

Problem Set 6: Banking Wrap-Up

[Submit Assignment](#)

Due Apr 30 by 11:59pm

Points 0

Submitting a file upload

File Types py

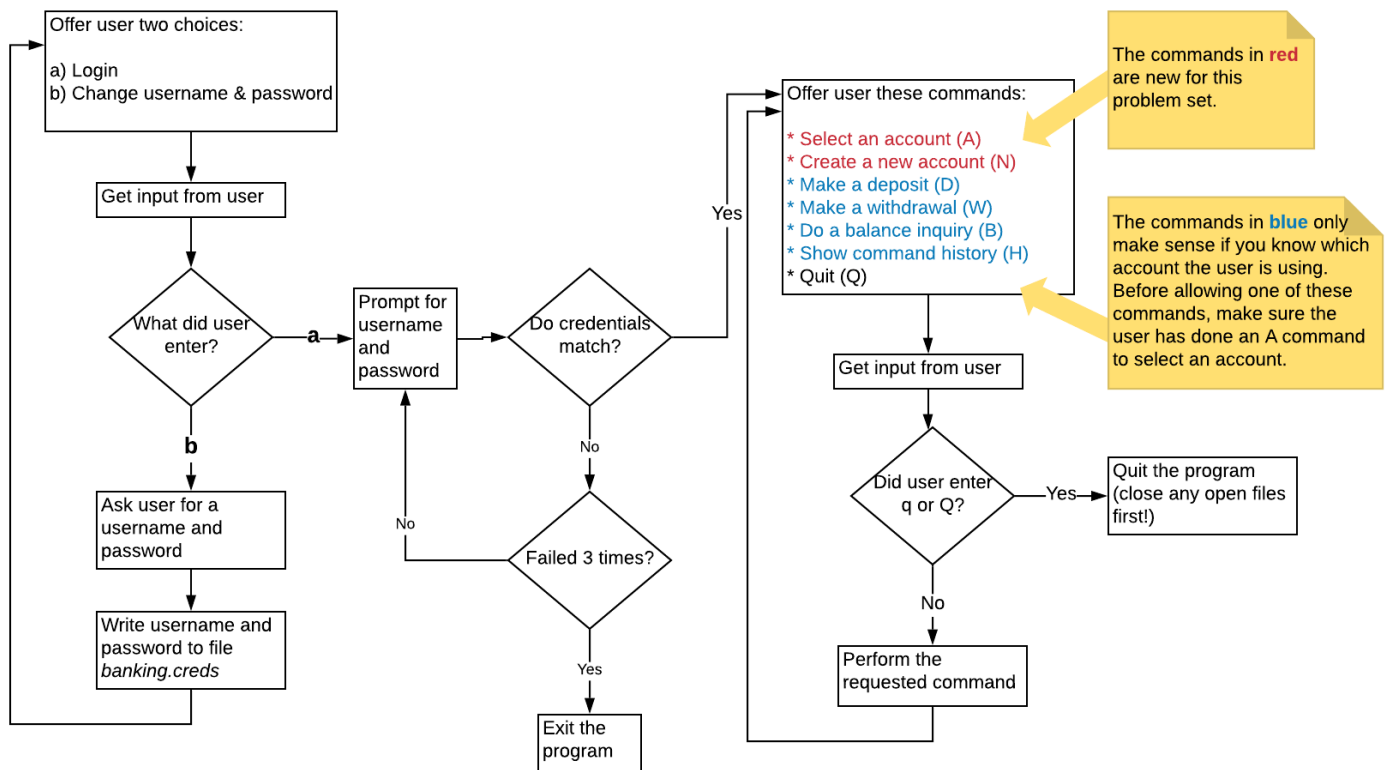
Problem Set 6: Banking Wrap-Up

For this problem set, start with the code you submitted for Problem Set 5. Copy the file and rename it for PS6.

You have a nicely functioning bank account application at this point, but it doesn't save any data. Every time you run it, it starts with a zero balance and an empty command history. In this problem set, you'll save your data to a file. You'll also expand the application's capabilities so that a user can:

- change the username and password
- open new accounts
- select an account to work with after logging in

The instructions for this problem set are complicated. This flowchart might help you understand what is being requested. Once you complete this problem set, your code should behave as depicted in the flowchart.



Step 1: Modify your code to read the credentials from a file

Your code is currently using a hardcoded username and password. What you're going to do now is eliminate the hardcoded credentials and instead read them in from a file.

In the same directory as your code, create a new file named *banking.creds*. You can use PyCharm for this -- just create a new "File." On the first line of the file, type the username you have been using. Hit Enter, and on the second line, type the password you have been using. Save the file.

Now modify your code as follows:

- Delete the hardcoded username and password from your code
- Before prompting the user to enter a username and password, open *banking.creds* and read in the username and password. The rest of the login logic is unchanged, i.e., prompt the user for credentials, compare them to what you read in from the file, etc.

Step 2: Let the user change the username and password

Currently, when you run your Python code, the first thing it does is prompt the user to login. Modify your code so that it matches the flowchart above. Instead of immediately prompting the user to log in, give the user a menu of two choices:

1. Login
2. Change username/password

If the user chooses to login, your code behaves the same as before. If the user chooses to change the username and password, your code will now have to:

- Prompt for a new username and password
- Write the username and password to the *banking.creds* file. The file will already exist because you made it manually in the last step. This is fine; just overwrite it with the new credentials. **Do not add/append to the file -- completely overwrite it so that the new credentials replace the old ones.**


Step 3: Add the ability to open a new account

So far, your application has consisted of one account, and that account didn't even have an account number. Typically, however, bank accounts have numbers. In this step, you will give your application the ability to create a new, numbered account.

Step 3a -- Write a *new_account* function

Write a function named *new_account*. Your function:

- Takes no arguments
- Should randomly generate an account number greater than 9,999 and less than 100,000 (so, a five-digit number that does not start with a zero).
- Should create a new data file to hold the balance and transaction history for this account. Keep reading to see how to do this.

At this point, the account is brand new, so it has a balance of \$0, and there is no transaction history yet. You need to write these details (the balance and the empty transaction history) to a data file that is specific to this account. However, you *do not* have to write code to create and manage the data file! Instead, download [*accountio.py*](#)  (in Canvas under Files->code snippets) and use the *write_account* function in that file to get the job done.

You will see that *accountio.py* contains two functions: *write_account* and *read_account*. When you call *write_account* with the proper arguments, it will create an account data file for you, and it will write the account's balance and transaction history to the file (you will supply those as arguments to the function). The data file that *write_account* creates will be named *account_NNNNNN.data*, where *NNNNNN* will be the 5-digit account number.

Look at the code in *accountio.py* and read the comments -- you will be able to figure out how to use *write_account* to create and write to the data file.

Note: DO NOT (DO NOT, DO NOT, DO NOT) copy/paste the code from *accountio.py* into your code. You must instead put *accountio.py* in the same directory as your code and use a Python *import* statement to import its two functions.

- Should return the account number that you generated randomly.

Step 3b -- Add the N/n command

Modify your command loop to add a new command, *n* or *N*, to create a new account. If the user selects this command:

- Call your *new_account* function
- Print a message telling the user the newly generated account number

Important Note: The user can create multiple accounts by doing the *n/N* command more than once. Each account will get its own account number and its own data file, but *there is only one set of login credentials for the whole project*. There is just one *banking.creds* file that holds the username and password. Each numbered account *does not* get its own username and password.

The idea to have multiple bank accounts isn't new or specific to this problem set -- people commonly have both a checking and savings account, for example. Each account has its own account number, but to use those accounts at an ATM or through online banking, you don't have a separate username/password or PIN for each account. Rather, one set of credentials lets you in, and once you have logged in, you can access your various accounts.

Step 4: Make your code aware of account numbers

Now that you can create a numbered account, every time you perform a command, your application needs to know which account you mean to use. One way to accomplish this is to add a command to the command loop for selecting an account. Let's do this with a new command, *a* or *A*. This command should:

- Prompt the user to enter an account number
- Use *read_account* (this is one of the functions in *accountio.py*) to read in the data for the given account. Look at the code and comments for *read_account* in the *accountio.py* file to figure out how to use this function.

Note that if the user enters an invalid account number, the account data file won't exist, so *read_account* will raise an exception. If this happens, print an error message and let the command loop continue. Once you get a valid account number from the user, ***all future commands should apply to this account until the user changes accounts with another a/A command***. If the user wants to change accounts and work with a different one, he/she can enter another *a/A* command and select a different account.

There is an important implication of the move to numbered accounts: it won't make sense to do a *D*, *W*, *B*, or *H* command unless you know which account the user wants to work with. The easy way to handle this is with an error message; if the user tries a *D*, *W*, *B*, or *H* command before selecting an account, print a message, e.g., "Select an account first". Then let the command loop continue as usual.

Note: You **do not** have to restructure your code to force the user to do an *A/a* command as a first step; that's too much work!

Note: You will need to invent some way of knowing if the user has selected an account to work with.

Step 5: Save the account data

So far, when you created a new numbered account, you used *write_account* to start the account's data file. At that point, the account was new and had a zero balance. Now, you need to keep the data file up to date after every command that can change the account.

After every *D* or *W* command, call *write_account* to record the new balance and the latest transaction history for the account.

Step 6: Read in the account data

What good is saving the account's data to a file if we don't read the file in at some point? The right time to read an account's data file is when that account is selected with the `a/A` command.

Modify the code for your `a/A` command so that it reads in the balance and command history from the `account_NNNNN.data` file. **Your program should then use this balance and history as the starting point for subsequent commands.** Use the `read_account` function from `accountio.py` to do this -- look at the code and comments for the function to figure out how to use it.

Step 7: Test your code

You know what to do. :-)

Submitting Your Work

Submit your Python file to Canvas before the due date.

2157 Problem Set Rubric		
Criteria	Ratings	Pts
Code correctness Up to 8 points will be earned by passing tests that call your code and check for correct results.		8.0 pts
Code quality Up to 2 points will be awarded based on the quality of your code: 2 points if your code is excellent. It is clean and compact, and it is very sophisticated given your experience level. There is little if any room for improvement. 1 point if your code is good. It is representative of a beginning coder, so there are some poor coding practices, but very few. There is room for improvement, but your code indicates that you are paying attention to good coding technique. 0 points if your code is fair to poor. It contains several poor coding practices. There is significant room for improvement.		2.0 pts
		Total Points: 10.0