CS 112 - Spring 2022 - Programming Assignment 5

Memory, Lists, and Other Collections

Due Date: Monday April 4, Noon (12 pm)

This assignment is an Individual Effort assignment. The Honor Code Applies.

The purpose of this assignment is to practice with aliasing, multi-dimensional lists, and other collections, including dictionaries.

See the "**assignment basics**" file for more detailed information about getting assistance, running the test file, grading, commenting, and many other extremely important things. Each assignment is governed by the rules in that document.

Needed files:
Download the attached file to use the tests we will use when grading your assignment

- `tester5.py`
- `PA5template.py`

Background

Understanding how Python handles memory is essential for building programs that can solve more significant problems. In completing the following tasks you will explore different ways that indirect variable types can be used effectively, and exercise your understanding.

General Restrictions

Any function which violates any of the following will receive zero points, regardless of the autograder.

- You are **not allowed** to **`import`** anything.
- You are **not allowed** to use slicing.
- You are **not allowed** to use the **`min`** or **`max`** functions.
- You are **not allowed** to use any string methods.
- You are **not allowed** to use anything that hasn't already been covered in class.
- Pay particular attention to any restrictions written in the task descriptions below.
- Make sure you comment your code where appropriate.

Testing

A template file (PA5template.py) is provided which includes the basic outline of the functions you are required to complete. The tester will be calling your functions with different arguments and checking the return values to decide the correctness of your code. **Your functions should not ask for user input and should not print anything**. When you run your code with the tester or submit your code to **Gradescope** you can see the results of the tests.

Note: if you want to perform your won't ests, you do not need to modify the tester. Simply call the functions you've written providing arguments you want to check. Be sure to remove any tests you've added before submitting.

Submitting your code:
Submit your .py file (correctly named) to Gradescope under **Programming Assignment 5**

- Link to Gradescope is in our Blackboard page under "Programming Assignments submit to Gradescope here"
- Do not forget to name your file netID_2xx_PA5.py, where netID is NOT your G-number but your GMU email ID and 2xx is your lab section number.
- Do not forget to copy and paste the **honor code statement** from the **Assignment Basics** document to the beginning of your submission .py file.
- Do not forget to comment your code
- **No hard coding!**

Grading Rubric:

- **5pts** - Correct Submission
- **5pts** - Well-Documented
- **40pts** - Calculations Correct

— — — — — — — — — — — —

Total: **50pts**

Note: if your code does not run (crashes due to errors), it will receive at most 12 points. No exceptions. As stated on our syllabus, turning in running code is essential.

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:

`def increment_attendance(seating, locations):`

Given a 2-D list **`seating`**, and a sequence of **`locations`**, increment the value at each location by one. The **`seating`** matrix can be any shape, but will contain only numbers. The **`locations`** consist of any number of **pairs** of indices, guaranteed to be valid indices into the seating matrix. This function should not return any value, and should modify the seating matrix ***in-place***.

Examples:

```
seats = [[0, 0, 0], [0, 0, 0]]         #two rows, three columns
locs = [(0,0), (1,2)]                  #top-left, and bottom-right
increment_attendance(seats, locs)      #no return value!
print(seats)                      #prints [[1, 0, 0], [0, 0, 1]]

seats = [[0,1], [2,3], [4,5]]          #three rows, two columns
locs = [(0,0), (1,0), (0,0), (2,0)]    #top-left twice
increment_attendance(seats, locs)      #no return value!
print(seats)                      prints [[2, 1], [3, 3], [5, 5]]

seats = [[1, 1, 1], [2, 2, 2], [3, 3, 3]]   #3x3
locs = [(1,1), (1,1), (1,1), (1,1), (1,1)]  #middle x5
increment_attendance(seats, locs)      #no return value!
print(seats)    #prints [[1, 1, 1], [2, 7, 2], [3, 3, 3]]
```

`def drop_lowest(scores, drop_number):`

Given a 2-D list of **`scores`**, and a list containing the number of low scores to drop (**`drop_number`**), remove the corresponding lowest scores from each row. The **`scores`** list can be any shape, but the **`drop_number`** list will have as many elements as the **`scores`** list has rows, and the number of low scores to drop will always be less than the total number of scores in each row. This function should not return any value, and should modify the **`scores`** list ***in-place***.

Examples:

```
scores = [[10, 9, 7, 8], [1, 1, 0], [50, 50, 50, 50], [75, 100]]
drop_number = [1, 2, 1, 0]
drop_lowest(scores, drop_number)       #no return value!
print(scores)    #prints
                 # [[10, 9, 8], [1], [50, 50, 50], [75, 100]]

scores = [[0, 50], [10, 9, 10], [85, 85], [0, 0, 0, 0]]
drop_number = [1, 1, 0, 2]
drop_lowest(scores, drop_number)       #no return value!
print(scores)    #prints
                 # [[50], [10, 10], [85, 85], [0, 0]]
```

`def organize_grades(grades, assignment_types,max_possible):`

Given a list of **`grades`** (numbers), a list of **`assignment_types`** (strings), and a list of the **`max_possible`** score for each grade, create and **`return`** a dictionary where each key is one of the assignment types, and each value is a list of the percentage grades in that assignment type. The assignment types can be any of the strings "`zy`", "`lab`", "`pa`", "`mid1`", "`mid2`", or "`final`". If no grades are given for one of these assignment types, the key-value pair in the returned dictionary should have an empty list for the corresponding key.

Examples:

```
grades = [10, 1, 50, 100, 100, 100]
atype = ['zy', 'lab', 'pa', 'mid1', 'mid2', 'final']
max_possible = [10, 1, 50, 100, 100, 100]
gbook = organize_grades(grades, atype, max_possible)
print(gbook)     #prints
# {'zy': [1.0], 'lab': [1.0], 'pa': [1.0], 'mid1': [1.0], 'mid2':
[1.0], 'final': [1.0]}

grades = [10, 9, 8, 7]
atype = ['zy', 'zy', 'zy', 'zy']
max_possible = [10, 10, 10, 10]
gbook = organize_grades(grades, atype, max_possible)
print(gbook)     #prints
# {'zy': [1.0, 0.9, 0.8, 0.7], 'lab': [], 'pa': [], 'mid1': [],
'mid2': [], 'final': []}

grades = [8, 45, 9, 41, 1]
atype = ['zy', 'pa', 'zy', 'pa', 'lab']
max_possible = [10, 50, 10, 50, 1]
gbook = organize_grades(grades, atype, max_possible)
print(gbook)     #prints
# {'zy': [0.8, 0.9], 'lab': [1.0], 'pa': [0.9, 0.82], 'mid1': [],
'mid2': [], 'final': []}
```

`def gbook_averages(gbook):`

Given a dictionary **gbook** containing assignment types (strings) as keys, and lists of grades (numbers) as values, **`return`** a **new** dictionary with the same keys, and with the average of each list of grades for the given key. The parameter **`gbook`** should **not** have its contents modified. The average can be computed as the sum of all values in the list divided by the length of the list.

Examples:

```
gbook = {'zy': [1.0], 'lab': [1.0], 'pa': [1.0], 'mid1': [1.0],
'mid2': [1.0], 'final': [1.0]}
avgs = gbook_averages(gbook)
print(avgs)      #prints
# {'zy': 1.0, 'lab': 1.0, 'pa': 1.0, 'mid1': 1.0, 'mid2': 1.0,
'final': 1.0}
```

```
gbook = {'zy': [1.0, 0.9, 0.8, 0.7], 'lab': [], 'pa': [], 'mid1':
[], 'mid2': [], 'final': []}
avgs = gbook_averages(gbook)
print(avgs)      #prints
# {'zy': 0.8500000000000001, 'lab': 0.0, 'pa': 0.0, 'mid1': 0.0,
'mid2': 0.0, 'final': 0.0}

gbook = {'zy': [0.8, 0.9], 'lab': [1.0], 'pa': [0.9, 0.82],
'mid1': [], 'mid2': [], 'final': []}
avgs = gbook_averages(gbook)
print(avgs)      #prints
# {'zy': 0.8500000000000001, 'lab': 1.0, 'pa': 0.86, 'mid1': 0.0,
'mid2': 0.0, 'final': 0.0}
```

def course_grade(gbook, weights, replace):

Given a dictionary **gbook** containing assignment types (strings) as keys with the average grade for that assignment type as the value, a dictionary **weights** with assignment types as keys and the course grade weights for each assignment type, and a dictionary **replace** where both keys and values are assignment types and represents whether a given assignment type (key) should replace an assignment type with a lower value (value), **return** a floating point number representing the final course grade, computed by combining the assignment averages with their corresponding weights, and replacing grades where appropriate.

Examples:

```
gbook = {'zy': 1.0, 'lab': 1.0, 'pa': 1.0, 'mid1': 1.0, 'mid2':
1.0, 'final': 1.0}
weights = {'zy': 0.05, 'lab': 0.1, 'pa': 0.4, 'mid1': 0.1,
'mid2': 0.1, 'final': 0.25}
replace = {'mid1':'final', 'mid2':'final'}
cgrade = course_grade(gbook, weights, replace)
print(cgrade)         #prints 1.0

gbook = {'zy': 0.8500000000000001, 'lab': 0.9, 'pa': 0.8, 'mid1':
0.85, 'mid2': 0.9, 'final': 0.8}
weights = {'zy': 0.05, 'lab': 0.1, 'pa': 0.4, 'mid1': 0.1,
'mid2': 0.1, 'final': 0.25}
replace = {'mid1':'final', 'mid2':'final'}
cgrade = course_grade(gbook, weights, replace)
print(cgrade)         #prints 0.8525

gbook = {'zy': 0.8500000000000001, 'lab': 1.0, 'pa': 0.86,
'mid1': 0.8, 'mid2': 0.9, 'final': 0.85}
weights = {'zy': 0.05, 'lab': 0.1, 'pa': 0.4, 'mid1': 0.1,
'mid2': 0.1, 'final': 0.25}
replace = {'mid1':'final', 'mid2':'final'}
cgrade = course_grade(gbook, weights, replace)
print(cgrade)         #prints 0.8815
```