HW 6: Recursion, exceptions, and file processing

Due November 10th, 6pm, codePost.io
Please submit all files created for this assignment to codePost. You can submit as many times as you like up until the deadline. ***If you are planning to use one or more late days, please notify us by sending a private Piazza message to "Instructors".***

Familiarize yourself with how to approach a programming assignment and review the grading rubric on this assignment's page in Canvas before getting started.

Code style, documentation, and other requirements
All code style, documentation, and other requirements from the previous assignments apply to this one, too. You do not need to write a `main()` function for problem 1 in this assignment.

Tip for using try-except: It is safe to assume that functions will only be passed values of the correct data type. For example, if your function expects a list of strings as a parameter, you do not need to use try-except to verify that the supplied argument is in fact a list of strings. However, if you need to cast a value from one data type to another (e.g. converting a user-entered string to an int), you should not assume that the data type can be converted—use try-except to catch any exceptions that occur.

We will be checking that you are using GitHub regularly (at least 4 commits per homework). GitHub usage counts toward your professional skills score.

There is no written component for this assignment.

Programming Component (100 pts)

Problem 1: vowelsearch.py & test_vowelsearch.py

Write a **recursive** function called `contains_vowel` that, given a list of strings, returns True if every string in the list contains a vowel, and False otherwise. Here are some example inputs and outputs:

- `contains_vowel([“garage”, “this”, “man”]) => True`
- `contains_vowel([“ffff”, “this”, “man”]) => False`
- `contains_vowel([]) => False`

Make sure your function handles **any list of strings**, and returns the appropriate response. There are no restrictions on the string or list methods that you can use. You may also write any helper functions if you choose.

Don't let the short description fool you… this is a tricky problem. The key is identifying all possible input scenarios. The examples above are by no means exhaustive.

The codePost correctness tests total 24 points. Your function must be recursive in order to keep these correctness points. This means there should not be any loops in your solution.

Problem 2: recipes.py & test_recipes.py

Your task is to write a program that will enable users to save and read recipes. This problem will need a `main()` function. As usual, all input and print statements should ideally be in `main()`. However, with all the input validation that is required, you may find your `main()` function gets very long. It's OK to split your `main()` function into smaller functions for better readability but make sure you put as much of your program's logic into testable functions as possible.

Tip: Use VS Code's File > Open dialog to open the folder that you'll keep your code in. That way, files will be written in the same directory as your code and the program will also look in that directory when trying to read files. If you don't do this, VS Code will save and look for files wherever it's running from, which is usually the top level directory on your computer.

When the program starts, prompt the user to choose what they would like to do from this menu: `"MENU: 1 - Save a new recipe, 2 - Read a recipe, 3 - Quit "`.

Check that the user has provided valid input—either 1, 2, or 3. Don't assume they entered an integer! If their input is invalid, print `"Invalid choice."` then repeat the prompt until they get it right.

Saving a new recipe

Prompt the user to enter the recipe in stages using the following prompts (in order):

- `"Enter the ingredients on one line. Separate each ingredient with a comma. "`
- `"Enter the directions (1 paragraph only): "`
- `"Enter the time needed in minutes: "`
- `"Enter the name of the recipe: "`

Some of the input needs to be validated. Validation should happen immediately after the user responds to a prompt. For example, validate the ingredients before continuing to prompt for the recipe directions. Here are the validation requirements:

- Ingredients. Split the input string into a list at the commas and remove any leading or trailing whitespace from each item. For example, if the user enters `" 2 slices of bread , 2 tbsp peanut butter"`, it should be converted to `["2 slices of bread", "2 tbsp peanut butter"]`. The ingredient list must contain at least one non-empty string to be considered valid. For example, " " would be an invalid ingredient because, once leading / trailing white space is removed, it is an empty string. If the user does not provide at least one valid ingredient, print the message "`Recipe must have at least one ingredient.`" Prompt them to try again using the same prompt as before. Keep prompting until they provide valid input. No other validation is needed for the ingredient list.

- Time. A valid time is an integer greater than or equal to 0. Do not assume the user will enter an integer. If the user's input is invalid, print "`Invalid time. Must be an integer greater than or equal to 0.`" then prompt them to try again using the same prompt as before. Keep prompting until they provide valid input.
- Directions and recipe name. These inputs do not need to be validated. Anything is allowed, even an empty string.

When all input has been provided and validated, use the recipe name to create a filename, which you will later save the recipe to. To create the filename, convert the recipe name to lowercase, remove any leading or trailing whitespace, convert any other white space to underscores, then remove any remaining non-alphanumeric characters and add ".txt" to the end. If, after conversion, the filename is empty (or just ".txt"), you will need to ask the user to enter an alternative name. Print the message, "`Unable to create the filename.`" Then, use this prompt: "`Enter a string containing only letters, numbers, and spaces `". Use the user's new string (leave the original recipe name untouched) to create the filename. If the filename continues to be invalid, keep prompting until they get it right.

Now you are ready to write the recipe to file! Use the filename created in the previous step. If the file already exists, it should be overwritten. Write the recipe to the file using the following format:

- The first line of the file should be the recipe name as entered by the user.
- Leave a blank line (use the newline character, "`\n`").
- On the next line, write "`Ingredients:`"
- Leave a blank line
- Write each item in the ingredient list, one on each line.
- Leave a blank line
- On the next line, write "`Time: <user entered time> minutes`", where `<user entered time>` is the number entered by the user.
- Leave a blank line
- One the next line, write "`Directions:`"
- On the next line, write the directions provided by the user.

You can find example recipe files in the lecture-code repo > Starter code > HW 6 folder.

When the file has been written, print "`<recipe name> recipe saved to <filename>`", where `<recipe name>` is the recipe name provided by the user and `<filename>` is the file that it was saved to.

Prompt the user with the menu again.

Reading a recipe

Prompt the user to `"Enter the name of the recipe: "` they want to read. Note that the prompt is asking for the *recipe* name, not the filename. Use the same process used to create a file name in the section above to convert the user's input to a filename, and then attempt to open the file. If all goes well, print the complete contents of the file to the terminal. Otherwise, print "`Unable to print <filename>`", where `<filename>` is the name of the file you tried to open.

Prompt the user with the menu again.

Quitting

Allow the program to terminate.

Test all functions that you're able to test. You're not expected to test functions that perform file read/write operations.