Homework 6

Intro to Programming (Term III/2017–18)

built on 2018/06/16 at 16:21:26 **due:** *thu, jun 28 @ 11:59pm*

This assignment will give you some practice writing larger programs, focusing somewhat on tuples and recursion. You will write some code and hand it in electronically. As with previous assignments:

- For every function you write that returns a value, write at least 5 test cases using `assert`. Put all these tests (asserts) in a separate file called `unittests.py`, which you will also hand in. Use the **import** command to access code you write in a different file
- Before you hand in your solutions, delete or comment out all extraneous **print** statements. As a rule of thumb, if the question doesn't ask you to print it, you shouldn't print it.
- All your `asserts` must be kept in a function. When you're done testing, don't call this function.
- For functions that read input from a file, don't write assert's for that function. Instead, write assert's for their helper functions.

This assignment will be graded out of 85 points although you may earn up 98 points. If you score over 85, that's extra credit.

Overview:

Problem	File Name
1.	`passwd.py`
2.	`findme.py`
3.	`eto.py`

Problem	File Name
4.	`diet.py`
5.	`t2048.py`
***Test Cases**	`unittests.py`

Collaboration

We interpret collaboration very liberally. You may work with other students. However, each student ***must*** write up and hand in his or her assignment separately. Let us repeat: You need to write your own code. You must not look at or copy someone else's code. You need to write up answers to written problems individually. The fact that you can recreate the solution from memory will be taken as proof that you actually understood it, and you may actually be interviewed about your answers.

Be sure to indicate who you have worked with (refer to the hand-in instructions).

Logistics

We're using a script to grade your submission before any human being looks at it. Sadly, the script is not as forgiving as we are. *So, make sure you follow the instructions strictly.* It's a bad omen when the course staff has to manually recover your file because the script doesn't like it. Hence:

- Save your work in a file as described in the task description. This will be different for each task. **Do not save your file(s) with names other than specified.**
- You'll zip these files into a single file called `a6.zip` and you will upload this one zip file to Canvas before the due date.
- Before handing anything in, you should thoroughly test everything you write.
- At the beginning of each of your solution files, write down the number of hours (roughly) you spent on that particular task, and the names of the people you collaborated with as comments. As an example, each of your files should look like this:

```
# Assignment XX, Task YY
# Name: Eye Loveprograming
# Collaborators: John Nonexistent
# Time Spent: 4:00 hrs

... your real program continues here ...
```

- The course staff is here to help. We'll steer you toward solutions. Catch us in real-life or online on Canvas discussion.

Task 1: Strong Password (12 points)

For this task, save your work in `passwd.py`

Countless websites invented (often bogus) rules to promote strong password usage. A typical website enforces rules such as these:

1. At least 1 lower-case letter (a-z);
2. At least 1 numerical digit (0-9);
3. At least 1 upper-case letter (A-Z);
4. At least 1 special character from `$#@`
5. Minimum length of password: 6
6. Maximum length of password: 12
7. No character or digit can appear twice consecutively.

You will write a function `passwordOK(p_str)` that takes a password string and returns a Boolean indicating whether this password conforms to all the rules above.

Example:

- `passwordOK('ABd1234@1')` should return `True`.
- `passwordOK('f#9')` should return `False` (missing an upper-case letter among other things).
- `passwordOK('Abbc1$f')` should return `False` (b appears twice consecutively).

Task 2: Find Me, Recursively (12 points)

For this task, save your work in `findme.py`

Implement a *recursive* function `findMe(elt, lst)` that takes a value `elt` and a list `lst`, and returns the lowest index where `elt` appears in that list. It will return `None` if `elt` doesn't appear in the the list.

Your function `findMe(elt, lst)` must be recursive. That is, it calls itself on a smaller list and relies on the result of this self-calling to formulate the answer to the original request using a recursive strategy. Moreover, you are not allowed to use loops (for or while). Neither could you use the built-in `find` method. Pretty much, you can only use return, the if statement, and normal arithmetic, Boolean, list indices and slicing.

Examples:

- `findMe(7, [1,2,9,5,3])` should return `None`.
- `findMe(9, [1,2,9,5,3,9])` should return `2`.
- `findMe("nag", [1,2,9,5,"nag",9])` should return `4`.

WARNING: No points for non-recursive solutions!

Task 3: Even Then Odd (12 points)

For this task, save your work in `eto.py`

You are to implement a *recursive* function `eto(lst)` that takes a list of integers `lst` and returns a list containing the very same elements except arranged so that all the even numbers appear before all the odd numbers. Among the odd numbers, they can be arranged in any order. Among the even numbers, they can be arranged in any order. All your function needs to make sure is for all even numbers to come before all odd numbers in the list you return.

For example:

- A valid answer for `eto([3,1,2])` is `[2,1,3]`.
- A valid answer for `eto([4,2,8])` is `[8,4,2]`.
- A valid answer for `eto([8,-2,3,-3,-1,5,8,-1,5])` is `[8,-2,8,5,-1,5,-1,-3,3]`.

Restrictions: This will be the only function you'll write. Your function cannot call any other functions other than itself and basic list manipulation functions.

Task 4: Diet (12 points)

For this task, save your work in `diet.py`

Meow[1] eats a lot, and she loves to know in gory detail what she eats in every meal. As her assistant, you are going to implement a function `mealCal(meal, recipes, db)` that operates as follows:

- The parameter `meal` is a list of strings, listing the dishes she is having. There may be redundant items: if Meow likes it enough, she may consume multiple servings of the same dish. For example, `meal = ["T-Bone", "T-Bone", "Green Salad1"]`.
- The parameter `recipes` is a list of strings, representing a "book" of recipes. For example[2],

  ```
  recipes = ["Pork Stew:Cabbage*5,Carrot*1,Fatty Pork*10",
             "Green Salad1:Cabbage*10,Carrot*2,Pineapple*5",
             "T-Bone:Carrot*2,Steak Meat*1"]
  ```

  Each item is a string indicating the name of the dish, followed by a colon, then a comma separated list of ingredient names together with their quantities. In the example presented, the item `"T-Bone:Carrot*2,Steak Meat*1"` indicates that the dish "T-Bone" uses 2 units of Carrot and 1 unit of Steak Meat.
- The parameter `db` is a list of strings, representing a database of how much carbohydrate, protein, and fat each ingredient contains. Each item is listed as the ingredient's name, followed by a colon, and then *three* comma-separated numbers (int or float) denoting, respectively, the amounts of carb, protein, and fat in grams. Here is an example:

  ```
  db = ["Cabbage:4,2,0", "Carrot:9,1,5", "Fatty Pork:431,1,5",
        "Pineapple:7,1,0", "Steak Meat:5,20,10", "Rabbit Meat:7,2,20"].
  ```

  As a specific example, the entry for Cabbage indicates that 1 unit of "Cabbage" has 4g of carbohydrate, 3g of protein, and 0g of fat.
- The function then returns the amount of calories resulted from eating this meal. Remember that
  - Each gram of carbohydrate yields 4 calories.
  - Each gram of protein yields 4 calories.
  - Each gram of fat yields 9 calories.

  To give a complete example, plugging in the sample combination just described into `mealCal`, we have that `mealCal(meal, recipes, db)` computes the following:
  - First, we derive the amount of calories Meow gets from a T-Bone dish. This dish has 2 carrots and 1 unit of steak meat, so that's $(9\cdot4+1\cdot4+5\cdot9)\times2 = 85\times2 = 170$ calories from carrots and $5\cdot4+20\cdot4+10\cdot9 = 190$ calories from the meat. Hence, this dish has $170+190 = 360$ calories.
  - Then, we derive the amount of calories Meow gets from her other T-Bone dish. The amount is the same: 360 calories.
  - Then, we derive the amount of calories Meow gets from her salad dish: $(4\cdot4+2\cdot4)\times10+(9\cdot4+1\cdot4+5\cdot9)\times2+(7\cdot4+1\cdot4+0\cdot9)\times5 = 240+170+160 = 570$.

  Hence, `mealCal`, in this case, will return 1290.0.

Remarks:

- Your answers may be floating-point numbers; we'll accept any answers within 10^{-5} of our model answers.

[1] a fictitious character from last year
[2] The recipes and nutritional 'facts' are totally made up.

- The input ensures: (1) every dish in Meow's meal exists in the recipe; and (2) every ingredient used exists in the db.

Task 5: The 2048 Game (50 points)

For this task, save your work in `t2048.py`

You probably have heard (and played) the famous 2048 puzzle game. It is a game created by Gabriele Cirullia, a then 19-year-old developer, over the course of one weekend and has become a global hit.

The Original Game: Originally, 2048 is played on a 4-by-4 grid, with numbered tiles that slide up, down, left, or right using the arrow keys. Every turn, a new tile with value either 2 or 4 will appear at a random empty spot.

When the game player chooses the direction of movement for that turn using an arrow key, tiles slide as far as possible in the chosen direction until they are stopped by either another tile or the edge of the grid. If two tiles of the same number collide while moving, they will merge into a tile with the total value of the two tiles that collided. The resulting tile, however, cannot merge with another tile again in the same move. The objective is to form a 2048 tile.

The best way to learn about this game is to play it: `http://gabrielecirulli.github.io/2048/`

Our Game: Your mission for this task is to create a variant of the 2048 game that is fully working. We have written for you code that implements a (very) primitive user interface, and you are responsible for the game logic, as well as a number of other convenient functions. Once you complete the implementation, you will get a working game.

Our version of the game will be played on an m-by-n grid. This grid is a rectangle of an arbitrary size. Like in the original game, there are *four* movements possible: up, down, left, right. Initially, there is one tile with value 2 or 4 appearing at a randomly chosen spot. Then, in every turn until the game is over, the following events will take place:

- The player chooses a direction of movement.
- The game slides the tiles as far as possible in the direction of movement, combining tiles as appropriate (see the discussion that follows for details).
- If the move is not void (see below), the game creates a new tile with value either 2 or 4, appearing at a random empty spot.

To explain what happens when the player slides the grid, we describe the effects in *three* steps (for pedagogical reasons, your code must implement these steps):

STEP 1: *Slide in the Direction of Movement.* You push the tiles as far as possible in the direction of movement until they cannot be moved in that direction any further. Similarly, you can imagine gravity pulling in the direction of movement, so all the blocks follow the gravitational force before stopping at the grid's edge. As an example, we show on the right a grid and what happens after STEP 1 when the direction of movement is *right*.

EXAMPLE. Suppose the direction of movement is going *right*. STEP 1 transforms the grid on the left below to the one on the right.

2		4		4	8
	2		2		8
2	2	2		2	4
4	2		2	2	8
16					

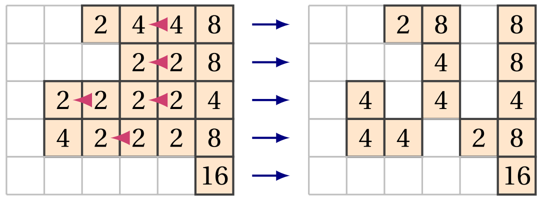
→

		2	4	4	8
			2	2	8
	2	2	2	2	4
	4	2	2	2	8
					16

STEP 2: *Compact Similar Blocks.* For each row or column in the direction of movement (DOM), you will start at one end of the grid and walk in the DOM *through* the grid to the other end. Along this walk, upon encountering two adjacent tiles of the same value, the latter is merged *into* the former, their point values combined. This means there will no longer be a tile at the latter location. Then, the

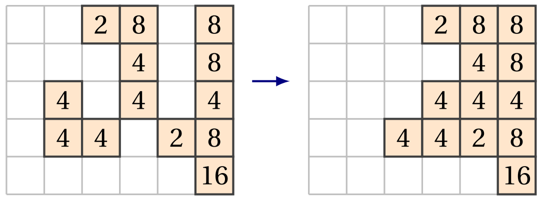
walk continues. Keep in mind that more than one merges can take place per walk.

EXAMPLE. Below, the left grid was processed by STEP 1; the right grid shows the effects of STEP 2. Notice how the tiles are combined using this process, paying attention especially to the rows that more than one merges take place.



STEP 3: *Slide Once More.* You may have created gaps in STEP 2. Identical to STEP 1, Step 3 pushes the tiles as far as possible in the direction of movement until they cannot be moved in that direction any further.

EXAMPLE. Applying STEP 3 to the result of STEP 2 yields the grid on the right below.



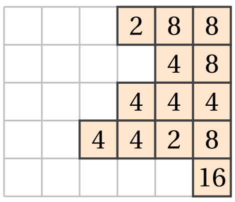
Special Circumstances: It is possible that a move will not cause any change to the grid. We call this type of move a *void move*. Your program will handle treat this case specially.

The game is considered *over* when the current grid is full and (ii) all directions of movement result in void move.

How to Get Started? We are providing a starter package containing a primitive graphical user-interface (GUI) and function stubs for you in a separate file. You will download the starter package from the course website.

All functions that you have to implement for this problem are in `t2048.py`. This file is in turn used by the GUI. When we grade your submission, we'll both play the game using the GUI and use a script to interact with your functions in this file directly.

Board Representation: The board (aka. grid) in this game is represented as a list of lists (2d list). Each tile on the board is represented as string, where `' '` (a single space) is an empty tile; otherwise, it is a string that is the value of the tile (e.g., `'128'`). As an example, the figure below show a 2-d list encoding and its corresponding board:



```
boardA =
  [[' ', ' ', ' ', '2', '8', '8'],
   [' ', ' ', ' ', ' ', '4', '8'],
   [' ', ' ', ' ', '4', '4', '4'],
   [' ', ' ', '4', '4', '2', '8'],
   [' ', ' ', ' ', ' ', ' ', '16']]
```

What're You Implementing? There are a number of functions you are going to implement (see also the stubs in `t2048.py`). The first family of functions is concerned with movements: `doKeyUp(board)`, `doKeyDown(board)`, `doKeyLeft(board)`, `doKeyRight(board)`, corresponding to the directions of movement up, down, left, and right, respectively. For each of these functions:

- *Input:* a board/grid whose dimension is as specified by the input.

- *Output:* returns a tuple `(changed, new_board)`, where `changed` is a Boolean indicating whether or not the board has changed; and `new_board` is the board that results after that particular movement is made.

You are also to implement a few functions that help operate the game:

- `emptyPos(board)` takes as input a board (represented as described above) and returns a list of `(row, col)` tuples of empty spots on this board. For example, for the board `[[' ', '2'],['4', '']]`, the function will return `[(0,0), (1,1)]`.
- `hist(board)` takes as input a board (represented like before) and returns a dictionary that maps every tile value that appears in the board currently to the number of times it occurs on the board. For example, for the board, `[[' ', '2', '2'],['4', '2', '']]`, it will return `{2: 3, 4: 1}`. Notice that the tile values in the output dictionary are integers (not strings).
- `isGameOver(board)` takes as input a board (represented like before) and returns the game on that given board is over (see the definition above). That is, it checks whether the given board has any possible move remaining.

Grading: We allocate 50 points for this problem. For each function we ask you to implement, we'll test it thoroughly using a script and grade it manually for style and clarity. You should provide unittests for all of the functions.