ECS 32A: Programming Assignment #5

Instructor: Aaron Kaloti

Summer Session #1 2020

Contents

1 Changelog

You should always refer to the latest version of this document.

- v.1: Initial version.

2 Due Date

This assignment is due the night of Thursday, July 23. Gradescope will say 12:30 AM on Friday, July 24, due to the "grace period" (as described in the syllabus). *Do not rely on the grace period for extra time; this is risky.*

Some students tend to email me very close to the deadline. This is also a bad idea. There is no guarantee that I will check my email right before the deadline.

3 Reminder about Gradescope Active Submission

Once the deadline occurs, whatever submission is your **active submission** will be the one that dictates your final score. By default, your active submission will be your latest submission. However, you can change which submission is your active submission, which I talked about during the Thursday (6/25) lecture.

4 Manual Review

See the style guide on Canvas here. A small portion of your grade for this assignment will be decided by me or one of the TAs; one of us will manually check your code and check that you made a decent effort at following the study guide. Blatant violations that suggest you didn't even look at the study guide could very easily earn you a zero for the manual review portion of your score.

Do not ask (over email or office hours) questions such as, "Could you take a look at my code and tell me if I will get a 100% for the manual review portion?" You should, however, ask about specific components of the style guide that you are not sure about.

For this programming assignment, I am no longer requiring docstrings.

5 Grading Breakdown

This assignment is worth 9% of your final grade and will be worth 90 points on Gradescope (with 85 of those points being decided by the autograder). Here is the breakdown of the 90 points by part:

- Part #1: 10
- Part #2: 25
- Part #3: 10
- Part #4: 10
- Part #5: 15
- Part #6: 15
- Manual review (see above section): 5

6 Problems

All code should be placed in a file called `prog5.py`.

You are not allowed to `import` any modules.

Only the below methods are permitted (not all of them are necessary). Any method not listed below is prohibited. As mentioned in lecture, a method is a special kind of function that is called by specifying a value (usually a variable) followed by a dot/period and finally followed by the function call, e.g. `split()` is a method and can be called through something like `s.split()`, where `s` is a string.

- String methods: `format()`, `split()`.
- List methods: `append()`, `extend()`.
- Dictionary methods: `items()`, `values()`, `setdefault()`.

Beware of copy/pasting strings from this PDF. Because I created this PDF with LaTeX, the quotation marks (single and I think double as well) are stylized in a way such that if you copy/paste them into a Python program, they will not be recognized as quotation marks and thus will cause syntax errors.

Due to the complex nature of some of the inputs in this part, you may find it beneficial to be able to repeat previous commands. If you are using Python IDLE, then note that to the syllabus, I added directions on how to scroll to previous commands in the Python IDLE interpreter. The Mu editor unfortunately does not allow this, since you have to reset the entire interpreter session every time you make changes. However, you could always type certain inputs that you wish to keep using into some text file that you can copy/paste from.

Parts #1 and #2 do not require nested loops. For these two parts, if you find yourself wanting to use nested loops, make sure you are careful about it. I would strongly advise against it, because I think that using nested loops where they are not necessary tends to lead to hopelessly complex approaches.

None of the parts in this assignment require more than two layers of nesting. That is, you might have to use a loop that is within another loop, but you do not have to use a loop that is within a loop that is within yet another loop.

Unless otherwise specified (i.e. a specific kind of erroneous input and how to handle it is explained), you may assume the arguments passed in are always valid / as expected. For example, for part #3, you may assume that the caller will not pass in a list of lists of *strings* as the first argument.

6.1 Part #1: Count in Other

Write a function called `count_in_other` that takes as argument two lists. The function should return the number of elements in the second list that are in the first list.

Hint: If you make proper use of the `in` operator, then you should not have to use nested loops for this part.

Below are examples of how your function should behave. As shown in the fourth example, if the second list has non-unique elements that appear in the first list, then those elements each contribute to the count.

```
>>> count_in_other([3,8,5,2,1,-10,20],[2,8,30,14,-10])
3
>>> count_in_other([3,8,5,2,1,-10,20],[-10,50,5,2,12])
3
>>> count_in_other([3,8,5,2,1,-10,20],[-10,50])
1
>>> count_in_other([3,8,5,2,1,-10,20],[-10,-10])
2
```

6.2 Part #2: Buy Items

Write a function called `buy_items` that takes three dictionaries as arguments. Each of these dictionaries maps strings to integers. This function is meant to model the attempt of a consumer to buy something at a store. Below is a description of each argument.

- The first dictionary represents what a store has in stock. Each key is a string representing the name of an item in the store, and the corresponding value is the number of that item that is in stock in that store.
- The second dictionary represents what the consumer is trying to buy. Each key is a string containing the name of an item, and the corresponding value is the number of that item that the user is trying to buy.
- The third dictionary represents the prices of items. Each key is again an item name, and the corresponding value is the price of that item (in dollars). For the purposes of `buy_items()`, we are assuming that all prices are integers, not floating-point numbers.

Assuming that the arguments are "valid" (see below), `buy_items()` should return the total price that the consumer must pay. Consider the example below. In this example, the user wants 2 units of deodorant and a toothpaste. The store has 100 units of toothpaste and 20 units of deodorant, so the consumer's request is satisfiable. Thus, the user will purchase 2 units of deodorant, each 3 dollars, and 1 toothpaste, which is 1 dollar, spending a total of 7 dollars in the process.

```
>>> buy_items({"toothpaste": 100, "toilet paper": 1, "deodorant": 20}, {"deodorant": 2, "toothpaste": 1},
    {"toothpaste": 1, "deodorant": 3, "toilet paper": 3})
7
```

Below are situations in which the arguments are considered invalid and what the appropriate return values should be. The situations are shown below from highest to lowest priority, meaning that if the first and third situations both occur, for example, then the return value should be -1, not -3.

1. If at least one key-value pair in any of the three dictionaries has a nonpositive value, then `buy_items()` should return -1.
   - I recommend defining a helper function called `all_positive_values()` that takes as argument a dictionary and returns `True` if all values in the dictionary are positive and `False` otherwise.
2. If not every item in stock (i.e. every item that is mapped to something in the first dictionary) has a price (as given in the third dictionary), then `buy_items()` should return -2.
3. If the store does not have enough of an item that the user wants (or if the store does not have said item at all), then `buy_items()` should return -3.

Here is another example of valid arguments.

```
>>> buy_items({"diapers": 20, "pedialyte": 5, "shirts": 15, "televisions": 30, "ramen": 200}, {"pedialyte"
    : 2, "ramen": 3}, {"diapers": 4, "pedialyte": 6, "shirts": 10, "televisions": 150, "ramen": 2})
18
```

Here are examples involving invalid arguments.

```
>>> # In the below example, item C has no price.
>>> buy_items({"item A": 1, "item B": 2, "item C": 3}, {"item B": 1}, {"item A": 15, "item B": 10})
-2
>>> # In the below example, item C is requested but not in stock.
>>> buy_items({"item A": 1, "item B": 4}, {"item A": 1, "item C": 1}, {"item A": 3, "item B": 10})
-3
>>> # In the below example, the store doesn't have enough of item B.
>>> buy_items({"item A": 5, "item B": 5}, {"item A": 3, "item B": 6}, {"item A": 1, "item B": 2})
-3
>>> # In the below example, the third dictionary has a nonpositive value in one of the key-value pairs.
>>> buy_items({"item A": 5, "item B": 1}, {"item B": 1}, {"item A": 0, "item B": 1})
-1
```

Hint: As with part #1, if you make proper use of the `in` operator, then you should not have to use nested loops for this part.

6.3 Part #3: Count Less Than

Write a function called `count_less_than` that takes as its two arguments a list of lists of integers and an integer k. `count_less_than()` should return the number of integers in the first argument that are less than k *unless* at least one of the integers in the first argument is 0, in which case `count_less_than()` should return -1.

Below are some examples of how your function should behave.

```
>>> count_less_than([[5,8,2,4],[-1,7,20],[3],[5,15,2,12,19]],10)
9
>>> count_less_than([[5,8,2,4],[-1,0,20],[3],[5,15,2,12,19]],10)
-1
>>> count_less_than([[100,-10,-3],[15,2,3,4,5,6],[200,-12,-15]],30)
10
>>> count_less_than([[100,-10,-3],[15,2,3,4,5,6],[200,-12,-15]],-13)
1
>>> count_less_than([[100,-10,-3],[15,2,3,4,5,6],[200,-12,-15]],-1000)
0
```

6.4 Part #4: Class Absences

Write a function called `count_absences` that takes a 2D list representing a seating chart (as shown in class) and a 2D list representing the actual attendance (i.e. who is sitting where on a given day). Your function should return the number of students who are not in their proper seat. An empty seat will always be denoted by an empty string.

You may assume that:

- Each row of the seating chart has the same length.
- Each row of the attendance has the same length as each row in the seating chart.
- The seating chart has the same number of rows that the attendance does.

Hint #1: We know that the program has to look at *every* element of the 2D list at some point, so we can infer that nested loops are helpful. Start by using nested loops to print the contents of the seating chart.

Hint #2: When testing this function with 2D lists, you may want to use short (i.e. one letter) strings instead of real names and small 2D lists instead of large ones. This will save you time.

Hint #3: I talk briefly about this specific problem in the 07/16 lecture. (As stated in the lecture capture outline, the timestamp is 1:24:40.) Hopefully what I say there might help.

Below are some examples of how your program should behave.

```
>>> seating_chart = [["Carly","Sam","Freddy","Drake","Josh"],
             ["Homer","Bart","Marge","Lisa","Maggie"],
             ["Robin","Starfire","Raven","Cyborg","Beast Boy"]]
>>> attendance = [["Carly","","Freddy","","Josh"],
          ["Homer","","Marge","Lisa","Maggie"],
          ["Starfire","Robin","Raven","Beast Boy","Cyborg"]]
>>> count_absences(seating_chart,attendance)
7
>>> attendance2 = [["Sam","Carly","Freddy","","Josh"],
           ["Homer","","Marge","Lisa",""],
           ["Batman","Luna","Nevermore","Cyclops","Beast Man"]]
>>> count_absences(seating_chart,attendance2)
10
```

6.5 Part #5: Find in Other

This note isn't specifically relevant to this problem, but if you haven't already, make sure that you read the note above about which methods you are allowed to use.

Write a function called `find_in_other` that takes as argument two lists of integers. The function should return a list of lists. Each integer in the first argument should correspond to a list in the return value, and this corresponding list should contain all indices (in increasing order) at which that integer occurs in the second list. For example, `find_in_other ([18,-3,2],[10,-5,2,4,-3,12,-3,6])` should return `[[],[4,6],[2]]` because 18 appears at no index in the second list, -3 appears at indices 4 and 6 in the second list, and 2 appears at index 2 in the third list. As another example, `find_in_other ([8,20],[4,8,12,16,20,16,12,8,4])` should return `[[1,7],[4]]` because 8 appears at indices 1 and 7 in the second list and 20 appears at index 4 in the second list.

6.6 Part #6: Count k Even Divisors

Implement a function called `count_has_k_even_divisors` that takes two integers as arguments (we'll call them n and k). This function should return the number of integers between 1 and n (inclusive) that have *exactly* k positive integers that evenly divide them. Recall that an integer a evenly divides an integer b if $b \div a$ has no remainder.

Below are examples of how your function should behave. `count_has_k_even_divisors(5,2)` returns 3 because the integers 2, 3, and 5 all have $k = 2$ integers in the range from 1 to $n = 5$ that evenly divide them. For example, 2 is evenly divided by 1 and 2, and 5 is evenly divided by 1 and 5. Similarly, `foo2(12,3)` returns 2 because the integers 4 and 9 have $k = 3$ integers in the range from 1 to $n = 12$ that evenly divide them. For example, 9 is evenly divided by 1, 3, and 9.

```
>>> count_has_k_even_divisors(5,2)
3
>>> count_has_k_even_divisors(12,3)
2
```

(This note is only relevant to those who are trying to *avoid* using nested loops for this problem.) You are not allowed to hardcode the number of integers that evenly divide each integer. For example, 1 is evenly divided by 1 integer (1), 2 is evenly divided by 2 integers (1 and 2), 3 is evenly divded by 2 integers (1 and 3), and 4 is evenly divided by 3 integers (1, 2, and 4). Consequently, I don't want to see some list of hardcoded values in your code such as `[1, 2, 2, 3, ...]`. Many of the autograder test cases will involve particularly high values of n.

7 Autograder Details

All of the details about the autograder that were given in the directions for the previous programming assignments are still relevant here, so I do not repeat them here.

7.1 Test Cases' Inputs

See the document `hw5_autograder_inputs.pdf` that I will upload to Canvas (and send an announcement about) once the autograder is ready. The reason I have placed these inputs into a separate document is that this separate document was created with Markdown rather than with LaTeX, so this should avoid the issues with copy/pasting strings and other sources of potentially distorted characters (e.g. quotation marks) from the PDF.

UCDAVIS
COMPUTER SCIENCE