your functions
        ii. You can create any number of helper functions (with docstrings)
        iii. Add brief comments to all non-trivial lines of code
2. Test your code by running (and passing) all of the provided test cases in the given ***bank_accounts_tests.py***
    a. **Write additional test cases as noted in each test function** and make sure they pass as expected. Your test cases should be distinct.
3. Make sure your program and the testing file run without errors!

## Functions to Implement

Be sure to add docstrings to all your functions and comments to your code.

*init_bank_accounts(accounts, deposits, withdrawals):*

- Loads the given 3 files, stores the information for individual bank accounts in a dictionary and calculates the account balance
- *accounts* .txt file contains information about bank accounts
  - Each row contains an account number, a first name, and a last name, separated by vertical pipe (|)
  - Example:
    `1|Brandon|Krakowsky`
- *deposits* .csv file contains a list of deposits for a given account number
  - Each row contains an account number, and a list of deposit amounts, separated by a comma (,)
  - Example:
    `1,234.5,6352.89,1,97.60`
- *withdrawals* .csv file contains a list of withdrawals for a given account number
  - Each row contains an account number, and a list of withdrawal amounts, separated by a comma (,)
  - Example:
    `1,56.3,72.1`
- Stores all of the account information in a dictionary named 'bank_accounts', where the account number is the key, and the value is a nested dictionary. The keys in the nested dictionary are first_name, last_name, and balance, with the corresponding values.
  - Example:
    ```
    {'1': {'first_name': 'Brandon', 'last_name': 'Krakowsky',
    'balance': 6557.59}}
    ```
- This function calculates the total balance for each account by taking the total deposit amount and subtracting the total withdrawal amount
- Returns the 'bank_accounts' dictionary

*get_account_info(bank_accounts, account_number):*

- Returns the account information for the given *account_number* as a dictionary
- Example:
  ```
  {'first_name': 'Brandon', 'last_name': 'Krakowsky', 'balance':
  6557.59}
  ```
- If the account doesn't exist, returns None

*withdraw(bank_accounts, account_number, amount):*

- Withdraws the given amount from the account with the given *account_number*
- Rounds the new balance to 2 decimal places (Uses *round_balance()* function)

- If the account doesn't exist, prints a friendly message
- Raises a Runtime Error if the given amount is greater than the available balance
- Prints the new balance

*deposit(bank_accounts, account_number, amount):*

- Deposits the given amount into the account with the given account_number
- Rounds the new balance to 2 decimal places (Uses *round_balance()* function)
- If the account doesn't exist, prints a friendly message
- Prints the new balance

*purchase(bank_accounts, account_number, amounts):*

- Makes a purchase with the total of the given amounts from the account with the given *account_number*
- *amounts* is a list of floats
- If the account doesn't exist, prints a friendly message
- Calculates the total purchase amount based on the sum of the given amounts, plus (6%) sales tax (Uses *calculate_sales_tax()* function)
- Raises a Runtime Error if the total purchase amount is greater than the available balance
- Prints the new balance

*sort_accounts(bank_accounts, sort_type, sort_direction):*

- Converts the key:value pairs in the given *bank_accounts* dictionary to a list of tuples and sorts based on the given *sort_type* and *sort_direction*
- Returns the sorted list of tuples
- If the *sort_type* argument is the string ‘account_number’, sorts the list of tuples based on the account number (e.g. ‘3’, '5') in the given *sort_direction* (e.g. 'asc', 'desc')
- Example sorted results based on 'account_number' in ascending order:
```
('1', {'first_name': 'Brandon', 'last_name': 'Krakowsky',
'balance': 6557.59}),
('2', {'first_name': 'Chenyun', 'last_name': 'Wei', 'balance':
4716.89})
('3', {'first_name': 'Dingyi', 'last_name': 'Shen', 'balance':
4.14})
```
- Otherwise, if the sort_type argument is 'first_name', 'last_name', or 'balance', sorts the list based on the associated values (e.g. 'Brandon', 'Krakowsky', or 6557.59) in the given *sort_direction* (e.g. 'asc', 'desc')
- Example sorted results based on 'balance' in descending order:

```
('6', {'first_name': 'Karishma', 'last_name': 'Jain', 'balance':
6700.19}),
('1', {'first_name': 'Brandon', 'last_name': 'Krakowsky',
'balance': 6557.59}),
('2', {'first_name': 'Chenyun', 'last_name': 'Wei', 'balance':
4716.89})
```

- Example sorted results based on 'last_name' in ascending order:

```
('4', {'first_name': 'Zhe', 'last_name': 'Cai', 'balance':
114.31}),
('9', {'first_name': 'Ruijie', 'last_name': 'Cao', 'balance':
651.44}),
('7', {'first_name': 'Huize', 'last_name': 'Huang', 'balance':
0})
```

- Example sorted results based on 'first_name' in descending order:

```
('4', {'first_name': 'Zhe', 'last_name': 'Cai', 'balance':
114.31}),
('10', {'first_name': 'Tianshi', 'last_name': 'Wang', 'balance':
0.0})
('9', {'first_name': 'Ruijie', 'last_name': 'Cao', 'balance':
651.44}),
('8', {'first_name': 'Paranya', 'last_name': 'Jareonvongrayab',
'balance': 326.5})
```

- If given *sort_type* is not an acceptable value (e.g. 'account_number', 'first_name', 'last_name', 'balance'), this function does nothing except print a friendly message and return None
- If given *sort_direction* is not an acceptable value (e.g. 'asc', 'desc'), assume the *sort_direction* is 'desc'

*export_statement(bank_accounts, account_number, output_file):*

- Exports the given account information to the given output file in the following format:

```
First Name: Huize
Last Name: Huang
Balance: 34.57
```

- If the account doesn't exist, print a friendly message and do nothing

*round_balance(bank_accounts, account_number):*

- Rounds the account balance of the given *account_number* to two decimal places
    - Note, this function actually updates the account balance in the *bank_accounts* dictionary

*calculate_sales_tax(amount):*

- Calculates and returns a 6% sales tax for the given *amount*

*main():*

- Loads and gets all account info by calling *init_bank_accounts()* function
  ```
  bank_accounts = init_bank_accounts('accounts.txt',
  'deposits.csv', 'withdrawals.csv')
  ```
- While the program is running
  - Prints welcome message and input options for the user
    - Input ‘1’ to get account info
    - Input ‘2’ to make a deposit
    - Input ‘3’ to make a withdrawal
    - Input ‘4’ to make a purchase
    - Input ‘5’ to sort accounts
    - Input ‘6’ to export a statement
    - Input ‘0’ to leave the banking program
  - Prompts for user input and tries to cast to an int
    - If the input is invalid, prints a friendly message and re-prints the options above for the user
  - If the input is ‘1’
    - Prompts for account number, calls *get_account_info()*, and prints account info
  - If the input is ‘2’
    - Prompts for account number, prompts for amount to deposit, and calls *deposit()* to make a deposit
    - If the deposit amount is invalid (non-numeric), prints a friendly message and does nothing
  - If the input is ‘3’
    - Prompts for account number, prompts for amount to withdraw, and calls *withdraw()* to make a withdrawal
    - If the withdrawal amount is invalid (non-numeric), prints a friendly message and does nothing
  - If the input is ‘4’
    - Prompts for account number and prompts for purchase amounts as comma separated list
    - Converts purchase amounts to list of floats
    - Calls *purchase()* to make the purchase
    - If purchase amounts are invalid (non-numeric), prints a friendly message and does nothing

- If the input is ‘5’
  - Prompts for sort type as 'account_number', 'first_name', 'last_name', or 'balance'
  - Prompts for sort direction as 'asc' or 'desc'
  - Calls and prints result of *sort_accounts()*
- If the input is ‘6’
  - Prompts for account number and calls *export_statement()* to export a .txt file
- If the input is ‘0’
  - Prints a friendly message and leaves the banking program

Make sure to add the following code to the bottom of your program to launch the *main()* function:

```
if __name__ == "__main__":

    main()
```

## Unit Testing

To test your code, **we have provided you with a SUBSET OF ALL of the unit tests for this assignment**. When we grade, we will run additional tests against your program.

## Program Output

Print out what the program is doing as it goes along. We have provided a *sample_behaviors.txt* file which shows some sample runs of the program -- yours should provide similar information.

## Evaluation

1. Does your program work as expected? - 10 points
   a. Can you make a deposit to or withdrawal from one of the accounts in the .txt file? Can you make multiple purchases at once? Can you export a correctly formatted bank statement? Can you sort all bank accounts based on user provided criteria?
   b. Does your program print clear and useful error messages when applicable? For

example, if you try to withdraw from a non-existent account.

2. Did you implement the functions correctly? - 10 points
    a. Does your program successfully load and parse the 3 files: accounts, deposits, and withdrawals, and store all the data in a dictionary database?
    b. Are you reusing functions in multiple places in your program? For example, the “purchase” function can use the “withdraw” function.  Several functions can use the “get_account_info” function.
3. Unit testing - 10 points
    a. Did you pass all of the provided unit tests?
    b. Did you write at least 2 additional test cases for each function, where noted?
    c. Do you test both typical examples and edge cases?
    d. Does your program pass all of your own tests?
4. Style - 4 points
    a. Appropriate naming of variables
    b. Naming of helper functions (with docstrings)
    c. Clear comments in your code

## Submission

Submit ***bank_acounts.py***, ***bank_accounts_tests.py***, and any other data files you used for testing.