Assignment #2 – Sliding Puzzle

420-SF2-RE: Data Structures and Object Oriented Programming
March 3, 2025

Late submissions are not accepted!

In this assignment, you will write a program for the sliding puzzle game on a `n x n` tile board for `n > 2`. Given `n x n` tile board, where one of the tiles is empty and all other tiles are each integers from the set `{1, 2, …, n*n - 1}`, the objective is to obtain an increasing sorted order of the tiles from left to right and top to bottom by continuously sliding the tiles. Tiles can only slide to an adjacent empty cell. Possible valid moves are: left, right, up, down.

Constraints:

- For this assignment, you may only use what has been covered in class up to and including 2-dimensional lists. Use of anything else will result in severe deduction of marks.
- Design and efficiency of code will be taken into account for this assignment. For example, make sure you name your variables properly and in snake case; any repetitive code should be extracted into a helper function; etc.
- Make sure your function names and arguments passed to them are exactly as stated in this assignment so that the automated test cases will not fail because of naming functions incorrectly.

Part 1: due at the end of class on March 3

- **60% of Assignment#2 grade**
- **On Git in your Assignment#2 folder you are to add, commit, push your .py file called sliding_puzzle_part1.py**
- **On Moodle under Assignment#1 Part-1 Submission you are to upload your .py file called sliding_puzzle_part1.py**
- **Make sure you add your name and student ID at the top of the file as comments.**
- **Failure to follow submission instructions properly will result in a grade of 0 for Part-1 of Assignment#2.**

- It is recommended that you write the functions below in the order given.
- You may define helper functions if you deem it necessary
- Some of the code is already provided for you. Do not touch that code. However, you must use it in your program.

Functions to Implement for Part 1 (due end of class):

Note that as you progress writing your functions, you must use the previously defined functions by you or what was provided.

For any of the functions below (or for any chunk of code in your program) that does not compile will receive an automatic grade of 0. If a portion of your code is not working comment it out in your .py file before submission.

1. **(time estimate: 5 minutes)** Write a function called `tileLabels` that takes the side dimension `n` of the tile board and returns a list of strings numbered `1` to $n^2 - 1$ (in that order) and in addition the string `'  '` (2 spaces) to designate a blank tile.

   For example:
   for `n = 3`, your function should return the list `['1', '2', '3', '4', '5', '6', '7', '8', '  ']`

2. **(time estimate: 15 minutes)** Write a function called `getNewPuzzle` that takes an argument `n` (as the dimension of the tile board) and returns `n x n` puzzle ready to be played (i.e. a list of sublists, where the list contains `n` sublists and each sublist contains `n` labels). Note that you must use the `tileLabels` function and all labels should be shuffled in random order before creating the new puzzle ready to be played. If the tile numbers are single digit numbers, then there is a space after them. If the tile numbers are 2 digit numbers, then there is no space after them.

   For example:

   for `n = 3`, here is an example of what your function may return after tile labels are shuffled:

   `[ ['3 ', '  ', '8 '], ['1 ', '5 ', '2 '], ['6 ', '4 ', '7 '] ]`

   And the board visually will look as follows:

3		8
1	5	2
6	4	7

3. **(time estimate: 10 minutes)** Write a function called `findEmptyTile` that takes the board of tiles (i.e. the list of sublists) and returns as a 2-tuple the `(row_number, column_number)` of the empty tile on the board.

   For example:

   For the following board below, which is represented by the following list:

   `[ [‘3 ’, ‘  ’, ‘8 ’], [‘1 ’, ‘5 ’, ‘2 ’], [‘6 ’, ‘4 ’, ‘7 ’] ]`

   your function returns `(0, 1)` as the location of the empty tile.

3		8
1	5	2
6	4	7

4. **(time estimate: 20 minutes)** Write a function `nextMove` that takes the board of tiles as an argument and based on the user's input determines and returns the next move to be played on the board.

   The 4 keys that will be used for moving tiles as the game is being played are **`W`**, **`A`**, **`S`**, **`D`** letters on the keyboard.

   - If the user enters **`W`**, then a tile is moved up into the empty tile space
   - If the user enters **`A`**, then a tile is moved to the left into the empty tile space
   - If the user enters **`D`**, then a tile is moved to the right into the empty tile space
   - If the user enters **`S`**, then a tile is moved down into the empty tile space

This function first displays to the user the **possible valid moves** for the current board. **Hence, your function should determine all valid moves for the given board.** Then it prompts the user to enter one of the available **valid moves** for the current board. See the example below.

For example:
For the board below, the prompt that will be shown to the user is given below the board (**make sure your formatting matches this**). Note that, the key **A** is missing and the user only sees **( )**, since the empty tile is on the edge of the board and there is no tile that can be moved to the right into the empty cell.

3	7	5
1	6	
4	8	2

```
                          (W)
Enter WASD (or QUIT): ( ) (S) (D)
> {waiting for user's input here}
```

- If the user enters `'quit'` (either lowercase or uppercase), the program should completely terminate. Note that you may use `sys.exit()` here if you wish.
- If the user enters an invalid move, prompt the user again with the same valid moves as before and allow the user to enter their move. This should be done continuously until the user enters one of the available valid moves on the board.
- If the user enters a valid move, your function should return the move made by the user as a string.

Note that you must use previously defined functions.

Part 2: on March 7 by 23:59

- **40% of Assignment#2 grade**
- **On Git to your sliding_puzzle_part1.py file commit and push your changes of part2**
- **On Moodle under Assignment#2 Part-2 Submission upload your final .py file called sliding_puzzle_part2 (note that only Part-2 will be graded here). You must also upload a .txt file with a link to your final git submission.**
- **Make sure you add your name and student ID at the top of the .py file as comments.**
- **Failure to follow submission instructions properly will result in a grade of 0 for Part-2 of Assignment#2.**

- It is recommended that you write the functions below in the order given.
- You may define helper functions if you deem it necessary
- Some of the code is already provided for you. Do not touch that code. However, you must use it in your program.

Functions to Implement for Part 2:

For any of the functions below or the Main Program that does not compile will receive an automatic grade of 0. If a portion of your code is not working comment it out in your .py file before submission.

5. Write a function `makeMove` that takes two arguments: the board of tiles and the next move made by the user. This function assumes that the next move is already valid. This function updates the board with the next move and returns nothing.

   For example:

   initial board, next move is **`W`** ➔ updated board after the move **`W`** is made

3	7	5
1	6	
4	8	2

3	7	5
1	6	2
4	8	

6. In your **main program** you are to do the following:
   a. Print short welcome message to the user with instructions to the game.
   b. Prompt the user to enter the value of `n` (i.e. the dimension of the board).
   c. Generate a new puzzle board, display it to the user and allow the user to play using the functions you defined both in Parts 1 and 2.
      - Note that each time a user makes a move and the board is updated, the updated board must be displayed back to the user.
      - If the player wins the game print out a congratulatory message. For 9 tile board and one empty tile, you should allow the user to make at most 31 moves. For 15 tile board and one empty tile, you should allow the user to make at most 80 moves. In such cases your program should print to the user "`Best of luck next time!`".
      - It is recommended that you check your program first with 9 tile board since 9 square puzzle is easier to solve than the 15 square puzzle.