# SY201 – Cyber Operations Fundamentals; Programming Assignment `06`

ATM

Name: ______________________________ Alpha: __________________________

Begin this assignment by making sure your repo is up to date. Change to **`~/repo201`** and type:

```
git pull
```

All the files required for this assignment, including an electronic version of this handout, are available in:

```
~/repo201/programming/pa06
```

---

*Read this entire handout at least once before you begin your research or start coding*

## 1. **Background**:

When you login to your Naval Academy e-mail account you enter your username and password to authenticate (prove) who you are. It would represent a major security flaw if the system validated your password by simply looking it up in a table of stored passwords. For example, if your username was `m841234` and your password was `gonavy1234`, keeping passwords as shown in Table 1 (left) would be a very bad idea. If that table were compromised, the system's usernames and passwords would be revealed.

Instead, modern systems actually store *hashes* of passwords rather than the passwords themselves[1]. When you type your username and password to authenticate, the system runs your typed password through a *hash function* and looks for you in a table of usernames like the one shown in Table 1 (right). If the hash of the password you typed matches the hash that's kept on file you're in!

| username | password |
|---|---|
| m840154 | billTheGoat |
| m849983 | midStore345 |
| m841234 | Gonavy1234 |
| m846245 | cyberMajor |

| username | password hash (md5) |
|---|---|
| m840154 | da53e91ddf332ad2fbfdaaeadedf3448 |
| m849983 | a11369aa7952b320e31d00a256776f77 |
| m841234 | d3b91abaf03a858494084842c4c433fc |
| m846245 | 4448f90b9f7a434002dabbc7b89b3a16 |

Table 1. Two Ways to Store Passwords

[1] *Additional techniques are used to make systems secure, such as salting and stretching passwords.*

## 2. **Your Program Should Do This:**

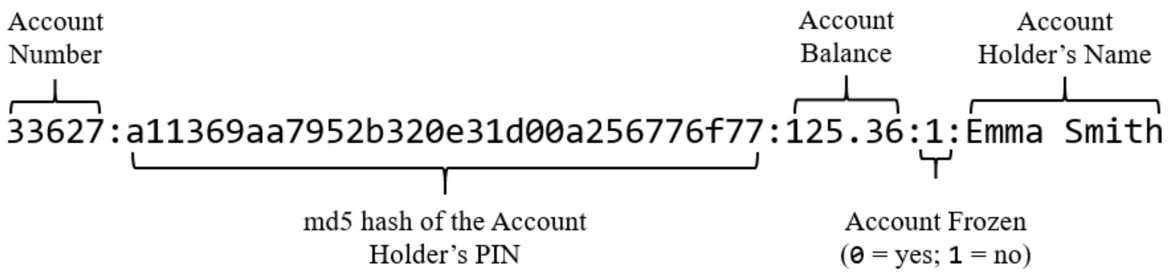
For this assignment you're going to write a python program that simulates an Automated Teller Machine (ATM). Your program will first authenticate account holders who use a 5-digit account number and a 4-digit Personal Identification Number (PIN) to "*login*" to your simulator. Your program will then allow them to perform one of the following options:

```
1. Withdraw funds
2. Deposit funds
3. Balance inquiry
4. Change PIN
5. Quit
```

Your instructor will provide you with a file that allows your program to initialize the simulator with user accounts. The file will be called `accfile.dat`, and will contain multiple lines (separated by '`\n`'), each with the following format:

- 5-digit account number
- 1 colon
- 32-character hash of a PIN (created using `md5`)
- 1 colon
- Dollar amount (account balance)
- 1 colon
- A single character (flag) set to `0` (account is frozen) or `1` (account is not frozen)
- 1 colon
- The remainder of the line is the name of the account holder, which can be any number of characters.

Here's a sample of what a single line from `acctfile.dat` will look like:


Account Number
Account Balance
Account Holder's Name
33627:a11369aa7952b320e31d00a256776f77:125.36:1:Emma Smith
md5 hash of the Account Holder's PIN
Account Frozen (0 = yes; 1 = no)

3. **Design Requirements:**

a. Think carefully about how you want to approach this assignment before you start coding. It's rarely effective when you open up your editor and just start typing. Think first about the kinds of variables you will use, their names, and how the different sections of your code will interact. Take out a piece of paper and pencil and sketch out the major components of your design (collect input, validate input, perform calculations, present results, etc.). Draw simple flow diagrams or write code-like snippets (called *pseudocode*) that capture the major steps in your program. Your design sketch and / or pseudocode is a required deliverable for this assignment.

b. The recommended approach for this assignment is to use five lists:

1. `acctNumsList`
2. `hashedPINsList`
3. `balanceList`
4. `frozenList`
5. `nameList`

For example, if you had an account data file with these three lines:

```
05194:1ddfa4ccdcea53130b500eaabb190e4b:29294.03:0:Reynoldus Psara
34634:ab452534c5ce28c4fbb0e102d4a4fb2e:35109.89:1:Sybille Stelymes
72327:7180cffd6a8e829dacfc2a31b3f72ece:13241.81:1:Cilia Monchaty
```

You would store that in five separate lists as follows:

| acctNumsList | hashedPINsList | balanceList | frozenList | nameList |
|---|---|---|---|---|
| "05194" | "1ddfa4ccdcea53130b500eaabb190e4b" | 29294.03 | "0" | "Reynoldus Psara" |
| "34634" | "ab452534c5ce28c4fbb0e102d4a4fb2e" | 35109.89 | "1" | "Sybille Stelymes" |
| "72327" | "7180cffd6a8e829dacfc2a31b3f72ece" | 13241.81 | "1" | "Cilia Monchaty" |

Pay careful attention to the data types in the lists above. Most of the data is stored as a strings, except items in the `balanceList` (`float`).

Also, key to the implementation of this structure is that the relative positions of items in your lists don't change, i.e.: the second account number in `acctNumsList` goes with the second hashed PIN in `hashedPINsList`, and the second balance in `balanceList`, etc.

c. You should hard-code the account data file needed to initialize the simulation as a global variable; you don't need to prompt for an account data file every time you run your program. Open the file for reading (`"r"`) and read one line at a time. Individual account fields (PIN, balance, etc.) are separated by colons, so you can break each line apart and append each field to its associated list. When this step is complete, you'll have five lists containing all the information for the account holders in your ATM and your simulation will be ready to run.

d. Upon completing step (c), above, close the account file. When the user exits your program, before quitting, you will re-open the same file for writing (`"w"`) and write all the data in your lists back to the file in the correct format. This will overwrite the original information in the file, but that's okay. Think if it this way: *You're saving the state of your ATM at the end of the business day, so when you reopen in the morning you can restore it and pick up where you left off.*

e. Next, ask for a 5-digit account number. If the number doesn't exist in the `acctNumsList`, present an appropriate warning message and prompt the user again. Keep doing this until you get an account number that exists in your ATM.

f. Once you get a valid account number, ask for the associated PIN. If the user types in a correct PIN, present your program's options. If the user enters an incorrect PIN, provide a warning message and prompt for the PIN again. If the user enters an incorrect PIN three times, then freeze the associated account (set the correct value in the `frozenList`), print a warning about too many PIN tries, and exit your program.

g. Your ATM will only dispense \$10 and \$20 bills. When you dispense funds, your program will print the number of bills of each denomination that are dispensed and decrement the account holder's available balance by the amount withdrawn. Think carefully about what this means when a user asks for a withdrawal that is not evenly divisible by 10.

h. When a user asks to show the account balance, display the following information: account holder's name, account number, and balance. For example:

```
Acct Holder: Reynoldus Psara
Acct Number: 05194
Acct Balance: $29,294.03
```

i. You must use formatted output, including dollar signs, when you display dollar amounts.

j. If a user tries to login with an account that's already frozen, print a notification that the account is frozen and exit your program. Do not ask for the PIN on a frozen account.

k. Your program will run in a loop (repeatedly displaying and executing options) until `Quit` (Option 5) is selected.

l. You must make use of functions in this assignment. At a minimum you must design and use the following five functions:

   i. `def getInt(prompt, lower, upper):`

      This function prompts the user for an integer in the range: [`lower,upper`], using `prompt` and returns a valid integer to the calling program. The function must notify the user if an invalid integer is entered (e.g. a string) or if the integer entered is not within [`lower,upper`]. For these cases, keep looping until the user provides compliant input.

   ii. `def isCorrectPIN(pin, index, hashedPINsList):`

      This function takes the `pin` parameter and hashes it using an `md5` digest. It then compares that result to the item located at position `index` in `hashedPINsList`. If the hashes match, `isCorrectPIN` returns `True`. If not, `isCorrectPIN` returns `False`.

   iii. `def userLogin(acctNumsList, hashedPINsList, frozenList):`

      This function prints your program's welcome banner and asks for the account number and PIN. It will return an integer that represents the index in the `acctNumsList` where the selected account number resides. This integer will then be used throughout your program as you process data in your lists.

Some special notes about `userLogin`:

1. If the entered account number does not exist in `acctNumsList`, keep asking until you get a valid account number.

2. If the account is frozen, then `userLogin` will return `-1`.

3. `userLogin` will call your `isCorrectPIN` function to validate the user. If `isCorrectPIN` returns `False` (meaning the user entered an incorrect PIN three times), then `userLogin` will freeze the account and return `-1`.

iv. `def changePIN(index, hashedPINsList):`

This function takes an `index` and a list of hashed PINs and does not return a value. It prompts the user to enter a new PIN, and updates the `hashedPINsList` at position `index` with the `md5` hash of the new PIN.

You must ensure a valid new PIN in the format `NNNN` is entered, where `N` is a digit in the inclusive range `[0,9]`.

Once you get a valid PIN, you must ask the user to type-in the PIN again for verification. You only need to do this verification once. If the two PINs don't match, then no updates to the list will happen and your function should print an appropriate notification message of your choosing and return to the main menu.

If the two PINs match, update `hashedPINsList` and print an appropriate notification message of your choosing. *Remember, this function has no return value.*

v. `def writeLineToFile(f, index, acctNumsList, hashedPINsList, balanceList, frozenList, nameList):`

This function takes seven parameters and does not have a return value. It takes an open file handle (`f`) and gets the item located at position `index` in each of the provided lists. It then creates a string that complies with the file format specification given in paragraph (2) and writes that string to the file using `f`.

*Note (1): `f` must be a handle for a file that is already open with the correct mode for writing. Do not open or close `f` inside `writeLineToFile`. Simply write the correct line to the file.*

*Note (2): The above presentation of the `writeLineToFile` function definition may be new to you. It conforms to the Python 3 style guide[2], which allows you neatly organize code that exceeds a single line length.*

4. **Error Handling**

   a. You must not allow a user to withdraw more funds than the available balance in the account. If an attempt is made to withdraw more than the account balance, print a notification message and continue asking for an amount to withdraw until a valid amount is entered.

   b. You must not allow a user to withdraw a negative amount of money. This would have the undesirable effect of actually acting like a deposit to an account. If an attempt is made to withdraw a negative amount of money, print a notification message and continue asking for an amount to withdraw until a valid amount is entered.

   c. You must not allow operations on an account that is frozen. If a frozen account is selected in the simulation, print a suitable warning message and exit the program without asking for a PIN.

   d. All withdrawals must be able to be serviced using only \$10 and \$20 dollar bills. If an attempted withdrawal cannot be serviced with \$10 and \$20 dollar bills, print a suitable notification message and re-display the options menu (start again).

   e. There are other edge cases for this program and many potential possibilities for it to crash or exhibit bad behavior. You should take care of as many edge cases as you can possibly identify.

[2] *https://www.python.org/dev/peps/pep-0008/*

5. **Program Flow**

Here's a sample run of the program

```
*** Welcome Cyber Citizens National Bank ***

Account Number: 20361
PIN: 9157

1. Withdraw funds
2. Deposit funds
3. Balance inquiry
4. Change PIN
5. Quit

Selection: 3

Acct Holder: Merbal Van Nieuwkirk
Acct Number: 20361
Acct Balance: $44,352.50

1. Withdraw funds
2. Deposit funds
3. Balance inquiry
4. Change PIN
5. Quit

Selection: 1

Amt to withdraw: 25010

Dispensing cash...
$20 bills: 1,250
$10 bills: 1
```

```
1. Withdraw funds
2. Deposit funds
3. Balance inquiry
4. Change PIN
5. Quit

Selection: 3

Acct Holder: Merbal Van Nieuwkirk
Acct Number: 20361
Acct Balance: $19,342.50

1. Withdraw funds
2. Deposit funds
3. Balance inquiry
4. Change PIN
5. Quit

Selection: 4

Enter new PIN: 1234
Confirm new PIN: 1234
PIN has been updated.

1. Withdraw funds
2. Deposit funds
3. Balance inquiry
4. Change PIN
5. Quit

Selection: 5
```

6. **Testing:**

Thoroughly testing your program can be a difficult task. For complex programs there may be hundreds or thousands of possible cases to test. Try to think about what kind of input will cause bad behavior in your program and run those tests.

7. **Assumptions You May Make:**

   a. The `acctfile.dat` file will be properly formatted, with no hidden anomalies.

   b. The `acctfile.dat` file will exist in the current directory (you don't need to "*try*

– *except*" when you open the file).

c. Deposits can be of ***any positive amount***, including cents. They do not have to be in multiples of $10 or $20.

d. You can use generic notification messages for all invalid integers or floats your user may try to input (menu selections, withdrawal amounts and deposit amounts). For example, if someone selects option `6` for your program you could respond with: `Integer out of bounds.`

e. All account numbers in the `acctfile.dat` file will be unique (no duplicates).

f. Zero (`0`) is a valid amount to withdraw from or deposit to an account.

8. **Hints:**

a. Give your variables meaningful, descriptive names.

b. Since your program will be designed to overwrite the data file, you'll end up making a fresh copy from the repo every time you want to start over.

c. Keep account numbers, PINs, and frozen flags as string data types. It will make comparison operations much simpler. If a PIN is a string, what operations can you use to ensure its length is four, and it's comprised only of digits?

d. Make sure to cast account balances to `float` when reading them in, and back to `string` when writing them out.

e. It sure would be great if we had a `getFloat()` function to handle deposits. ☺

f. When you're validating PINs (inside the `changePIN` function) you don't need to check if the second PIN attempt is in the proper format (`NNNN`). If it doesn't match the first PIN (which you already validated) then it doesn't matter.

g. There is no upper limit on deposits. Research `math.inf`.

h. The directory for this project includes an executable version (called `atm`) that you can run to get a sense for how your program should operate. It also includes the required data file (`acctfile.dat`), and a separate version of the data file with all the PINs exposed (`acctfile.txt`). You'll need `acctfile.txt` with the visible PINs so you can run tests on your code.

Let's assume you're working in your `~/shares/sy201` directory. You can run `atm` by first copying it to your working directory and then adjusting its permissions with the following commands:

i. `cp ~/repo201/programming/pa06/atm .`
   (*the trailing space and period in the command above are important*)
ii. `chmod 755 atm`

You only have to perform steps (i) and (ii) once. After that, you can run the program anytime you want by typing: `./atm`. Just make sure the `acctfile.dat` file is located in the same directory as `atm` when you run it.

9. **Deliverables And Due Dates:**

   a. Using paper, and pen or pencil, complete your pseudocode / flow diagram *before* you start coding. You may use more than one page for this if you need it. Be sure to indicate your name and alpha on this page.

   b. Your completed source code (the one ending in `.py`).

   c. Any custom library functions you write, all contained in a single directory e.g. `utils`.

   d. Submit parts (a) through (c) by 2359 on Tuesday, 06 November, in accordance with your instructor's directions.