CS 112 – Fall 2020 – Programming Assignment 7

Multi-Dimensional Lists

Due Date: Sunday, October 18th, 11:59pm

Make sure to abide by the instructions given in the *Assignment Basics* file, found in Blackboard under Assignments.

Background

The purpose of this assignment is to practice building, inspecting, and modifying multi-dimensional lists effectively. This often requires nested for-loops, but not always. It involves thinking of the structure like an *M*x*N* matrix of labeled spots, each with a row and column index, or a higher dimensional structure like a 3D matrix (e.g. *M*x*N*x*K*) that has indexes for height, width and depth. Multi-dimensional lists are not conceptually more difficult than single-dimension lists, but in practice the nested loops, aliasing, and more complex traversals and interactions merit some extra practice and thought.

Guidelines

- You are **not** allowed to `import` anything.
- You are **not** allowed to use `sets` and `dictionaries`
- You are **not** allowed to use anything that hasn't been covered in class, including the *list comprehension construct* (if you don't know what this is, don't worry, it's impossible to use it by accident!)
- You may use `continue, break`
- From built-in functions, you are only allowed to call `range()`, `len()`, `int()`, `str()`
- From string methods, you are only allowed to use `.split()` `.lower()` and `.isdigit()`
- From list methods, you are allowed to use only `.append()` and `.insert()`. Please **do not ask on piazza** whether you can use `.sort()`, `.remove()`, `.pop()` `.count()` etc.
- You are **not** allowed to use **slicing** in any form.
- The **only** allowed method to remove an item from a list is the `del` operator.
- You are **not** allowed to hard-code exhaustive `if-elif-elif` statements to handle all possible list lengths. Your code should work with any list length as specified by the problem.

Testing

In this assignment testing will be done as before. You will start working on the assignment without a tester, and you will do your own testing based on the examples we provide in this document as well as other examples you can come up with. A few days later we will provide an actual tester but **don't wait** for it in order to start working on the assignment as there won't be enough time to complete it before the deadline. Remember, programming assignments are a WEEK-long project; they are designed to not be completed in just one day or two. The purpose of the delayed release of the tester is for you to put more emphasis/effort on writing logically correct programs instead of trying to pass certain tests only. When we post the tester, we are going to omit some of the test cases that will be used for grading. This means that there might be errors in your code even if the tester is giving you no errors. You must do your own checks to make sure that you have not missed anything, and your code is correct. You do ***not*** need to modify the tester, just test on your own any way you like. Again, the goal is to help you put more focus on writing logically correct programs instead of trying to pass certain tests only.

Grading Rubric

Submitted correctly:	2	# see assignment basics file for file requirements!
Code is well commented:	8	# see assignment basics file for how to comment!
Tester calculations correct:	90	# see assignment basics file for how to test!
TOTAL:	100	

Note: If your code does not run and crashes due to errors, it will receive **zero** points. Turning in running code is essential. Running code, in this context, does not mean your code must work 100% or you receive a 0; it means, your code might be incorrect, but it at least compiles and runs.

Functions

The signature of each function is provided below, do ***not*** make any changes to them otherwise the tester will not work properly. The following are the functions you must implement.

create_3DFill(info)

Description: This function creates a list of list of lists (i.e 3D List). Given the dimensions of a 3D list and the value to be inserted; this function will create and return the specified 3D list.

Parameter: A string written in the format of: "(value to insert) | row x col x depth".

Return value: list of list of list

Example:

print(create_3DFill("(hi) | 2 x 4 x 3"))
*#notice how () around hi is **not** inserted into the matrix*
↓
[[['hi', 'hi', 'hi', 'hi'], ['hi', 'hi', 'hi', 'hi']], [['hi', 'hi', 'hi', 'hi'], ['hi', 'hi', 'hi', 'hi']], [['hi', 'hi', 'hi', 'hi'], ['hi', 'hi', 'hi', 'hi']]]

print(create_3DFill("(1) | 1 x 2 x 2")) → [[[1, 1]], [[1, 1]]]

print(create_3DFill(" (*) | 3 X 2 X 2"))
#notice how extra spaces and capital X does not matter, and the call still works
↓
[[['*', '*'], ['*', '*'], ['*', '*']], [['*', '*'], ['*', '*'], ['*', '*']]]

When it is valid:
When the input is passed to the 3DFill function, the format is always:
"(value to insert) | row x col x depth"

However, if there are various spacing within that format, that input **should still be valid**:
For example:
" (value) | row x col x depth" or **" (value) | r x col x d "**

Hint: What can you use to get rid of the spaces all together so you can get (valuetoinsert)|rowxcolxdepth?

If the x in the dimension is either (lowercase) x or (capital) X, the input **should still be valid**:
" (value to insert) | row X col x depth" or **"(value to insert) | row X col X depth "**

Side note: By the way, if the input is "((hi)) | 2 x 3 x 4", then you will be inputting (hi) into the 2x3x4 list.

When it is not valid and when your function should return an empty list:
- If one or both of the parenthesis is missing i.e **"value) | 2 x 2 x 2"** or **"value|2x2x2"**
- If the | is missing i.e **"(value) 2 x 2 x 2"**
- If one or both x is missing i.e **"(value) | 2 x 2 3"** or **"(value) | 2 2 2"**
- If the dimensions are not valid numeric values i.e **"(value) | @x b x 3"** or **"(value) | hello"**
- If any of the numeric values are negative i.e **"(value) | -2 x 3 x 4"**
- If all three values of the dimension are NOT given. i.e **"(value) | 2 x 2"** or **"(value) | 2"**

In the case that all three values of the dimensions <u>are</u> given but there are cases of 0:
One 0:
- "(value) | 2 x 2 x 0" is valid because we still have row and col, just do not have depth. This will still allow us to return a 2D list.
- "(value) | 2 x 0 x 2" is not valid because we do not have enough information, col information is missing. This will return an empty list.
- "(value) | 0 x 3 x 2" is not valid because we do not have the row information. Return empty list.

Two 0s:
- In this case, the **ONLY** valid input is the below and all other values other than 1 **is invalid**:
  "(value) | 1 x 0 x 0" **"(value) | 0 x 1 x 0"** **"(value) | 0 x 0 x 1"** and this returns **value**.

categorize_info(lst)

Description: This function creates a list of lists. This function takes in a list (Mx3) of demographic information, namely: a person's name, their age and their undergraduate year status; the function then categorizes this list into a list of names organized by age and year status.

Parameter: A Mx3 list containing demographics information of a person's name, age, and undergraduate year status (i.e freshman, sophomore, junior, senior).

Return value: A list containing the categorized values based on age and year.

Assumptions:
We will assume that the input list always contains valid values i.e name is an actual name, age is an actual age, year is an actual year (always). However, the strings might be upper case or lower case or a combination of the two. i.e Senior, senior or SeNiOR
The input value is always in the order of:
[[name, age, year], [name, age, year], [name, age, year]…

Example:

x = [["jane", 18, "freshman"], ["joe", 18, "senior"], ["sue", 20, "junior"], ["ahmed", 18, "freshman"], ["maryam", 22, "senior"]]

print(categorize_info(x)) →

[[[18, 'jane', 'joe', 'ahmed'], [20, 'sue'], [22, 'maryam']], [['freshman', 'jane', 'ahmed'], ['senior', 'joe', 'maryam'], ['junior', 'sue']]]

x = [["Moe", 17, "Freshman"], ["Tian", 17, "Sophmore"], ["Bill", 20, "Sophmore"], ["Karen", 25, "Senior"]]

print(categorize_info(x)) →

[[[17, 'Moe', 'Tian'], [20, 'Bill'], [25, 'Karen']], [['Freshman', 'Moe'], ['Sophmore', 'Tian', 'Bill'], ['Senior', 'Karen']]]

drawMe_five(n)

Description: This function creates a nxn list (square list). We will assume n is always an odd number greater than or equal to five (no need to check for it). This function will create a nxn matrix filling in the relevant index positions with either "*" or ".", creating a drawing of the number 5.

Parameter: an integer value n (we will assume n is always odd and will always be >= 5).

Return value: a nxn matrix containing "*" or "." in corresponding positions, drawing a 5.

Example:

```
a = drawMe_five(9) #creates 9x9

for i in range(len(a)):
    for j in range(len(a[i])):
        print(a[i][j], end = ' ')
    print()
```

→

```
. . * * * * * * .
. * . . . . . . .
. * . . . . . . .
. * . . . . . . .
. * * * * * * . .
. . . . . . . * .
. . . . . . . * .
. . . . . . . * .
. * * * * * * . .
```

```
a = drawMe_five(5)  #creates 5x5

for i in range(len(a)):
    for j in range(len(a[i])):
        print(a[i][j], end = ' ')
    print()
```

→

```
. . * * .
. * . . .
. * * . .
. . . * .
. * * . .
```

find_Battleship(lst, guess)

Description: This function returns a string. This function takes in a MxN matrix and a 1D list. The MxN matrix contains the board for a battleship game and the 1D list contains a guess of what type of ship and what location we believe the battleship or part of the battleship to be. This function will check the guess and will return one of three string values: “it’s a hit!” or “it’s a miss!” or “it’s a hit, but not that ship”.

Assumptions: The MxN matrix and the 1D list will always contain valid values. The MxN matrix will always contain just one of each ship type (no duplicates).
The ship types are the following:
Carrier = represented by an X (takes up one space on the board)
Cruiser = represented by XX, always horizontal (takes up two spaces on the board)
Destroyer = represented by XXX, always vertical (takes up three spaces on the board)
There is no overlapping and the X value of one ship cannot be also part of another ship. There is always at least one space between Xs of different ships on the same row and same column.

Parameter: An MxN matrix containing the battleship game board information. A 1D list containing a guess for a specific ship and where it or part of it might be. The name of the ship could be passed in as all lower case, or all upper case or a combination of the two.

Return value: one of three string values: “it’s a hit!” or “it’s a miss!” or “it’s a hit, but not that ship”. [make sure the spelling and upper/lower casing and punctuation of your return string is exactly like what you see here].

Example:

```
b = [["X", "*", "*", "*", "*"],
     ["*", "*", "*", "*", "X"],
     ["X", "X", "*", "*", "X"],
     ["*", "*", "*", "*", "X"]]

g = ['Carrier', 2, 1]    #the guess input order is always: name of ship, row guess, col guess
result = find_Battleship(b, g)
print(result)                                →        it's a hit, but not that ship

g = ['cruiser', 2, 0]
result = find_Battleship(b, g)
print(result)                                →        it's a hit!

g = ['DestroYer', 3, 2]        #notice the lower/upper case use for the word does not matter
result = find_Battleship(b, g)
print(result)                                →        it's a miss!
```