CMSC 201 Spring 2019

Lab 09 – 2D Lists

Assignment: Lab 09 – 2D Lists
Due Date: **During discussion**, April 1st through April 4th
Value: 10 points (8 points during lab, 2 points for Pre Lab quiz)

This week's lab will put into practice the new concepts you learned about lists: lists of lists (two-dimensional lists), mutability, and passing lists to functions.

(Having concepts explained in a new and different way can often lead to a better understanding, so make sure to pay attention as your TA explains.)

Part 1A: Review – Traversing Lists

Remember from previous assignments that we can access the items in a list sequentially using a **`while`** loop. This process is called ***traversing*** and is often used when we want to perform the same action with each element of a list. We can use an index as our loop variable, and increment its value after each loop. In the body of the **`while`** loop we can include any code that we want to run based on the value of each element of the list.

In this example from Pre-Lab 5, the given code would traverse the **`names`** list below, printing out that each person is awesome:

```
# this variable can be called anything
# it starts at zero because that's the first index
index = 0
while index < len(names):
    print( names[index], "is awesome!")
    index += 1
```

Part 1B (Review) – Visualizing 2D Lists

In Python, two-dimensional lists (or 2D lists, as we'll call them from now on) are simply lists of lists. We've worked with lists of integers, lists of strings, and other variable types – a list of lists isn't much different.

If we visualize the 2D list, we can think of each "sublist" as a row of a grid and each element of a "sublist" as a column in that row.

We can initialize a 2D list as follows:

```
my2DList = [ [13,18,21], [30,40,50], [77,88,99] ]
```

We can draw the list as a table or matrix as shown below. (Note that the gray boxes represent the indices of the list, and are not part of the list in any way.)

	0	1	2
0	13	18	21
1	30	40	50
2	77	88	99

Part 1C (Review) – Accessing 2D Lists

We access 2D lists in a way very similar to the way that we access lists in general. The general format to access an element of a 2D List is:

```
listName[row][column]
```

For example, if we wanted to access the value 21 in the previous example, we would use `my2DList[0][2]`.

Notice that the row index always comes before the column index. If we think about how we accessed the value 21 above:

(1) We look at the element of the "outside" list that is at index 0
- Because each element is a list itself, `my2DList[0]` will be the first row (or "sublist") of the 2D list

(2) We can then index into the selected "inside" list and access its 3rd element by using the index 2: `my2Dlist[0][2]`
- This gives us the third element of the first list, the value 21

We already know that we can use `len()` to determine the number of items in a one-dimensional list. However, this means that calling `len(my2Dlist)` will return the number of rows in `my2DList`, not the total number of items. This is because the elements of `my2DList` are the "sublists" of the "outside" list, not each individual element.

If we want the number of items in an individual row, we would use `len(my2Dlist[rowNum])` instead, because `my2DList[rowNum]` is an "inside" list (or "sublist") and will therefore be a one-dimensional list of individual elements.

Part 1D (Review) – Mutating 2D Lists

We can change the contents of a list in two ways:

(1) Assigning a new value to a current element of a list.
(2) Appending to or removing from a list.

To update the contents of a list, we simply assign a new value to a specific index. For example, we can run the following line of code on the list below:

```
my2DList[0][2] = 25
```

	0	1	2
0	13	18	21
1	30	40	50
2	77	88	99

After this code is run the list looks like this (the change is highlighted for you):

	0	1	2
0	13	18	**25**
1	30	40	50
2	77	88	99

We can also use **`append()`** and **`remove()`**, similar to how we used them for one-dimensional lists. We can call these functions on the entire list to allow us to add or remove a row. We can also call them on the list at a specific index (a specific row) to add or remove from that row of the list. It is important to note that the rows of a 2D list do not need to be the same length.

For example, the following is a valid definition of a 2D list:

```
myUnbalancedList = [ ["A", 1], [2.3, 5.6, "hello"], \
                   ["pizza"] ]
```

	0	1	2
0	`"A"`	1	
1	2.3	5.6	`"hello"`
2	`"pizza"`		

Also notice that like with one-dimensional lists, all elements of the list do not need to be of the same type. This makes sense if we remember that we are dealing with a list of lists.

Here are some examples of using **`append()`** and **`remove()`**, on **`my2DList`**.

We last left our list like this:

	0	1	2
0	13	18	25
1	30	40	50
2	77	88	99

```
my2DList.append( [101, 121] )
```

This line of code will add a new row to the end of our list:

	0	1	2
0	13	18	25
1	30	40	50
2	77	88	99
3	**101**	**121**	

```
my2DList[1].remove(40)
```

This line of code will remove the value **`40`** from the second row (index 1). This causes the value **`50`** to shift over, and shortens the second row's length.

	0	1	2
0	13	18	25
1	30	**50**	
2	77	88	99
3	101	121	

Note that **`my2DList.remove(50)`** would <u>not</u> work on the list. There is no value **`50`** in the main list – it only exists within the second row.

In the same way, the membership "**`in`**" will only look one "level" deep. If we asked Python: **`13 in my2DList`** it would return **`False`**. We would need to ask **`13 in my2DList[0]`** for the result to be **`True`**.

Part 2: Exercise

In this lab, you'll be downloading a file and completing it by writing the code necessary to complete two functions that make use of a 2D list.

The program you'll be finishing uses a two-dimensional list to store a list of expenditures, with each interior list containing the spending for a specific category. Once complete, the program will print out the categories and expenditures, the total spent in each category, and the overall total expenditures.

Tasks

Starting:

- ☐ Copy the **`given_spend.py`** file from Dr. Gibson's **`pub`** directory
  - ☐ It should be renamed to **`spending.py`**
- ☐ Complete the file header comment at the top

Functions:

- ☐ Write the **`while`** loop to complete **`printSpending()`**
- ☐ Write the **`while`** loop for **`tallySpending()`**
- ☐ Write the function call for **`tallySpending()`**
- ☐ Uncomment the **`print()`** function call in **`main()`**

General:

- ☐ Run and test your code as needed
- ☐ Show your work to your TA

Part 3A: Downloading the File

First, create the **`lab09`** folder using the **`mkdir`** command -- the folder needs to be inside your **`Labs`** folder as well.

Next, copy a file into your **`lab09`** folder using the **`cp`** command.

```
cp /afs/umbc.edu/users/k/k/k38/pub/cs201/given_spend.py spending.py
```

This will copy the file **`given_spend.py`** from Dr. Gibson's public folder into your current folder, and will change the file's name to **`spend.py`** instead.

The first thing you should do in your file is complete the file header comment, filling in your name, section number, email, and the date.

Part 3B: Finishing the Functions

At this point, if you try to run the file, it will only print out the category titles for the expenditures, and nothing else. That is because the file is only partially completed for you.

You will need to update the file to complete the two function definitions, and one function call. If you open the file, you should see comments boxed in by **#** signs – these are where you need to write new code. Read the function header comments to see the details about the two functions.

You'll need to **think carefully** about the two-dimensional **`spending`** list and how its contents are organized (and how you can access them). It might be helpful to draw a visual representation, and to label the indexes.

For **`printSpending()`** and **`tallySpending()`**, you'll need to write a **`while`** loop. Make sure to **pay attention** to what the function's already written code does, as well as things like variable names.

Once both functions are complete, you need to call **`tallySpending()`** down in **`main()`**, as well as uncommenting the **`print()`** statement below it. You shouldn't need to write any other code.

(See the next page for sample output.)

Here is some sample output of the completed program.
There is no user input, and yours should be **identical** once it's complete.

(The printed list of expenditures for eating out is long enough to wrap around in this document, but is should not wrap around when you run it in the terminal.)

```
bash-4.1$ python spending.py

Spending:
---------------
EAT OUT :       15.86   42.79   13.8    9.16    9.16
5.42    49.3
VEHICLE :       25.68   89      7.99    23.11
GROCERIES :     54.33   30.27   5.88    40.71
UTILITIES :     165.31  17.32
PERSONAL :      25      14.91   40.52
ENTERTAIN :     15.66   13      5.99    33.04
PET CARE :      100     13.91   48.98   17.22   42.45
RENTING :       1300


In category EAT OUT     $145.49 was spent
In category VEHICLE     $145.78 was spent
In category GROCERIES   $131.19 was spent
In category UTILITIES   $182.63 was spent
In category PERSONAL    $80.43 was spent
In category ENTERTAIN   $67.69 was spent
In category PET CARE    $222.56 was spent
In category RENTING     $1300 was spent

You spent a total of    $2275.77 this month
```

Part 4: Completing Your Lab

Since this is an in-person lab, you do not need to use the **`submit`** command to complete your lab. Instead, raise your hand to let your TA know that you are finished.

They will come over and check your work – they may ask you to run your program for them, and they may also want to see your code. Once they've checked your work, they'll give you a score for the lab, and you are free to leave.

Starting:

- ☐ Copy the **`given_spend.py`** file from Dr. Gibson's **`pub`** directory
  - ☐ It should be renamed to **`spending.py`**
- ☐ Complete the file header comment at the top

Functions:

- ☐ Write the **`while`** loop to complete **`printSpending()`**
- ☐ Write the **`while`** loop for **`tallySpending()`**
- ☐ Write the function call for **`tallySpending()`**
- ☐ Uncomment the **`print()`** function call in **`main()`**

General:

- ☐ Run and test your code as needed
- ☐ Show your work to your TA

IMPORTANT: If you leave the lab without the TA checking your work, you will receive a **zero** for this week's lab. Make sure you have been given a grade before you leave!