Lab 6: Scoring

Our game is now essentially playable but there is one obvious missing component. In this exercise, we will add a scoring mechanism and print the score at each step when we print the grid. This is the final addition we need before we start building a `tkinter` GUI for the game.

Points are earned every time a merge happens. The number of points earned is equal to the value of the newly merged tile.

Consider the following case:

```
[[None, None, None, None],
 [None,    2, None,    2],
 [None, None, None, None],
 [None, None,    2,    2]]
```

On moving left (or right) the two pairs of `2`'s will merge, producing two `4`'s and earning **eight** points. On moving up (or down) only one pair will merge, earning **four** points.

Create the score attribute

The first thing we will do, is to create a `score` attribute on our `Game` class. We can initialise it to zero in the `__init__` method by just adding a single line like this.

```python
    def __init__(self):
        self.grid = [[None, None, None, None],
                     [None, None, None, None],
                     [None, None, None, None],
                     [None, None, None, None]]

        self.set_random_empty_tile(2)
        self.set_random_empty_tile(2)

        self.moves = {
            "W": core.move_up,
            "A": core.move_left,
            "S": core.move_down,
            "D": core.move_right
        }

        self.score = 0 # <- this is new
```

> We have added a single additional line like this
>
> ```
> self.score = 0
> ```

Showing the score attribute

Now, in order to see the score, we can update the **str** method. Modifying the last line in the method to include the score in the string representation of the game.

```
    def __str__(self):
        tiles = [[str(t or ".").center(4) for t in row] for row in self.grid]
        result = "\n".join([" ".join(row) for row in tiles])
        msg = ""
        if not self.playing:
            msg = "\nYOU QUIT THE GAME\n"
        if self.game_over:
            msg = "\nGAME OVER\n"
        return f"\nSCORE: {self.score}\n\n{result}\n{msg}" # <- this has changed
```

Calculating the score

We need to calculate the additional points earned by each move. This is a task for the core module.

Notice that the points are the same for left and right moves and for up and down moves. So we only need two function, we will implement `horizontal_points` and `vertical_points` functions in the `core` module. These functions will not be used to modify the grid, they will simply calculate the potential points.

The `horizontal_points` function will do all the work. The `vertical_points` function can be very simple because it just needs to transpose the grid and call the `horizontal_points` function like this:

```
def vertical_points(grid):
    grid = transpose(grid)
    return horizontal_points(grid)
```

To calculate the potential points from a horizontal move, we need to first stack all the tiles to the left. Then we need to look for pairs, just like we did when looking for potential merges. So our function goes something like this.

```python
def horizontal_points(grid):
    grid = [stack_left(row) for row in grid]
    points = 0
    for row in range(4):
        for col in range(3):
            if grid[row][col] and grid[row][col] == grid[row][col + 1]:
                points += grid[row][col] * 2
    return points
```

We create a new grid by stacking each row with `stack_left`. Then we initialise the points to zero and loop over each row. In each row, we check the three possible pairs. If they are not `None` and are the same, we add to the points variable. Once all the loops have ended, we return the points variable.

This looks ok, but we should test it to make sure.

Testing the score calculation

We can implement some tests using the usual pattern. Create two classes, `TestHorizontalPoints` and `TestVerticalPoints`. Add a few test functions to each class to test the function returns the correct points. Just make up some sensible example grids.

Be careful to include a zero points example and an example where some points are earned. Try to include things like pairs of `None` values and multiple possible pairs in a row.

```python
class TestHorizontalPoints(unittest.TestCase):

    def test_no_points(self):
        input  = [[None, None, None, None],
                  [   2,    4,    8,    4],
                  [   2,    4,    8,    4],
                  [   2,    4,    8,    4]]
        self.assertEqual(core.horizontal_points(input), 0)

    def test_some_points(self):
        input  = [[None, None, None, None],
                  [   2,    2,    8,    4],
                  [   2,    4,    8,    8],
                  [   2,    4,    8,    4]]
        self.assertEqual(core.horizontal_points(input), 20)

class TestVerticalPoints(unittest.TestCase):

    def test_no_points(self):
        input  = [[None, None, None, None],
                  [   2, None,    8,    4],
                  [   4,    8,    4,    2],
                  [   2,    4,    8,    4]]
        self.assertEqual(core.vertical_points(input), 0)

    def test_some_points(self):
        input  = [[None, None, None, None],
                  [   2,    2,    8,    4],
                  [   2,    4,    8,    8],
                  [   2,    4,    8,    4]]
        self.assertEqual(core.vertical_points(input), 28)
```

Try to run the tests.

Oh dear, some of the tests are failing!

Find out where the problem is and see if you can work out why the calculations are not working properly.

> Hint: There is some double counting happening here.
>
> The `grid` variable inside the `horizontal_points` function can be modified as we calculate the score.

Don't worry if you can't solve it. Move on and we will provide the solution at the end.

Updating the score

The next step is to integrate the new `core` functions into our game and update the `self.score` attribute accordingly at each move. To do this, we will need to know whether to use `horizontal_points` or `vertical_points`. We can do this with another dictionary to map the functions to the user-provided command.

Add the following into your `__init__` method:

```python
        self.point_functions = {
            "W": core.vertical_points,
            "A": core.horizontal_points,
            "S": core.vertical_points,
            "D": core.horizontal_points
        }
```

Here we are mapping the "W" and "S" commands to the vertical function and "A" and "S" to the horizontal function. So, given the user-provided command, we can access the correct function.

The final addition is to calculate the new score in the `process_command` method. We do this just before we apply the move to the grid. This allows us to use the existing grid data to calculate the score.

```python
    def process_command(self, command):
        next_grid = self.moves[command](self.grid)
        if next_grid != self.grid:
            self.score += self.point_functions[command](self.grid)
            self.grid = next_grid
            new_tile = choice([2, 2, 2, 4])
            self.set_random_empty_tile(new_tile)
            self.game_over = core.is_game_over(self.grid)
```

Try the game. You should see that the score now updates after each merge.

Solution

The tests failed above because three similar tiles in a row were being treated as two merges. The answer is to set any matching tiles to `None` once you have added their score to the result.

```python
def horizontal_points(data):
    data = [stack_left(row) for row in data]
    points = 0
    for row in range(4):
        for col in range(3):
            if data[row][col] and data[row][col] == data[row][col + 1]:
                points += data[row][col] * 2
                # These next two lines are new
                data[row][col] = None
                data[row][col + 1] = None
    return points
```

Challenges

Create a new file `gui.py`. Using the `game` module as a template, spend some time trying to build an equivalent game class using `tkinter`.

Show a score label at the top. Add a frame to hold the tiles. Make each tile a simple label.