CSCI 140 Final Project: Password Manager

Due on Thursday December 15, 2022, by 2200 hrs EST

For this project, you will create a class to represent a Password Manager and write a main program which creates and tests an instance of this class. The Password Manager can create and store logins and passwords to websites for a user. Your Password Manager will store a collection of web site addresses with their associated username and password. What we mean by site, username, and password are, for example:

Site: www.gmail.com Username: a_student Password:@5tud3nt!

Your PasswordManager class definition will make use of the password_specs and password_gen functions from Project 2. You are given correct implementations of these as functions, which you will need to make minor edits to for them to properly in this new context (more details below).

Part 1: The PasswordManager class

Your PasswordManager class will need to store the following data:

- **The name of the user.** The user can and must specify this. This is the owner of the password manager instance.
- **A master password.** The user can and must specify this. This is a common feature of commercial password managers – access to the password manager is controlled by a master password, and the master password is required to make certain changes. In our implementation, we will require the master password when the user wants to change the password for a site.
- **The collection of sites with their associated usernames and passwords.** You will store this as a Pandas Dataframe. Consider that the search index/row_name will be the web site and the username and password should be retrievable using that index/row_name. This collection starts out empty when the PasswordManager object is initially created.

Some important points to consider:

- The skeleton file PasswordManager.py includes a full outline of the methods you need to implement as well as guidelines for implementing them.
- This implementation assumes that each site stored is unique. This means that only one username and password can be stored for each site.

Constructor

The PasswordManager constructor should work as follows:

- The user will pass in two arguments: name – the name of the owner of this PasswordManager and master_pw, which is the master password selected by the user. Both of these should be stored as private attributes of the PasswordManager object. You have already been given variables to hold these attributes.
- The constructor should also create an empty dataframe that will hold the websites and their associated data. You have already been given the code for this.

__password_specs(self, length = 14, min_spec = 0, max_spec = 0, min_num = 0, min_upper = 0)

- **__password_specs** is a private method that works nearly identically to the password_specs function you wrote for Project 2.
- This method takes five optional arguments: length – the desired length of password, min_spec – the minimum number of special characters, max_spec – the maximum number of special characters, min_num – the minimum number of numeric digits, and min_upper – the minimum number of uppercase characters. The function returns a list containing four integers: a number of special characters, a number of numeric digits, a number of uppercase letters, and a number of lowercase letters. The method should print nothing. Remember that since the method accepts optional arguments it may be called with from 0 to 4 arguments. Notice there is no argument pertaining to lowercase letters. For more details, refer to the Project 2 specifications.

- You have been given correct code for this method. **The only thing you need to do is add in code to handle a maximum number of special characters.** This is a very minor edit. You should not add an additional line. Do not overthink this.

__password_gen (self, criteria = None, length = 14, spec_char = '@!&', repeat = True, min_spec = 0, max_spec = 0, min_num = 0, min_upper = 0)

- **__password_gen** is a private method that works identically to the password_gen function in Project 2 with three exceptions (see below). You have been given code that is algorithmically correct. You need to adjust the code so that it works as a method, and implement the criteria functionality (explained below)
- This method takes 7 optional arguments. It returns a password that meets the specifications. There are two changes from the functionality in Project 2. It accepts an optional argument called max_spec. The only functionality of this argument is that it is passed in to password_specs (since that also takes a max_spec argument) to specify the maximum number of special characters.
- The second change is that this function can optionally accept a dictionary called criteria. The entries in criteria will correspond to the other arguments the function takes. Here are some examples:

  1. criteria could be the dictionary: { 'min_num ':2, 'min_upper':3, 'length':10}
  2. criteria could be the dictionary: { 'max_spec ':4, 'repeat ':False}

- Note that the user doesn't have to specify something for every parameter in criteria. You can assume that the dictionary will be in the correct format, i.e., the keys will be strings and the values will be integers, Booleans or strings as appropriate. You do not need to check the types of the keys or values.

- You need to write code where indicated to overwrite any of the arguments that were passed in with any values in the dictionary. This is VERY SIMPLE conceptually, though it make take several lines of code to accomplish. Do not overthink it.
- The error message has been removed. There is no else block for invalid specifications. **Do not add one.**

add_password(self, site, username, criteria = None)

- **add_password** is a public method that allows the user to add a website with username and password to the collection of stored websites
- **add_password** takes two required arguments: site, a website, and username. For example, site might be www.wm.edu and username might be a_student. It also takes an optional argument criteria which is a dictionary. criteria, if used, will be in the format discussed above (a dictionary). It will contain mappings of parameters for **__password_gen** and **__password_specs** to their values.
- **add_password** randomly generates a password for the site specified by the user according to the specifications (if any) in criteria. It should and MUST call the **__password_gen** method to do this.
- If the user provides impossible specifications, the site should not be added. Think about how to detect this. Error checking for the specifications occurs in **__password_gen** and MUST NOT be repeated in **add_password**. If the specifications are not valid, **add_password** should print an error message that reads 'Invalid Specifications!'
- If a website is already listed in the collection of sites this method should do nothing. It should not change the existing password, and it should not print an error message.
- This method returns nothing. (You should NOT add in a line that says return None)

validate(self, mp)

- **validate** is a public method that checks whether or not the string mp is the same as the master password stored in the object.
- **validate** returns a Boolean indicating if mp is the master password. This is VERY SIMPLE.

change_password(self, site, master_pass, new_pass = None, criteria = None)

- The **change_password** method takes two required arguments: site, the website to change the password for, and master_pass, a value for the master password (see below). The optional argument new_pass contains a new password that the user would like to replace the old password with. There is an optional argument criteria, which is a dictionary in the same format as shown above for **add_password**.

- **change_password** should first check if the user has entered the correct master password. You must make use of your **validate** method to do this. If the master password is not correct, this function should print an error message (Incorrect master password!) and return False.
- **change_password** also needs to check if the site exists and can be changed. If not, it should print an error message (Site does not exist!) and return False.
- If the user has provided a value for new_pass, the password for site should be updated to that value. In this case the function returns nothing (you should NOT have a line that says return None)
- If the user has not provided a value for new_pass, **change_password** should use **__password_gen** to create a new password with the dictionary criteria.
- **change_password** also needs to check if the **__password_gen** returns a new password. If not, it should print an error message (Invalid criteria!) and return False.
- If **__password_gen** returns a password, **change_password** should update the password and return nothing (you should NOT have a line that says return None)

remove_site(self, site)

- **remove_site** is a public method that allows the user to delete a website. It takes one required argument site, which is the site to remove, along with its associated data.
- **remove_site** should do nothing if the site does not exist. It should not return anything in any case (you should NOT have a line that says return None)

get_site_info(self, site)

- The **get_site_info** method takes one argument: site, a website for which to retrieve information.

This method should return a list containing the username and password for that site. Think carefully about how you want to return this list and the implications. (It would be better to use master password validation, but we’re not going to.)

get_name(self)

• **get_name** takes no arguments and returns the name of the owner of the PasswordManager. This is VERY SIMPLE. Do not overthink it.

get_site_list(self)

• **get_site_list** returns a list of all of the websites stored in the PasswordManager. This is JUST THE WEBSITES. It should not have any usernames or passwords. ASK IF YOU DO NOT UNDERSTAND THIS.

__str__

• The **__str__** method for the PasswordManager class will return a string that contains a header line in the format: 'Sites stored for NAME_OF_OWNER, and then all of the websites in the PasswordManager, one per line. Here's an example:

Sites stored for John Snow:

www.dragons.com

www.kingofthenorth.com

www.justfoundoutimthetrueking.com

We will test your method implementations using a series of test cases, so be sure to test them thoroughly. Each tests something different about your program. Your goal is to break your program, so be aggressive with your tests. Consider how calling one method could influence the values returned or used by several other methods.

Part 2: Creating and using a PasswordManager

You will write a main program that makes use of your PasswordManager class. Your main program should be saved and submitted in a separate file called Final_Main.py – we have provided a skeleton file for this. **Do not change the import lines at the top.** Your main program will create a single instance of a PasswordManager object, and will carry out a set of operations.

The specific instructions are:

- Create a PasswordManager with the name 'Student' and the master password 'FINAL'
- Add the site 'www.gmail.com' with the username 'a_student' and a randomly generated password using the default settings
- Add the site 'www.wm.edu' with the username 'WMstudent' and a randomly generated password that has a maximum of 2 special characters and a minimum of 2 upper case characters, all other parameters are defaults
- Change the password for 'www.gmail.com' to 'update1235'
- Get the site information for 'www.wm.edu'
- Print out a string representation of the PasswordManager object you made
- We have provided you with a space in the skeleton file to fill in your main program. The skeleton file re-iterates the instructions.

SUBMISSION EXPECTATIONS

Because the name of the classes, methods, and files **must be exactly as specified** in order for our testing to work, we have provided a skeleton file for you to fill in. Be certain not to change any names, and do not modify the numbers of parameters for each specified method.

PasswordManager.py: The skeleton file for the class definition attached to this project, but with your implementations added. The file must contain only the methods provided and you must not change their names or parameters.

Final_Main.py: The skeleton file for the main program, but with your code added.

Final.pdf: A brief write-up describing any difficulties you encountered and how you overcame them.

POINT VALUES and GRADING RUBRIC

PasswordManager method implementations:

__init__ completion: 2 points

__password_specs update: 2 points

__password_gen completion and update: 7 points (criteria part graded on style)

add_password: 15 points

change_password: 20 points

validate: 2 points

remove_site: 6 points

get_site_info: 5 points

get_name: 2 points

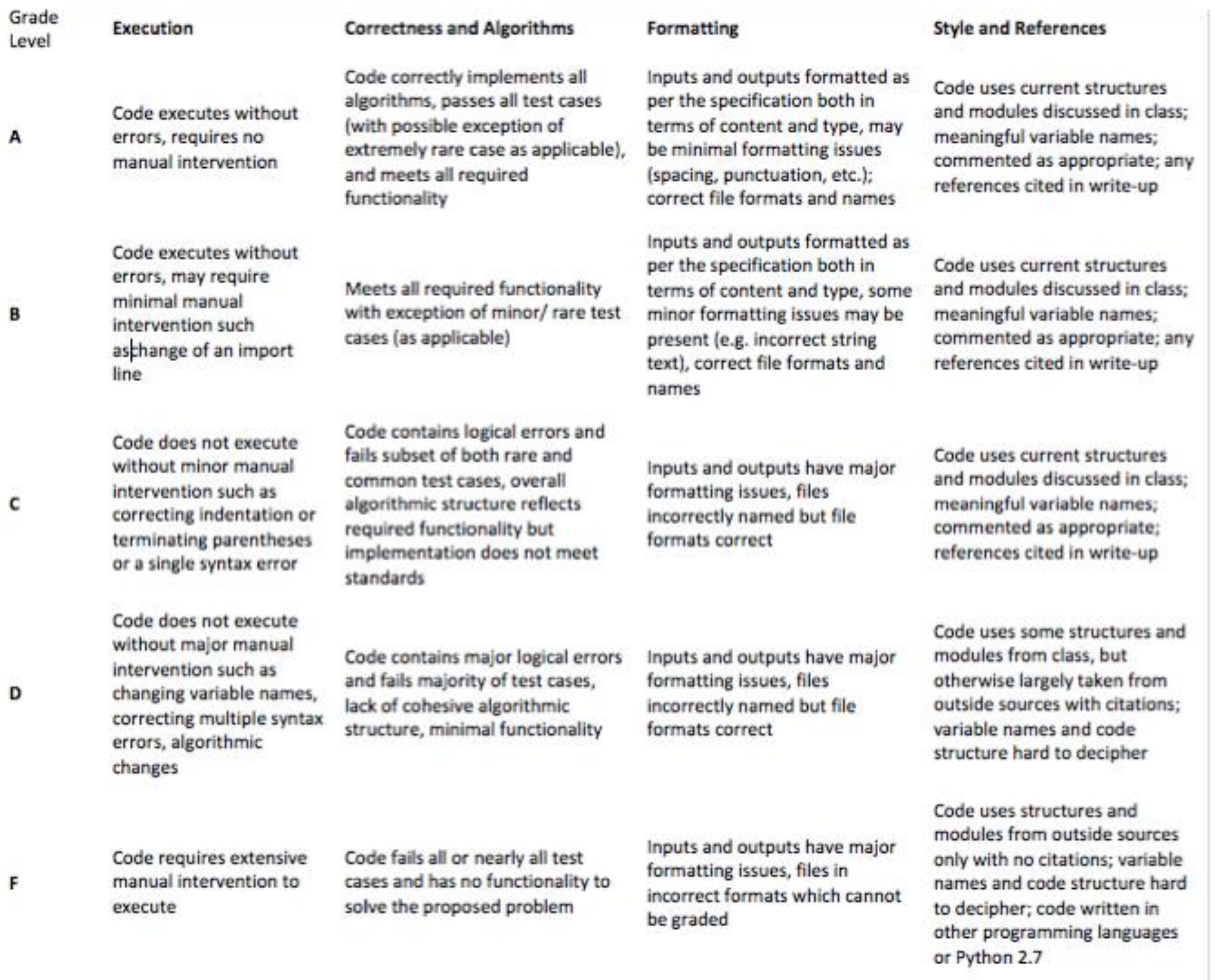
get_site_list: 5 points

__str__: 12 points

Main program: 10 points

Write-up: 3 points

Autograder points: 9



Grade Level	**Execution**	**Correctness and Algorithms**	**Formatting**	**Style and References**
A	Code executes without errors, requires no manual intervention	Code correctly implements all algorithms, passes all test cases (with possible exception of extremely rare case as applicable), and meets all required functionality	Inputs and outputs formatted as per the specification both in terms of content and type, may be minimal formatting issues (spacing, punctuation, etc.); correct file formats and names	Code uses current structures and modules discussed in class; meaningful variable names; commented as appropriate; any references cited in write-up
B	Code executes without errors, may require minimal manual intervention such aschange of an import line	Meets all required functionality with exception of minor/ rare test cases (as applicable)	Inputs and outputs formatted as per the specification both in terms of content and type, some minor formatting issues may be present (e.g. incorrect string text), correct file formats and names	Code uses current structures and modules discussed in class; meaningful variable names; commented as appropriate; any references cited in write-up
C	Code does not execute without minor manual intervention such as correcting indentation or terminating parentheses or a single syntax error	Code contains logical errors and fails subset of both rare and common test cases, overall algorithmic structure reflects required functionality but implementation does not meet standards	Inputs and outputs have major formatting issues, files incorrectly named but file formats correct	Code uses current structures and modules discussed in class; meaningful variable names; commented as appropriate; references cited in write-up
D	Code does not execute without major manual intervention such as changing variable names, correcting multiple syntax errors, algorithmic changes	Code contains major logical errors and fails majority of test cases, lack of cohesive algorithmic structure, minimal functionality	Inputs and outputs have major formatting issues, files incorrectly named but file formats correct	Code uses some structures and modules from class, but otherwise largely taken from outside sources with citations; variable names and code structure hard to decipher
F	Code requires extensive manual intervention to execute	Code fails all or nearly all test cases and has no functionality to solve the proposed problem	Inputs and outputs have major formatting issues, files in incorrect formats which cannot be graded	Code uses structures and modules from outside sources only with no citations; variable names and code structure hard to decipher; code written in other programming languages or Python 2.7