CSCI 140 Programming for Data Science Project 2

Due Monday, October 10, 2022 at 1700

In our second project, you are tasked with developing functions to randomly generate passwords according to user-provided specifications. This is the kind of functionality implemented in password managers like LastPass. You will create two functions, correct a third function, and write a short main program that demonstrates the utility of your functions. You will make extensive use of functions from the random module to complete these tasks.

About passwords and specifications

In order to create a password, the user is able to specify a length, a minimum number of special characters (#$% etc.), a group of permitted special characters, a minimum number of digits, and a minimum number of uppercase characters. The user can also specify whether characters may repeat in the password, that is, if each character must be unique or not.

You may have encountered websites where you are asked to create a password and it lists specifications like these. Your functions could be used to create an acceptable password for such sites.

For example, a user could ask for a password of length 10 with a minimum of 1 special character chosen from $@#, a minimum of 2 numbers, and a minimum of 3 uppercase letters, allowing repeats. Examples of passwords that meet these specifications are:

- nXE4VQG73#
- laS$9A$N3$
- Otq6$0ZaAL
- 39B8J@e7D1
- #F@3N3$0Ya

You must use the structures we have learned in this class to complete this assignment. Implementations using structures we have not covered may receive no credit. **For example, you may not use string and list methods since we have not formally talked about methods.**

Before you begin, it will be helpful for you to review what each function we have seen in the random module does and the type of data it returns.

Function One: Generating Password Specifications

Write a function called **password_specs (length = 14, min_spec = 0, min_num = 0, min_upper = 0)** which generates specifications for the number of characters of four types that should be in a password based on minimum acceptable values. The four character types are special characters (e.g. $#%), numeric digits (0123456789), uppercase letters (e.g. ABCD), and lowercase letters (e.g. abcd).

This function takes four optional arguments:

- **length:** the desired length of password. Default value = 14
- **min_spec:** the minimum number of special characters. Default value = 0
- **min_num:** the minimum number of numeric digits. Default value = 0
- **min_upper:** the minimum number of uppercase characters. Default value = 0

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 function should print nothing. Remember that since the function accepts optional arguments it may be called with from 0 to 4 arguments. Notice there is no argument pertaining to lowercase letters.

You can assume that the arguments will all be valid as in they will be of the right types and have valid values both in terms of being non-negative and providing logically valid specifications for what a password can contain in terms of minimum values and length (this is explained further for the next function).

For example, suppose we call the function as follows:

password_specs(10,1,2,3)
It might return: **[1, 3, 4, 2]**

This means that our password of length 10 should have 1 special character, 3 digits, 4 uppercase characters, and 2 lowercase characters. This meets the specifications.

This function DOES NOT generate the password. It provides the number of each kind of character the password will contain.

This may seem strange, as in, if we are passing numbers for how many of these things we want, what is the function doing? The key to this lies in considering that we are only specifying MINIMUM values for special characters, digits, and uppercase letters. Note there is no minimum for lowercase characters. Let's look at an example.

Suppose that we want a password with 6 characters. We require a **minimum** of 1 special character, 1 digit, and 2 uppercase letters. To build a password that meets these specifications, we could use any of the following:

- 1 special character, 1 digit, 2 uppercase letters, 2 lowercase letters
- 2 special characters, 1 digit, 2 uppercase letters, 1 lowercase letter
- 1 special character, 2 digits, 3 uppercase letters, 0 lowercase letters
- 1 special character, 1 digit, 3 uppercase letters, 1 lowercase letter

And so forth – there are many other possibilities. Why not just use the first one all of the time? We are trying to generate passwords that are secure and unique. Incorporating randomness and variation in the process gives a broader range of choices and makes it less likely that we will generate the same password twice.

The way we suggest you implement the password_specs( ) function is to define the number of each type of character in this order: special characters, digits, uppercase letters, lowercase letters. However, you can certainly implement it in a different way. This implementation of the function should not be extremely complex. **Don't overthink it.** You can actually do this in 5 or 6 lines with NO CONDITIONALS OR LOOPS! Though your implementation may differ.

Answer the questions below that relate to the password_specs() function. Working through these questions will help you understand how the function is supposed to work and give you some ideas about the suggested implementation.

Q1: Suppose that you have the following specifications: length of 8, minimum 2 special characters, minimum 1 digit, minimum 1 uppercase letter. The function randomly chooses the number of special characters to be 3. How many characters are still available to assigned to digits and letters?

A1 : 5

Q2: Suppose that you have the following specifications: length of 8, minimum 2 special characters, minimum 1 digit, minimum 1 uppercase letter. The function randomly chooses the number of special characters to be 3 and the number of digits to be 2. What are the possible values for the number of uppercase characters that can be in the password?

A2: 1/2/3

Q3: Suppose that you have the following specifications: length of 8, minimum 2 special characters, minimum 1 digit, minimum 1 uppercase letter. The function randomly chooses the number of special characters to be 2, the number of digits to be 1, and the number of uppercase characters to

be 2. What are the possible values for the number of lowercase characters that can be in the password?

A3: 3

Q4: Suppose that we call the function as follows: password_specs(1,1). What will be the values of the parameters used by the function?

A4: length = 1, special character = 1, digit = 0, uppercase letters = 0

Q5: Suppose that we call the function as follows: password_specs(min_num = 2). What will be the values of the other parameters used by the function?

A5: length = 14, special character = 0, digits = 2, uppercase letters = 0

Part Two: Generating the password

You have been given code for a function called **password_gen** which does not work properly. All of the errors in this code are related to indentation, logic, and not meeting the function specifications. You may not, and do not need to, add or delete any lines of code. You must correct this code in place, as is. If you are repeatedly deleting an entire line and re-writing it, you are on the wrong track.

password_gen(length = 14, spec_char = '@!&', repeat = True, min_spec = 0, min_num = 0, min_upper = 0) takes 6 optional arguments: :

- **length**: the desired length of password
- **spec_char:** a string containing permissible special characters
- **repeat**: a boolean which determines whether or not all characters in the password will be unique (True means characters can repeat, False means they cannot)
- **min_spec:** the minimum number of special characters
- **min_num:** the minimum number of numeric digits
- **min_upper**: the minimum number of uppercase characters

The function returns a password that meets the specifications in the argument list. For example: **password_gen(10, repeat=False)**

might return: **ABFoZdS@bY** (remember there is randomness in this process so there are MANY possibilities)

password_gen(10, repeat = True)

might return: **iX!&0zD7ii**

password_gen uses the password_specs function. In the event that you have not completed the password_specs function, you may use the password_specs_test function, which will return a hard-coded list of specifications. **ASK IF YOU DO NOT UNDERSTAND THIS!**

Things to consider for a correct working implementation of password_gen:

1. You do not have to worry about incorrect data types/values passed in to the function, this means that if an argument is supposed to be a non-negative integer, it will be a non-negative integer, if an argument is supposed to be a string, it will be a string. HOWEVER...
2. You do need to test and deal with incompatible argument values. That is, values the user inputs that cannot logically be used to make a valid password. Let's divide these up into a few groups.

- User asks for repeat = False and provides a string for spec_char that has fewer characters then specified by min_spec. For example, the user asks for repeat = False and provides min_spec = 6 and spec_char = '$#'. In this case the function should print an error message: "Invalid specifications." and return None
- User provides minimum specifications that exceed the total length. For example, suppose that length is 10 and the user asks for min_spec = 5, min_num = 7 – that would be over 10 characters total. In this case the function should print an error message: "Invalid specifications." and return None
- You also need to consider specifications generated by password_specs that may be invalid if repeats are not allowed. For example, suppose the user calls the function with repeat = False, spec_char = '#@$', min_spec = 2, min_num = 4, and length = 14. password_specs could generate the following output: [2, 11, 1, 0]. This corresponds to 2 special characters, 11 digits, 1 uppercase character, and 0 lowercase characters. However, there are only 10 possible digits: 0 1 2 3 4 5 6 7 8 9. If the user don't allow repeats, it is not possible to have 11 digits. When this occurs, you should have password_specs generate new specifications that will work if repeats are not allowed. Note that you may have to regenerate the specifications several times.

Before returning the password, password_gen should randomly permute (swap the order) of the characters in the password. For example, if the password were: a6%. The random permutation could result in the same password or any of the following: 6a%, %a6, 6%a, %6a, a%6.

Note that only ONE of these will be returned – whichever results from the random permutation of the letters.

Please see the following questions to ensure that you understand how this function works and how you should implement it.

Q1: The user calls password_gen with the following specifications: length = 8, spec_char = '$@#', min_spec = 3, min_num = 6, min_upper = 2, repeat = True. What will be the output?

A1: `Invalid specifications!`

Q2: The user calls password_gen with the following specifications: length = 8, spec_char = '$@#', min_spec = 3, min_num = 2, min_upper = 2, repeat = False.

A2: `&1@7#zAZ can be one of the possible outputs`

Q3: The user calls password_gen with the following specifications: length = 8, spec_char = '$@#', min_spec = 5, min_num = 1, min_upper = 1, repeat = False.

A3: `Invalid specifications!`

Q4: The user calls password_gen with the following specifications: length = 8, spec_char = '$@#', min_spec = 5, min_num = 1, min_upper = 1, repeat = True.

A2: `@@5W&@@# can be one of the possible outputs`

Q5: It is important to understand the difference between sampling with replacement and sampling without replacement to use the functions in the random library. If we have the following numbers: 2,10,19 which of the following could be samples of size 2 without replacement?

- `[2, 19]`
- `[2, 2]`
- `[19, 10]`
- `[10, 2]`
- `[10, 10]`

Q7: It is important to understand the difference between sampling with replacement and sampling without replacement to use the functions in the random library. If we have the following numbers: 2,10,19,50 which of the following could be samples of size 2 with replacement?

- `[2, 19]`
- `[2, 2]`
- `[19, 10]`
- `[10, 2]`
- `[10, 10]`
- `[50,19]`

Q8: Similar to the previous question, the gen_password function accomplishes multiple subtasks. To debug the code you were given, we suggest looking at it in chunks, where each chunk corresponds to a specific task. Though not exhaustive, some of these tasks are listed below. Match each task to the line of code that BEGINS (or may wholly contain) that task in the code you were given.

- To check if the arguments to the function call match the required specifications and whether to proceed with the password creation or print 'Invalid specifications'
- To use the password_specs function to create a list with the specifications ie. no. of special characters, digits, uppercase and lowercase letters
- If the arguments to the function call match the required specifications, we check for if repeats = True or repeats = False
- If repeat = True, generate a password that can have repeats (replacements)
- If repeat = False, generate a password that should not have repeats (replacements)
- Shuffle the generated password
- Add the contents of this password from list to a string.

Part Three: Checking passwords

When you create a password for web sites that have certain requirements e.g. one special character, one number, etc. The site will check the password you have entered to verify if it meets the requirements. You will write a function called check_password(password, length, min_spec, min_num, min_upper) that implements this functionality. check_password has five required arguments:

- **password:** the password to check (a string)
- **length:** the desired length of password
- **min_spec:** the minimum number of special characters
- **min_num:** the minimum number of numeric digits
- **min_upper:** the minimum number of uppercase characters. The function returns a Boolean: True if the password meets the specifications given by the arguments, and False if it does not.

This function should print nothing. The user will not pass in arguments of an incorrect type or that do not logically allow for a valid password.

Here are some examples:

check_password('AG$4agaga', 10, 2, 1, 1)

would return: **False**

check_password('apple', 5, 0, 0, 0)

would return: **True**

Answer the following question to ensure you understand how this function works:

Q: What does the function return for the various cases shown below? (True/False)

- check_password('bandana', 7, 3, 2, 1)
- check_password('banana', 6, 0, 0, 0)
- check_password('&1@7#zAZ', 8, 3, 2, 2)
- check_password('@@5W&@@#', 8, 6, 1, 1)
- check_password(' iX!&0zD7ii', 9, 2, 5, 2)

Part Four: Main Program

You will write a brief main program that makes use of your functions. This must be saved and submitted in a separate file called Project_2_Main.py – **if you put the main program in the same file as your functions expect ~5 point deduction.**

You must submit your main program in a separate file: Project_2_Main.py

Your program should be as efficient as possible – that means not repeating code and not specifying default arguments when possible. The main program must make use of your functions. You will be graded specifically on coding style for this part of the project.

1. Generate 4 passwords with the following specifications: length – 14, a minimum of 2 special characters, a minimum of 0 digits, a minimum of 1 uppercase character and no repeats. Special characters to choose from will be: *&@. These passwords should be stored in a list.
2. Print the list containing the 4 passwords you generated.
3. Check that each password in the list meets the specifications given in 1. You should print out a result that indicates that each password meets the criteria.

SUBMISSION EXPECTATIONS

Project_2.py :

Your implementations of the three functions specified in Parts 1, 2, and 3, including the corrected version of password_gen(). Do not change the names of parameters in the function def lines or the names of the functions – if you do so, it will affect your grade. **No additional code including input lines, print lines, and function calls should be submitted. Make sure that you have the correct import lines. If your code requires manual intervention to run such as adding/editing import lines, correcting indentation, removing print and input lines, etc., and/or correcting syntax errors, you can expect up to a 5 point deduction.**

Project_2_Main.py :

Your implementation of the main program specified in Part 4. Your function definitions must not be repeated in this file. They must be imported using the correct import line. **No additional code including input lines, print lines, and function calls should be submitted. Make sure that you have the correct import lines. If your code requires manual intervention to run such as adding/editing import lines, correcting indentation, removing print and input lines, etc., and/or correcting syntax errors, you can expect up to a 5 point deduction.**

Project_2.pdf :

A PDF document containing your reflections on the project. **You must also cite any sources you use. Please be aware that you can consult sources, but all code written must be your own. Programs copied in part or wholesale from the web or other sources will receive 0 points and will result in reporting of an Honor Code violation.**

POINT VALUES AND GRADING RUBRIC

Function style: 76.1

- password_specs(): 26.1 points
- password_gen() correction: 27.3 points
- check_password(): 22.7 points

Main program: 12 points -- NOTE: you will be graded specifically on programming style for the main program

Autograder tests: 9.4

Write-up: 2.5 points

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 as change 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