Homework 4

Total Points: 100 (Correctness and Style)+ 3 EC (Checkpoint)

Due Dates:

- **Entire Assignment: Tuesday, February 6th, 11:59pm**
- **Checkpoint (read below): Sunday, February 4th, 11:59pm**

Starter Files

Download hw04.zip. Inside the archive, you will find starter files for the questions of this homework. You cannot import anything to solve the problems.

IMPORTANT: Coding and Docstring Style

This is a reminder that your code style will be graded. Here are a few useful links:

Style Guide Document

Style Guide on Course Website

Style Guide Examples

Testing

At any point of the homework, use one of the following command to test your work:

```
>>> python3 -m doctest hw04.py
>>> py -m doctest hw04.py
>>> python -m doctest hw04.py
```

Checkpoint Submission

Due Date: Sunday, February 4th, 11:59pm (SD time)

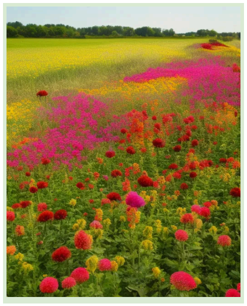
You can earn up to **3 points extra credit** by submitting the checkpoint by the due date above. In the checkpoint submission, you should complete **Question 1, 2** and submit the hw04.py file to gradescope.

Checkpoint submission is graded by completion, which means you can get full points if your code can pass some simple sanity check (no tests against edge cases). Note that in your final submission, you should still submit these questions, and you may modify your implementation if you noticed any errors.

General Notes and Requirements

1. **DO NOT IMPORT ANY PACKAGES.**
2. Please add your own doctests (**at least three**) as the given doctests are not sufficient. **You will be graded on the doctests**.
3. When a question requires **assert statements**: follow the assert guide to write assert statements to prevent any unexpected inputs. You will be graded on this. If a question does not require assert statements, assume all inputs are valid.

Required Questions



In Pythonia, you found there are many unknown plants and it is time to learn about them. Who knows, maybe you can learn how to use these plants to make your journey through the country more interesting?

Question 1:

Fortunately, you picked up a diary that may contain information about those plants. However the diary is poorly formatted and needs to be fixed before it can be used.

Write a function that takes in a path to a .txt file. The file may contain many paragraphs or blank lines. However, the following rules are certain:

1. The information about plants will always start from a header line in the format "Experiment Results:\n", **case sensitive, no trailing spaces.**
2. Following the header line will be the records you are looking for. The records are written in a fixed format: `name_of_plant`, `rating`, `one-word property`, separated either by **a comma OR a question mark**.

3. The end of records is marked by another line, “End\n”, **CASE INSENSITIVE, no trailing spaces**.

A plant may appear multiple times. Your function should return a dictionary with plant names as keys and a list of its properties as values.
Note:
1. The header line and the ending line will be always in the file and will never be part of the plant information. And they will only appear once.
2. **For each plant, assume each property will only appear once.**
3. If there is no plant records, return an empty dictionary
4. You can assume that there is **NO** blank line between “Experiment Results” and “End”
5. **No assert statements required**

For example:

In the file	Output	Explanation
I hate today’s breakfast! <blank line> Experiment Results: Harrada,12.3?Paralyze Harrada,50.5,Silence END <blank line> The functions are not healthy.	{'Harrada': ['Paralyze', 'Silence']}	Starting from “Experiment Results”, Harrada appeared twice with two properties revealed. “END” marked the end of the records. Other information is irrelevant.

Hints:

- **Think about how to locate the header line. Either .index() or a loop can be helpful**
- **.replace() may be helpful**

```python
def know_plant(path):
    """
    >>> know_plant('files/diary01.txt')
    {'Harrada': ['Paralyze', 'Silence']}
    >>> know_plant('files/diary02.txt')
    {'Harrada': ['Paralyze'], 'Flame Stalk': ['Invisibility']}
    """
```

Question 2:

After you have extracted the information about plants, you want to make potions (or poisons?...) from these plants.

Write a function `create_potion` that takes in a dictionary just like the one you created in Question 1 and a path to a .txt file, which will be generated by this function. Inside the file:

- every plant name in the dictionary should be written on a separate line (no trailing spaces).
- then add a line "Poison" if **more than** two plants in the dictionary have a "Paralyze" property, "Potion" otherwise. (no trailing spaces).
- On the last line, list all the properties mentioned in the dictionary, separated by a comma. (no trailing spaces).
    - If a property is mentioned multiple times, only record its first appearance. That is, **don't repeat the same property**.
- If the input dictionary is empty, only write "No Output" in the file.

Note:

1. All string comparisons are **case sensitive**. Thus "Paralyze" will be different from "paralyze".
2. Properties are listed by the order of appearance in the dictionaries.
3. Your file should **NOT** have a blank line at the end.
4. For each plant, assume each property will only appear once (same as in Question 1).
5. **No assert statements required.**

For example:

Input	In potion1.txt	Explanation

`{'Harrada': ['Paralyze', 'Silence']}`
 	
`"files/potion1.txt"`
	Harrada
	Potion
	Paralyze,Silence
		Harrada is the plant used.
		Since there is only one plant with "Paralyze" property, the output is a "Potion".
		The last line includes all the properties mentioned. Note the last line does not end with a comma

```
def create_potion(plants, path):
    """
    >>> dict1 = {'Harrada': ['Paralyze', 'Silence']}
    >>> create_potion(dict1, "files/potion1.txt")
    >>> with open("files/potion1.txt", 'r') as f:
    ...     for l in f:
    ...         print(l.strip())
    Harrada
    Potion
    Paralyze,Silence
    >>> dict2 = {'Harrada': ['Paralyze'], 'Flame Stalk':
['Paralyze'],\
'Stone Flower': ['Paralyze']}
    >>> create_potion(dict2, "files/potion2.txt")
    >>> with open("files/potion2.txt", 'r') as f:
    ...     for l in f:
    ...         print(l.strip())
    Harrada
    Flame Stalk
    Stone Flower
    Poison
    Paralyze
    """
```

Question 3:

Recall that lambda functions are one-line functions that only have a return value, which means anything that can be fit in one line (including one-line list comprehensions!) can be written as a lambda function. They can be called just like regular functions. For example,

```
>>> add_each = lambda lst: [x + 1 for x in lst]
>>> add_each([1, 2, 3])
[2, 3, 4]
```

Lambda functions can also take in multiple arguments just like regular functions. For example,

```
>>> foo = lambda n1, n2: n1 + n2
>>> foo(5, 2)
7
```

When we make potions, we want to give them names based on the plants that were used to create the potion. Write a function that takes a list of tuples, where each tuple contains the information about a plant: **(name, color, price)**:

1) Because we're on a budget, we want to get only the plants that are strictly **less than** $20.
2) If the number of plants that pass the price check is **even**, we get the **first three** characters of each plant name and put them together to create the potion's name.
3) If the number of plants that pass the price check is **odd** we get the **last three** characters of each plant name and put them together to create the potion's name.

Finally, convert the names so that the first character is capitalized and everything else is in lower case. Return an empty string if the list is empty.

Assumptions:

- You may assume all plant names have at least 3 characters.

Requirements:

- **Must use lambda and map and/or filter functions**
- **No explicit loops or list/set/dictionary comprehensions for the solution.**
- **Assert Statements Required For Input.** List comprehension is **allowed** for assert statements.

Assert Statement Hints:

1. Your function should only accept tuples with exactly 3 elements.
2. You may assume tuples with 3 elements will always be valid inputs. That is, assume the types of name, color, and price will always be valid.
3. Additional assert statements may be needed.

For example:

Input	Output	Explanation
`[('Harrada', 'yellow', 19), ('Flame Stalk', 'red', 13), ('Stone Flower', 'gray', 32), ('Crystal Snowflake', 'blue', 48), ('Foxglove', 'pink', 20), ('Dangerous Nightshade', 'purple', 18)]`	`'Adaalkade'`	Harr**ada**, Flame St**alk**, and Dangerous Nightsh**ade** are cheaper than 20, so the number is odd. The last three characters are taken from each of them to form the output.

```
def potion_name(plants):
    """
    >>> input_1 = [('Harrada', 'yellow', 19), ('Flame Stalk', 'red',
13), \
('Stone Flower', 'gray', 32), ('Crystal Snowflake', 'blue', 48), \
('Foxglove', 'pink', 20), ('Dangerous Nightshade', 'purple', 18)]
    >>> potion_name(input_1)
    'Adaalkade'

    >>> input_2 = [('White Snakeroot', 'white', 19), \
('Bitter Hogweed', 'black', 13)]
    >>> potion_name(input_2)
    'Whibit'

    >>> input_3 = []
    >>> potion_name(input_3)
    ''
    """
```

Question 4:

With potions in hand, we can trade them with others. Write a function that takes three inputs:

- **potions**: A list of strings representing potion names. These are the potions you have.
- **prices**: A list of non-negative integers representing the prices of potions. It has the same length as **potions.**

- **consumer_list**: a list of strings representing potions demanded by the consumer. **You should only consider those in your inventory.**

The consumer would like to buy EXACTLY TWO DISTINCT potions. That is, you cannot buy one potion twice. Your function should return the price of the most expensive combination.

Assumptions:
- There is at least one possible combination included in `consumer_list`
- Potion names are unique.

Hints:
- zip() may be useful. It allows you to combine two iterables into one.
- remove() may be helpful

Requirements
1. **Must use lambda and map and/or filter functions**
2. **No explicit loops or list/set/dictionary comprehensions**
3. **Assert Statements Required:**
    a. Only assert input **types**
    b. Do not worry about things like empty inputs, negative prices, no possible combinations.

```
def trade(potions, prices, consumer_list):
    """
    >>> trade(['health', 'magic'], [10, 10], ['health', 'magic'])
    20
    >>> trade(['health', 'magic', 'large_health'], [10, 20, 30], \
    ['health', 'large_health', 'magic'])
    50
    >>> trade(['health', 'magic', 'large_health', 'large_magic'],\
    [5, 10, 15, 20], ['magic', 'health', 'large_health'])
    25
    """
```

Question 5:

With multiple potions or poisons in hand, we can do a couple of operations with them.

1. **mix**: add potions names together and then reverse it. Return a string.

2. **reduce**: for each potion in a list, keep only the first 3 characters from potion names. If there are less than three characters, just use all characters. Return the modified list.
3. **eliminate:** remove all potions with names longer than 6 characters from the list. Return the new list.

Write a function that takes a list of potion names (string) and a list of operations (string) described above ("mix", "reduce", "eliminate"). Your function should execute the operations **in order**.

Assumptions:

- The order of operations is valid. For example, 'mix' will always be the last step.

Requirements:

1. You **must** fill the dictionary of `{name: lambda function}`, at the start of the given code. To call a function `name`, simply use `d[name](...)` (think about why this is the case).
2. You **cannot** use list comprehensions for all lambda functions. **Must use lambda and map and/or filter functions**
3. **Explicit loops are ALLOWED outside the given dictionary.**
4. **Assert Statements Required For Input.** List comprehension is **allowed** for assert statements.

Assert Statement Hints:

1. Your function should not accept empty list.
2. Your function should not accept invalid operations not mentioned above. This check is **case sensitive.**
3. Additional assert statements may be needed.

```
def potion_magic(potions, operations):
    """
    >>> potion_magic("incorrect input", ['mix'])
    Traceback (most recent call last):
    ...
    AssertionError

    >>> potion_magic(['health', 'magic'], ['eliminate', 'reduce'])
    ['hea', 'mag']
    >>> potion_magic(['health_large_potion', 'health'], ['eliminate',
'mix'])
```

```
    'htlaeh'
    """
```

Submission

When submitting to Gradescope, you can include the files folder by compressing it with your hw04.py file, creating a .zip file you can submit that zip file to Gradescope. If you have your hw04.py and files folder inside of another folder, make sure that you select hw04.py and the files folder directly to compress it, rather than compressing the encompassing folder, as it will cause issues with your submission.

You can do this by going into File Explorer or Finder and selecting **both** the files folder and the hw04.py file. Right-click and then select either

Windows: Send to -> Compressed (zipped) folder
Mac: Compress 2 items

Your .zip file should have the following files if you open it:

```
files/
--- diary01.txt
--- diary02.txt
--- ... (other files you create)
hw04.py
```

Please submit the homework via Gradescope. You may submit more than once before the deadline, and only the final submission will be graded. Please refer to the Gradescope FAQ if you have any issues with your submission.