Hints and Guidance: Python Review

CSCI 112, Lab 1

1. Before you begin coding, think about whether a specialized class will help you.

2. For example, the first two problems both deal with matrices represented as lists of lists. Odds are a `Matrix` class might come in handy.

3. Even such simple things as creating new matrices and printing them in a human-readable way will speed things up as you go along. For example, here is a program I wrote using my `Matrix` class and its output:

```
    print(rangeMatrix(4))
    print(rangeMatrix(5))
```

```
|   0   1   2   3|
|   4   5   6   7|
|   8   9  10  11|
|  12  13  14  15|

|   0   1   2   3   4|
|   5   6   7   8   9|
|  10  11  12  13  14|
|  15  16  17  18  19|
|  20  21  22  23  24|
```

4. It only takes a few minutes to write nice creation and `__str__` routines, and it will save you a **lot** of time down the road as you build and test your solutions!

5. Speaking of testing your solutions, why not write the unit tests **first**? There are examples of solutions in the problem statements. Code them up as unit tests before you start to write your solution, and all you have to do is hit run on the unit test file to see if you've got it right yet.

```
def test_rotate_transform(self):
    self.assertEqual(
        rotate_transform(
            [
                [2, 4],
                [0, 0]
                ], 1),
            [
                [0, 2],
                [0, 4]
                ])
```

6. To make sure you don't have any syntax errors in your unit test files, write dummy versions of your procedures that just return `None`, for example. When you run the tests, they will all fail, but you will have found any syntax mistakes in your unit test files, at least.

7. Again, it only takes a few minutes to set all this up, and it will save you a **lot** of time when you need to repeatedly test your code during development, and also, if you decide to improve your code by making it more modular, *etc.*, you can instantly check if you broke something. Unit tests are your friends!

8. Now that you've got your infrastructure ready (classes and unit tests), it's time to start thinking about actually solving the problem. As you proceed, you may add new pieces to your classes. Add unit tests for each new thing you add.

9. As you consider the problem, think about some fundamental questions:

   - What data structures might be useful? lists and dictionaries are the big two.
   - Should I use recursion or loops?
   - Should I use pure functions, which do not modify their inputs, or destructive functions, such as `x.append(y)` for lists or `d[x] = y` for dictionaries?

10. For example, in problem 1, the `rotate_transform` problem, it's pretty clear we're going to use lists. Would dictionaries be helpful?

    - A dictionary can make some things easier. With a list of lists, for example,

      ```
      ls[i][j] = x
      ```

      is problematic, since we have to make sure the indices are in bounds in the structure. However, with a dictionary

      ```
      d[i,j] = x
      ```

      always works.
    - Don't forget that it is sometimes easier to convert one data structure to another, do your work, and then convert it back when you're done.
    - The conversion from lists to dictionaries and back can be a method of your `Matrix` class.
    - Also remember that a dictionary is a persistent object. If a dictionary is a global variable, and you have various functions putting things into it, they *all* stay there, in one place, no matter how many different functions access it.
    - On the other hand, if you've got a `Matrix` class, you can replace

      ```
      ls[i][j] = x
      ```

      with

      ```
      m.assign(i, j, x)
      ```

      The object method of the `Matrix` class will take care of indices that are out of bounds, and while you're solving the problem you never have to worry about that again.

11. Still considering problem 1, let's think about both **recursion** and **iteration**.

12. Considering **iteration** first, iteration seems relevant, since we can loop over `i` and `j` and think about where they go in the solution.

    - From staring at the input and output, at right, it's clear that we have to make a mapping that, for the first row, starts out like this:

$$
\begin{array}{rcl}
(0,0) & \Rightarrow & (0,3) \\
(0,1) & \Rightarrow & (1,3) \\
(0,2) & \Rightarrow & (2,3) \\
(0,3) & \Rightarrow & (3,3)
\end{array}
$$

```
iteration
|   0   1   2   3|
|   4   5   6   7|
|   8   9  10  11|
|  12  13  14  15|

|  12   8   4   0|
|  13   9   5   1|
|  14  10   6   2|
|  15  11   7   3|
```

    - Can you express that in general terms?
    - Is there a mathematical formula relating initial and final positions?
    - Can you make it work for all rows and columns?

13. **Recursion**, on the other hand, involves breaking the original problem into smaller problems of the same kind.

    - For instance, we can cut the first column off, and get the two pieces at right.
    - Using recursion on the smaller piece results in the matrix at the bottom right.
    - Is there a way to put the sliced-off first column onto the rotated smaller matrix to solve the whole problem?
    - You also have to figure out where the recursion ends, the **base case** of the recursion, and what the answer is in that situation

```
recursion
|   0|
|   4|
|   8|
|  12|

|   1   2   3|
|   5   6   7|
|   9  10  11|
|  13  14  15|
```

```
recursion on smaller matrix
|  13   9   5   1|
|  14  10   6   2|
|  15  11   7   3|
```

    - You could alternatively think about slicing off the first **row** instead of the first column. Would all the pieces be easier, or harder that way?