Homework 1 - Password Utilities

Due Date: Tuesday, September 29, 5PM

Problem Description

These are the rules for valid University of Minnesota internet passwords.
Passwords must:

1. Be from 8-32 characters in length
    - example:
        - "a" invalid
        - "" invalid
        - "aA1!" invalid
        - "Lorem ipsum dolor sit amet, consectetur adipiscing elit. Morbi purus odio, euismod at egestas id, consectetur sit amet nullam1!" invalid
        - "A long Password with 24!" valid
2. Not end in a space, although they may contain spaces
    - example:
        - "3BlindMice! " invalid
        - "3 Blind Mice" valid
3. Contain characters from three of the following four categories:
    - English uppercase letters (A through Z)
    - English lowercase letters (a through z)
    - Digits (0 through 9)
    - Non-alphanumeric characters (e.g., !, #, %)
    - example:
        - "QWER1212" invalid
        - "qwer1212" invalid
        - "qwer!@#$" invalid
        - "QWER!@#$" invalid
        - "qwER12#$" valid
        - "qwer!@34" valid
        - "QWer1234" valid
4. Not contain the user's first name, middle name, or last name.
    - example: my full name is Daniel Allen Kluver
        - "Daniel Allen Kluver" invalid
        - "asdfl;h__daniel__asdf;lkjasd312f" invalid
        - "the conductor yells __allen__!" invalid
        - "dani3l All3n klvu3R" valid

    - Note that this rule ignores case:
        - "dANIel278" invalid
5. Not contain all or any three character part of the user's account name. Note that this rule ignores case, too.
    - example: my account name is kluve018
        - `"my name is kluve018 and I am super cool!" invalid
        - "I l__uve__ P1zza!" invalid
        - "I am worth 1__018__$" invalid
        - "got a __KlU__?" invalid
        - "K l u v e 0 1 8 @ u m n . e d u" valid

Homework Description

You will write five functions (and any other 'helper' functions you decide are necessary):

- isValid(password, first_name, middle_name, last_name, account_name)
    - Determines whether a password is valid for a particular user. Returns True if it is and False if it is not. If it is not valid, an error message will be printed to explain why, that is, which of the rules was violated.
- generateValidPassword(first_name, middle_name, last_name, account_name)
    - Generates a (random) valid password for a particular user.
- phraseToPass(pass_phrase)
    - Takes a series of words and returns a potential password by taking the first letter of each word in the passphrase. Also include any punctuation symbols like ".,!?()[]". This may help generate easier to remember passwords for the user (although it may generate invalid passwords). For example, it might be easy for you to remember the phrase "Computer Science at the U of M is fun!" and from it generate the password"CSatUoMif!"
- obfuscate(password)
    - Takes a simple password (e.g., as generated by phraseToPass) and performs several simple string replacements to make more random passwords (again, these may not be valid). For example it may be easy to remember "Bite Me" and from it generate "B1t3 |\/|3". We'll give some more detailed instructions about these replacements below.
- generatePasswordFromPassPhrase(pass_phrase, first_name, middle_name, last_name, account_name)
    - (You only need to write this function if you attempt the extra credit.) Generates a password for a particular user from a specified passphrase. This method can result in passwords that are both easy for a user to remember and hard for someone else (including an automated password cracking program) to guess. For example you may find "I am so enormously, Monstrously happy !" easy to remember, from it you can generate a password like "1as3, |\/|h!!" We'll give some more detailed instructions below.

You will write progressively more complicated versions of these functions at the C-level, B-level, and A-level.
Even if you are going to satisfy the A-level requirements, you still should first solve all the C-level requirements (then test to see that your solution is correct), then solve all the B-level requirements (and again test that your solution is correct) before completing the A-level requirements.

Homework requirements.

You are required to turn in two files passwords.py, containing the required functions, and README.txt, which must specify which level you are attempting/ claiming to have completed. As always, if you do the homework with a partner, both of you must include your names in the README.txt file.
To help you gauge your progress and test your code we have developed a test suite you can use to test your code. This test suite is available in the Assignment folder. Don't treat the test suite as complete: we will test and inspect your code using more tests than those we have provided.

Grade level descriptions

C-level

At this level you will write simplified versions of isValid and generateValidPassword satisfiyng only two of the five rules.

- isValid(password, first_name, middle_name, last_name, account_name)
    - Checks whether the password is a valid length (Rule 1) and does not end in a space (Rule 2). Returns True if both conditions are met and False otherwise.
- generateValidPassword(first_name, middle_name, last_name, account_name)
    - Generates and returns a random string that is at least 8 character long and at most 32 characters long and does not end in a space.

You will also implement phraseToPass to help take passphrases (which tend to be too long) and make shorter, but hard to guess passwords.

- phraseToPass(pass_phrase)
    - Return a string consisting of the first letter of each word and any punctuation symbols in the pass_phrase.
    - A word is a sequence of non-blank characters surrounded by blank spaces and/or the beginning/end of the string. For example, the words in "Computer Science at the U of M is fun!" are
    - “Computer", "Science","at", "the","U", "of","M", "is","fun!".
    - Notice that this method is not guaranteed to generate a valid password. That's OK.

B-level

At the B-level you should implement rules 1, 2, and 3 of the U of M password rule set.

- isValid(password, first_name, middle_name, last_name, account_name)
    - New functionality: Check to be sure that the password uses at least three of the four required character sets. (Rule 3)
- generateValidPassword(first_name, middle_name, last_name, account_name)
    - New functionality: Generate a password that uses at least three of the four required character sets. Note: there are (at least) two reasonable ways to do this.

You will also generate a function to help take passwords that do not have numbers and symbols and add numbers and symbols in an easy to remember way

- obfuscate(password)
    - Return a string consisting of the password after applying a few well-known substitution rules.
    - "and" ==> "&"
      "e" or "E" ==> "3"
      "i" (lowercase i) ==> "1" (number 1)
      "I" (uppercase I) ==> "|" (vertical bar)
      "o" (lowercase o) ==> "0" (number 0)
      "O" (uppercase o) ==> "0" (number 0)
      "W" (uppercase W) ==> "|/\|"
      "M" (uppercase M) ==> "|\/|"
    - Notice that this method is still not guaranteed to generate a valid password... and this is still OK!

A-level

At A level you should implement all 5 rules for valid UMN passwords.

- isValid(password, first_name, middle_name, last_name, account_name)
    - New functionality: Check to make sure that the rules concerning the user's name and accountname are followed.
- generateValidPassword(first_name, middle_name, last_name, account_name)
    - New functionality: Generate a password that follows the rules concerning the user's name and accountname.

Extra credit

You have the chance to get up to 5 extra credit points for your work at this level.
You should write a function generatePasswordFromPassPhrase(pass_phrase, first_name, middle_name, last_name, account_name)
This is like phraseToPass and obfuscate in that it helps users generate complex passwords from simple, easy to remember passphrases. The main difference is that this should always return a valid password.

There are two approaches to this. We'll describe them, then let you decide which one to pick. First, a clarification: by "valid", we mean "valid according to the definition of isValid you implemented at the A-level".

Approach 1. Apply the techniques you implemented at the C-level and B-level. Check to see if the resulting password is valid (according to your A-level definition of isValid). If it is, great, return it. If not, print out a string telling the user why it wasn't valid -- i.e., which of the password validity rules was not satisfied -- and ask the user to enter another phrase. Keep repeating this process until it results in a valid password.

Approach 2. Apply the techniques you implemented at the C-level and B-level. Check to see if the resulting password is valid (according to your A-level definition of isValid). If it is, great, return it. If not, modify the password to make it valid. (Hint: you might borrow some of the techniques you used for generateValidPassword.)
For either approach we encourage you to try to come up with more helpful techniques than just those we introduced at the C and B levels. For example you could consider more replacement rules, reverse parts of the passwords, or try to make the password a palindrome. Whatever you do, document your approach in your README.txt file. Include a brief explanation of why you picked this approach, and what its advantages and disadvantages are.