CSCI 150: Exam 3 – Take Home

Due by 8:10am
Wednesday, April 19

Please turn in a *single* `.py` file, named "Exam3Lastname.py" to the assignment on Teams

Use comments (`#`) to indicate which problem is which. **IMPORTANT:** Please use the same names for the functions and the same order for the parameters as we do in the problems. It makes it significantly easier for us to grade. Thank you!!

The *only* resources you may use are the following:

- any code you have created for class, including homework and labs
- any notes you have taken during class
- code and any other material your instructors have posted to either the lecture or lab Teams pages
- any information directly linked from the class homepage, `https://hendrix-cs.github.io/csci150/`
- anything in the official Python documentation, `https://docs.python.org/3/`

You may not talk to a classmate, friend (real-life or Facebook), Siri, or anyone other than me about this exam until you turn it in, nor search the Internet or library or any reference other than those listed above for assistance. You may not even mention anything about how long it took you to complete the exam, that you found problem #2 particularly difficult (or easy), or in fact talk at all about the exam or Computer Science with anyone other than your instructor from Monday, 8:10am–Wednesday, 8:10am. Anyone who *gives* answers is equally in violation of the Academic Integrity Policy as one who *receives* them. All suspected violations will be reported to that committee.

1. Write a function `value_finder` which takes in two parameters: a dictionary `d`, keyed on strings with value a list of integers, and an integer `n`. The function should return a list of all keys (if any) for which `n` appears in its associated value.

   For example:

   - If `d1 = {'a': [2, 6, 4], 'b': [1, 2], 'c': [], 'd': [2, 4, 7, 8]}`, then
     - `value_finder(d1, 4)` should return `['a', 'd']`
     - `value_finder(d1, 2)` should return `['a', 'b', 'd']`
     - `value_finder(d1, 3)` should return `[]`
   - If `d2 = {'abc': [1, 2, 3, 4], 'a': [1], 'xyz': [-3, -7, 1], 'hi': [2, -7, 8, 1]}`, then
     - `value_finder(d2, 4)` should return `['abc']`
     - `value_finder(d2, -7)` should return `[xyz', 'hi']`
     - `value_finder(d2, 1)` should return `['abc', 'a', 'xyz', 'hi']`

2. Write a function `string_score` which takes in a string `s` which could contain upper or lower case characters, punctuation, spaces, or other special characters. Each of the alphabetic letters has a score, which is given to you by a **dictionary, posted on the class Team page on the Take-Home Assignment**. You should simply copy this dictionary directly into your `.py` file. For each given string, you should generate a score, an integer value, which corresponds to the sum of the scores of each letter in the string. (Since the string might contain non-alphabetics, those will simply be skipped.) Note that upper and lower case letters should both be counted ('HI', 'hi', 'Hi', and 'hI' should each return the same score of 5.)

   For example:

   - `string_score('scrabble')` returns `14`
   - `string_score('Hendrix College')` returns `28` (notice that the space does not contribute to the score)
   - `string_score('hello')` returns `8`
   - `string_score('HELLO')` returns `8`
   - `string_score('Hello')` returns `8`
   - `string_score('Hello!!!')` also returns `8`
   - `string_score('%735)*&$')` returns `0`, since none of the characters are assigned a score.

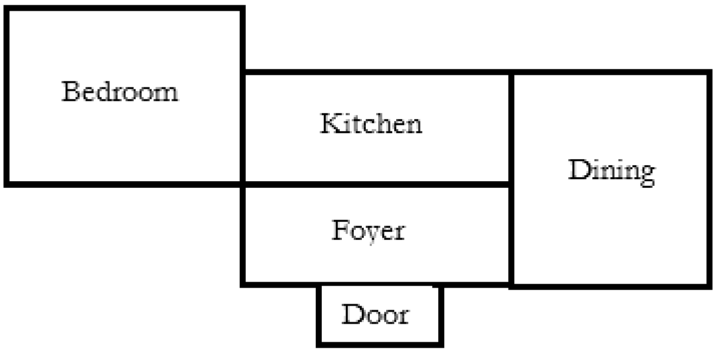
3. Consider using a dictionary `d` to represent a creepy abandoned mansion. The keys of the dictionary are the names of the rooms, and the values are lists of rooms adjacent to the key room. Write a function `way_out` which takes two parameters: a dictionary `d` representing a mansion, as described on the previous page, and a list of room names `lst`. This function should return `True` if the list represents a valid way to navigate through the house from the first room in the list to the `'Door'` room, and `False` otherwise.

   For example:

   The dictionary

```
mansion1 =\
{ "Foyer" : ["Kitchen", "Dining", "Door"] , "Dining" : ["Foyer", "Kitchen"],\
"Kitchen" : ["Bedroom", "Dining", "Foyer"], "Bedroom" : ["Kitchen"],\
"Door" : ["Foyer"]}
```

   represents a house that looks like this:



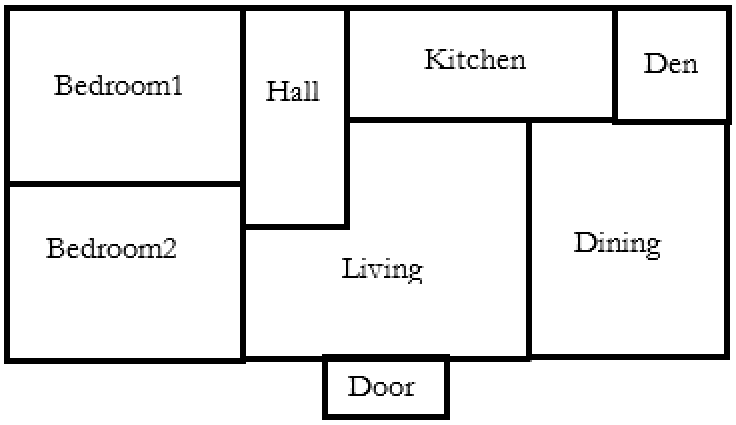
   - `way_out(mansion1, ["Dining", "Kitchen", "Foyer", "Door"])` should return `True` since you can go from the dining room directly to the kitchen, then to the foyer, and then to the door.
   - `way_out(mansion1, ["Dining", "Bedroom", "Foyer", "Door"])` would return `False` since it is not possible to go directly from the dining room to the bedroom
   - `way_out(mansion1, ["Dining", "Kitchen", "Bedroom", "Kitchen"])` would also return `False`, since, though this is a valid path through the mansion, it does not end at the door.

   (*problem continues on next page*)

The dictionary

```
mansion2 =\
{ "Bedroom1" : ["Bedroom2", "Hall"] , "Bedroom2" : ["Bedroom1", "Hall", "Living"],\
"Den": ["Dining", "Kitchen"], "Dining": ["Den", "Kitchen", "Living"],\
"Hall": ["Bedroom1", "Bedroom2", "Kitchen", "Living"],\
"Kitchen": ["Den", "Dining", "Hall", "Living"],\
"Living" : ["Bedroom2", "Dining", "Door", "Hall", "Kitchen"],\
"Door" : ["Living"]}
```

represents a house that looks like this:



- `way_out(mansion2, ["Den", "Kitchen", "Hall", "Bedroom2", "Living", "Door"])` should return `True` since this is a path out of the mansion.
- `way_out(mansion2, ["Kitchen", "Hall", "Dining", "Living", "Door"])` would return `False` since it is not possible to go directly from the hall to the dining room
- `way_out(mansion2, ["Dining", "Kitchen", "Hall", "Bedroom2"])` would also return `False`, since, though this is a valid path through the mansion, it does not end at the door.

IMPORTANT! Your code needs to work for any given mansion and path, not just the ones I have given as examples. You *do not* in any way need to construct the dictionary itself or the path – your function should simply evaluate a given dictionary and path to determine if it represents a valid way out.

4. Write a class `Store` which represents a (very simple) store which sells only one kind of item. Each `Store` has four attributes:

   - A boolean `is_open`, which is initially `False` when a `Store` object is created
   - An integer `capacity` which is set by a parameter when a `Store` is created (you can assume this will always be a positive number)
   - An integer `stock` which represents the current number of items in stock. Initially is set to the same value as `self.capacity`.
   - An integer `total_sold` which is the total number of items ever sold by the store. Initially set at 0.

   There will be a few methods:

   - `open_store()` which will set the value of `self.is_open` to `True`
   - `close_store()` which will set the value of `self.is_open` to `False`
   - `sell_item()` which will sell exactly one item (and therefore reduce the stock). However, if the store is closed, nothing is sold and a message is printed out to the user; likewise, if the value of `self.stock` is 0, nothing can be sold, and a (different) message should be printed. Finally, notice that when an item *is* sold, this should also increase the value of `self.total_sold` by 1.
   - `restock()` should reset the value of stock to the capacity; this can be used regardless of whether the store is open or closed.
   - `status()` should print whether the store is open, the number of items in stock currently, and the total number ever sold. Something like:
     `"The store is closed. It has 6 items in stock and has sold a total of 47 items."`

   Consider the following transcript:

```
>>> s = Store(4)
>>> s.close_store()
>>> s.open_store()
>>> s.sell_item()
>>> s.status()
The store is open. It has 3 items and has sold a total of 1 items.
>>> s.close_store()
>>> s.status()
The store is closed. It has 3 items and has sold a total of 1 items.
>>> s.sell_item()
The store is currently closed and cannot sell.
>>> s.restock()
>>> s.status()
The store is closed. It has 4 items and has sold a total of 1 items.
>>> s.open_store()
>>> s.sell_item()
>>> s.sell_item()
>>> s.sell_item()
>>> s.status()
The store is open. It has 1 items and has sold a total of 4 items.
>>> s.sell_item()
>>> s.status()
The store is open. It has 0 items and has sold a total of 5 items.
>>> s.sell_item()
The store does not have any items to sell right now.
```

```
>>> s.status()
The store is open. It has 0 items and has sold a total of 5 items.
>>> s.sell_item()
The store does not have any items to sell right now.
>>> s.restock()
>>> s.sell_item()
>>> s.status()
The store is open. It has 3 items and has sold a total of 6 items.
```

5. Write a class `character` which will represent a character in a (very simple) game. Each `Character` two attributes:

   - An integer `health` which always has value 20 when a character is created
   - An integer `strength` which is set as an integer parameter on character creation

   There will be a few methods:

   - `heal()` which increases a character's health by 5; however, if a character's health ever reaches 0, that character is dead, and cannot heal. A message should be printed out to the user in this case.
   - `attack(c)` where `c` is the variable name assigned to another character. The attack wil do damage (i.e. reduce character `c`'s health) by whatever the attacking character's `strength` is; however, again, a character with 0 health cannot attack, since they are dead. If an attack reduces the other character's health to or below zero, `c.health` should be reset to `0` and a message should be printed out to the user; finally, you cannot attack a dead character. If you try to a message should be printed.
   - `condition()` should print out a message like `"This character currently has a health of 7"` or whatever is appropriate; if the character is dead, this should be printed instead: `"This character has died."`

   Consider the following transcript:

```
>>> a = Character(3)
>>> b = Character(9)
>>> c = Character(6)
>>> a.condition()
This character currently has a health of 20.
>>> c.heal()
>>> c.heal()
>>> c.condition()
This character currently has a health of 30.
>>> b.attack(a)
>>> b.attack(a)
>>> a.condition()
This character currently has a health of 2.
>>> c.attack(a)
You killed them!
>>> a.condition()
This character has died.
>>> a.heal()
This character is dead and cannot be healed.
>>> b.attack(a)
That character is dead; you cannot attack them!
>>> a.attack(b)
You are dead and cannot attack!
>>> b.condition()
```

```
This character currently has a health of 20.
>>> c.attack(b)
>>> c.attack(b)
>>> c.attack(b)
You killed them!
```