National University of Singapore
School of Computing
CS1010S: Programming Methodology
Semester II, 2016/2017

Mission 10.1 - Side Quest
2048 Game

Release date: 15 October 2017
Due: 24 October 2017, 23:59

Required Files

- sidequest10.1-template.py
- puzzle.py

Background

You fidget as Grandmaster Ben continues his oration entitled "The Power of Two":

"It might not seem like much when twos combine, but as their powers combine, their value grows exponentially—you get an even higher power of two each time. Each of us has our own strengths, but when just two of us merge our abilities or, so to speak, combine our powers, our value likewise increases exponentially..."

His metaphor evokes an uncanny feeling in you as you recall your experience at the portal. All these coincidences are starting to seem a little puzzling.

Information

In this sidequest, you will implement a variation of the game *2048* by Gabriele Cirulli. This game is avaliable at `http://nussoc.github.io/2048/` and was originally hosted at `http://gabrielecirulli.github.io/2048/`.

The focus of this sidequest is on structuring data. You will be working with more complex data structures built using the 'glue' of tuples and lists which you have learnt and writing abstraction layers for them.

The template file `sidequest10.1-template.py` has been provided for you. In addition, another file `puzzle.py` has also been provided. It contains code that will render the graphical interface for your game. After you have written your solutions in the template file, copy only your functions onto coursemology. Do not `import puzzle` on coursemology.

This mission consists of **five** tasks, one of them optional.

Friendly Reminders

- If it is not obvious what you are doing with your code, add comments!
- Create your own test cases and try to test as many scenarios that your code will encounter as possible.
- Do not break abstraction barriers.

2048 Gameplay

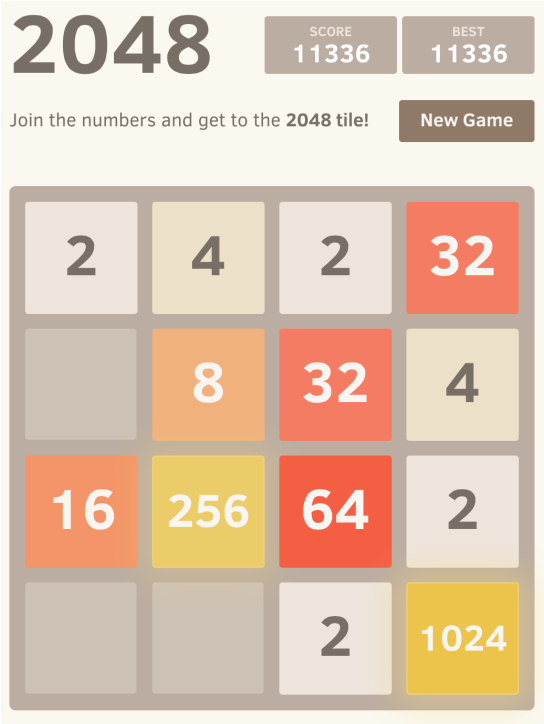

Figure 1: Screenshot of *2048* by Gabriele Cirulli.

The gameboard consists of a 4-by-4 grid where each cell is either empty or contains a tile that is labelled with a number. During gameplay, the player presses one of the four direction keys and all the tiles slide in that direction. When two tiles of the same value collide, they merge to become a tile whose value is their sum. Players start with two randomly placed tiles with the value 2 and a new tile of value 2 or 4 is added each time the player successfully makes a move. The player wins when the '2048' tile is obtained and loses if he cannot make any more moves.[1]

Mission Brief

We start by tackling the heart of the problem—the game matrix. In the first half the mission, you will be writing functions to handle the matrix for the start, middle and end parts of the gameplay. These will cumulate into a text-based version of the game.

[1] Description adapted from Wikipedia.

In the second half, you will be adding graphics to your game. A graphical framework, which has been written to work with a specific set of functions (which form an abstraction layer), is been provided for this purpose. You will be writing an abstract data type that has these specified functions to represent the game state. Besides the game matrix, the game state also helps us to keep track of auxiliary information like the score and past moves.

Recording the past moves allows us to implement an 'undo' function for the game. You final task is to make this improvement to the original game. How cool is that!

Working with Matrices

At the core of the game is the 4×4 game matrix. In general, 2-dimensional matrices may be represented as a list of nested lists. For example, a 4×4 matrix,

$$\begin{vmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{vmatrix}$$

is represented below with in nested lists format as:

```
[[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12], [13, 14, 15, 16]]
```

We will be using such a data structure to represent our game matrix. Zero will be used to represent an empty cell, while non-zero values represent tiles with that value.

Since we are working with sequences, our sequence processing skills will come in handy! The functions `accumulate` and `flatten` that you have encountered in the previous missions have been provided for you. You may include and use other list processing functions we have covered. The next section introduces Python's built-in functions `map` and `filter`.

Python's Built-in `map` and `filter`

Up till now, you have been using `map` and `filter` functions provided in the template files. As you have seen, they can be unreliable when you have to work with large datasets. We circumvent these issues by using Python's built-in mapping and filtering functions.

These functions are used in the same way as our regular `map` and `filter` except that we have to do the additional step of converting the result to a list or tuple:

```
>>> y = filter(lambda x: x%2, [1,2,3,4,5,6,7,8,9])
>>> y
<filter object at 0x1042b0f90>
>>> tuple(y)
(1, 3, 5, 7, 9)
```

Task 1: Starting a New Game (4 marks)

Using the representation we have just described, we now write functions to initialize the game matrix and add the starting tiles.

Task 1a: Empty Grid (1 mark)

Write a function `new_game_matrix(n)` that returns an empty square matrix of size $n \times n$. In this case, an empty matrix is defined as a matrix with all its values as zeroes.

Task 1b: Empty Cells (1 mark)

Write a function `has_zero(mat)` that returns `True` if the matrix `mat` has an entry with value zero and `False` otherwise.

Hint: Use `flatten`.

Task 1c: Random Number Generator (2 marks)

In this game, 2s are created at random empty locations on the matrix. For simplicity, we will not be creating 4s.

Write a function `add_two(mat)` that accepts an arbitrarily-sized matrix and creates the value 2 at a random zero location on the matrix. Return the original matrix if no zero locations are found.

Note: The `randint(a,b)` function from the `random` package will generate a random integer from a to b inclusive.

Task 2: Game Over (4 marks)

We need to be able to tell if the game has ended. After each move, the game can hold one of the following statuses:

- Win - The player has created a tile with the value 2048.
- Lose - There are no possible moves and no empty tiles on the board.
- Not over - The game is in neither of the above states.

A player has moves if either of these conditions are fulfilled:

- There are empty tiles on the board
- Any two adjacent tiles have the same value

Write a function `game_status(mat)` that takes a game matrix and returns `'win'` if the game is won, `'lose'` if the game is lost, and `'not over'` if the game is not over. Your function should be able to handle matrices of any size.

Task 3: Gameplay (11 marks)

We have left to write the game logic to make the game playable. We will first define a couple of helper functions, then write functions to handle the 'sliding' of the tiles and finally put them altogether into a playable game.

Task 3a: Transpose (1 mark)

A common operation applied to matrices is the transpose operation. Transposing a matrix flips a matrix along the diagonal running from the top left entry to the bottom right entry. Mathematically, this is expressed as:

```
matrix[i][j] = transposed_matrix[j][i] for all i,j
```

Note that you can also transpose matrices that are not square. A $n \times m$ matrix when transposed will return you a $m \times n$ matrix.

An example is given here:

```
>>> mat = [[1, 2, 3], [4, 5, 6]]
>>> transposed_mat = transpose(mat)
>>> transposed_mat
[[1, 4], [2, 5], [3, 6]]
```

Write a function `transpose(mat)` that takes in an arbitrary matrix and returns its transpose as a new list.

Hint: Refer to Tutorial 6.

Task 3b: Reverse (2 marks)

Reversing a matrix brings the rightmost entry to the left, and the leftmost entry to the right. For example,

```
>>> mat = [[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]]
>>> reversed_mat = reverse(mat)
>>> reversed_mat
[[4,3,2,1],[8,7,6,5],[12,11,10,9]]
```

Write a function `reverse(mat)` that takes in a matrix and returns that matrix with the order of values in each row reversed. The returned matrix should not be identical to `mat` (i.e. you should not change the input array).

Task 3c: Merging Tiles (7 marks)

Your task is to create four functions - `merge_up`, `merge_down`, `merge_left`, `merge_right` that will handle the movements in the respective direction for a given game matrix `mat`. For ease of description, we will state the rules for this movement of tiles in the left direction. We will refer the pre-movement matrix as the 'old' matrix and the post-movement one as the 'new' matrix. To obtain the new matrix, for each row of the old matrix, starting from the leftmost tile (exclude empty cells) of the row, process tiles in the following way:

- If there is no tile on its right, add a tile of the same value to leftmost available (empty) cell in the corresponding row of the new matrix.
- If the tile on its right T is of the same value as it, add a tile of their combined value to leftmost available cell in the corresponding row of the new matrix. Move on to process the tile on the right of T if there is one.
- If T has a different value than the current tile, add a tile that has the same value as the current tile to leftmost available cell in the corresponding row of the new matrix. Move on to process T.

Substitute the appropriate directions to obtain the other sets of instructions. For `merge_up` and `merge_down` you should be processing the old matrix columnwise, starting from the top and bottom respectively. You are encouraged to play the game and come up with some eqivalent formulation of the rules that might be easier to implement from.

Each of the four functions should take a game matrix `mat` as input and return a tuple in the form `(new_matrix, is_valid, score_increment)` where:

- `new_matrix` is the resulting matrix after applying the movement and merging rules above.
- `is_valid` is `False` if an invalid move is made and `True` otherwise. An invalid move is one where `mat` remains unchanged after applying the above rules, i.e. where `new_matrix` is equal to `mat`.
- `score_increment` is sum of the values of all newly formed tiles.

Sample Execution

```
>>> mat = [[2, 0], [0, 0]]
>>> new_mat = merge_left(mat)
>>> new_mat
([[2, 0], [0, 0]], False, 0)
# In this case, the matrix has not changed. Therefore, the score
# increment is 0 and the boolean is False
```

```
>>> mat2 = [[2, 0, 2, 2], [0, 0, 0, 4], [4, 0, 8, 4], [2, 0, 0, 0]]
>>> new_mat2 = merge_right(mat2)
>>> new_mat2
([[0, 0, 2, 4], [0, 0, 0, 4], [0, 4, 8, 4], [0, 0, 0, 2]], True, 4)
# In this case, only the first row has a merger where two 2s form a 4.
# Therefore the score increment is 4, and the boolean is True
# since the matrix has been changed. Note that in the first row of
# mat2, it is the rightmost 2s that are merged and not the first two
# 2s from the left.
```

Tasks

(i) Define `merge_left`. (4 marks)

Hint: Notice how each row can be processed independently? Use `map`.

(ii) Define `merge_up`, `merge_down` and `merge_right` in terms of `merge_left` with the help of `reverse` and `transpose`. You should not be rewritting the code in `merge_left`. (3 marks)

Task 3d: Play Time! (1 mark)

A simple text-based interface has been provided for you to test your functions. Uncomment `text_play()` to play the game. If you've written all your functions correctly, you should be able to play the game!

The inputs 'W', 'S', 'A', 'D' correspond the upward, downward, leftward and rightward movements respectively. Enter 'Q' to quit the game.

How would you test that the winning condition for the game works? (1 mark)

Task 4: Going Graphic (8 marks)

You have trained hard and it is time to see your efforts come together to birth the final product of this mission. Your task now is to get your game to work with a graphical interface by implementing an abstract data type (ADT) to represent your game state. The file `puzzle.py` contains code that will render a simple graphical interface for the game using your ADT. It assumes that your ADT has an abstraction layer consisting of (at least) the following functions:

- `make_new_game(n)` creates a representation of the game state for a new game. `n` is the size of the game matrix and `make_new_game` should add the two starting tiles to the game.
- `get_score(state)` return the score of the current game as an integer.
- `get_matrix(state)` returns the current game matrix. It should be a list of lists as per the first part of the mission.
- `up(state)`, `down(state)`, `left(state)` and `right(state)` each take in a game state and return a tuple containing (in order) the following:
  1. The updated game state after the respective move has been made.
  2. A boolean that is `False` if an invalid move has been made and `True` otherwise.

  Your functions `up`, `down`, `left` and `right` should add a new '2' tile to the grid if a successful move is made. Since the four functions behave in a similar manner, abstract out the common patterns between them.

  Hint: Think 'higher-order function'.

You game state should comprise of at least two items: the current game matrix and the current score. You should additionally define the following function for internal use:

- `make_state` that constucts a representation of a game state.

A sample execution run demonstrating how the ADT should work is provided for you on the last page.

Implement the Game State ADT, along with the above functions to manipulate it.

Playing the Game

Uncomment "`gamegrid = GameGrid(game_logic)`" to play the game. The direction controls are mapped to 'W', 'A', 'S', 'D' as before and the undo function has been mapped the 'Z' key on your keyboard.

Task 5: Keeping Game Records (Optional)

In this task, you will modify the game to allow users to undo their last three moves. In order to keep track of the latest three moves, we will make use of a modified stack. After each valid move, we need to remember the last game matrix and the current score increment. When executing an `undo` operation, we want to restore the game matrix we had before the current move and deduct the last increment to obtain the previous score.

(i) Implement the Game Record ADT.

You should define the following functions:

- `make_new_record(mat, increment)`
- `get_record_matrix(record)`
- `get_record_increment(record)`

(ii) Implement the Game Records Stack ADT.

You should define the following functions:

- `make_new_records()` creates a new game records stack.
- `is_empty(stack_of_records)` is either `True` or `False`.
- `push_record(new_record, stack_of_records)` returns the updated stack containing records of the latest three moves. `new_record` should be a Game Record.
- `pop_record(stack_of_records)` returns a tuple containing three items: the last game matrix, the last score increment, and the updated stack. If the stack is empty, return `None` for the matrix and increment.

You may define other appropriate functions for your ADT.

(iii) Modify the Game State ADT in the following ways:

- Instead of two items, you game state should be composed of three items: the current game matrix, the current score and a Game Records Stack. Modify the existing Game State ADT constructors to handle the additional component of the game state.
- Implement the accessor `get_records` that returns a stack.
- Modify your code such that the game's past records are updated each time a valid move is made for `up`, `down`, `left` and `right`.
- Write `undo(state)` takes in a game state and returns a tuple in the same format as `up` etc.

  A fourth or more consecutive `undo` operation is considered an invalid move and the game state should remain unchanged if an invalid move is made.

If you abstracted well and did not breech any barriers, you shouldn't have to modify too many of your functions.

Note: For this task, where you are required to 'modify' functions, you should override them by defining a function of the same name. Be sure the new function comes after the old one in your file.

Uncomment the lines at the end of the template file to play the improved game. Enjoy!

Sample Execution of Game State ADT

```
>>> game_state = make_new_game(4)
>>> get_matrix(game_state)
[[0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0], [0, 2, 2, 0]]
>>> game_state, is_valid = down(game_state)
>>> is_valid
False
>>> get_matrix(game_state)
[[0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0], [0, 2, 2, 0]]
>>> game_state, is_valid = left(game_state)
>>> get_matrix(game_state)
[[0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0], [4, 2, 0, 0]]
>>> get_score(game_state)
4
>>> game_state, is_valid = up(game_state)
>>> get_matrix(game_state)
[[4, 2, 0, 0], [0, 0, 2, 0], [0, 0, 0, 0], [0, 0, 0, 0]]
>>> game_state, is_valid = up(game_state)
>>> get_matrix(game_state)
[[4, 2, 2, 0], [0, 0, 0, 0], [0, 0, 0, 2], [0, 0, 0, 0]]
>>> game_state, is_valid = right(game_state)
>>> get_matrix(game_state)
[[0, 0, 4, 4], [0, 2, 0, 0], [0, 0, 0, 2], [0, 0, 0, 0]]
>>> get_score(game_state)
8
```

If you completed Task 5, continuing the execution, you will get:

```
>>> game_state, is_valid = undo(game_state) # first undo
>>> get_matrix(game_state)
[[4, 2, 2, 0], [0, 0, 0, 0], [0, 0, 0, 2], [0, 0, 0, 0]]
>>> game_state, is_valid = undo(game_state) # second undo
>>> get_matrix(game_state)
[[4, 2, 0, 0], [0, 0, 2, 0], [0, 0, 0, 0], [0, 0, 0, 0]]
>>> game_state, is_valid = undo(game_state) # third undo
>>> get_matrix(game_state)
[[0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0], [4, 2, 0, 0]]
>>> is_valid
True
>>> get_score(game_state)
4
>>> game_state, is_valid = undo(game_state) # fourth undo
>>> get_matrix(game_state)
[[0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0], [4, 2, 0, 0]]
>>> is_valid
False
```