Lab 3: Transforming the grid

Last week we completed our basic, fully tested `move_left` function for moving a single row to the left. This week we will work on expanding that core functionality.

1. We need it to work with a 4 × 4 grid.
2. We need to be able to move in all four directions.

The challenges posed at the end of the last exercise were to:

1. Upgrade the `TestMoveLeft` methods to take a 4 × 4 grid of tile data, and
2. Upgrade the `move_left` function to pass your tests.

To upgrade the tests, we need to provide input as a *list of lists*. Our most simple test case, where we move an empty grid (and nothing happens) would require data like this:

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

> The above formatting makes some code examples very long but has the advantage of showing the data in a grid.

A basic test case

Upgrading the `TestMoveLeft` test case is not so difficult. We simply expand the data from a single row to a full grid. We expect the `move_left` function to accept an entire grid as input and to return it transformed. This allows us to specify some test cases that clearly demonstrate whether the full move is working correctly.

```
class TestMoveLeft(unittest.TestCase):

    def test_empty(self):
        """An empty grid is unaffected by a merge"""
        input = [[None, None, None, None],
                 [None, None, None, None],
                 [None, None, None, None],
                 [None, None, None, None]]

        self.assertEqual(core.move_left(input), input)

    def test_value_after_pair(self):
        """Additional tiles should be stacked into the merged pair"""
        input =    [[None, 4,    4,  8],
                    [2,    None, 2,  2],
                    [None, 2,    2,  2],
                    [16,   None, 16, 4]]

        expected = [[8,  8, None, None],
                    [4,  2, None, None],
                    [4,  2, None, None],
                    [32, 4, None, None]]

        self.assertEqual(core.move_left(input), expected)

    def test_all_twos(self):
        "Two pairs are both merged and stacked"
        input =    [[2, 2, 2, 2],
                    [2, 2, 2, 2],
                    [2, 2, 2, 2],
                    [2, 2, 2, 2]]

        expected = [[4, 4, None, None],
                    [4, 4, None, None],
                    [4, 4, None, None],
                    [4, 4, None, None]]

        self.assertEqual(core.move_left(input), expected)
```

This looks a lot more complicated but if you read the code carefully, you will see that we are specifying the grid before and after the move and testing that our expectations are met.

Refactoring `core.py`

Now, to get these tests to pass we need to do some refactoring in `core.py`. First, we need to rename our `move_left` function to `row_left`. This indicates that it acts on a single row.

```
def row_left(row):
    """A full move involves stacking, merging and then stacking again"""
    stacked = stack_left(row)
    merged = merge_left(stacked)
    return stack_left(merged)
```

Now we can add a new `move_left` function which simply applied our tested logic to each row in the grid.

```
def move_left(grid):
    """moving a full grid to the left by moving each row to the left"""
    return [row_left(row) for row in grid]
```

> Remember list comprehensions? We are returning a new list composed of processed rows.

Now, we should be able to run the tests and see that they all pass.

```
$ python3 test.py
................
----------------------------------------------------------------------
Ran 16 tests in 0.001s

OK
```

Moving right

To move right, all we will do is reverse the grid, call our `move_left` method, and reverse the result. This saves us from implementing inverted versions of the stack and merge code.

The code is pretty simple. First, we can write a single test to confirm that we are correctly flipping the grid.

```
class TestReverse(unittest.TestCase):
    """reversing the grid should flip the tiles from left to right"""

    def test(self):
        input  = [[ 1,  2,  3,  4],
                  [ 5,  6,  7,  8],
                  [ 9, 10, 11, 12],
                  [13, 14, 15, 16]]

        output = [[ 4,  3,  2,  1],
                  [ 8,  7,  6,  5],
                  [12, 11, 10,  9],
                  [16, 15, 14, 13]]
        self.assertEqual(core.reverse(input), output)
```

Now a function to reverse the grid horizontally.

```
def reverse(grid):
    """flip the grid horizontally"""
    return [list(reversed(row)) for row in grid]
```

The code returns a simple list comprehension which flips each row. To flip a row, we call the built-in `reversed` function, but we also need to convert the result back into a list because the `reversed` function actually returns a special reverse iterator object. Notice that the rows are kept in the same order, so the impact is to flip the grid in one dimension only.

> The new test should now pass

Now we can create a `move_right` function which makes use of the existing `move_left` function.

```
def move_right(grid):
    """move the grid to the right by flipping the grid and moving left"""
    grid = reverse(grid)
    grid = move_left(grid)
    return reverse(grid)
```

The function simply flips the grid using our new function, calls the existing `move_left` function, and flips the result back before returning it.

> We won't test this, since each component is already tested so thoroughly. You could add a new test case for moving right if you wish.

Moving up

We can do vertical movement in a similar way. By flipping the grid on the diagonal (transposing), we can again use our `move_left` method to simulate an upward move.

So, for completeness, we can write our (final!) test.

```
class TestTranspose(unittest.TestCase):
    """Transpose should flip the grid on the diagonal"""
    def test(self):
        input  = [[ 1,  2,  3,  4],
                  [ 5,  6,  7,  8],
                  [ 9, 10, 11, 12],
                  [13, 14, 15, 16]]

        output = [[ 1,  5,  9, 13],
                  [ 2,  6, 10, 14],
                  [ 3,  7, 11, 15],
                  [ 4,  8, 12, 16]]
        self.assertEqual(core.transpose(input), output)
```

> Notice in our test, we move the first row into the first column and vice versa.

And create a new `transpose` method as follows:

```
def transpose(grid):
    """flip the grid diagonally"""
    return [list(col) for col in zip(*grid)]
```

> the new test should now pass

The method relies on a call to the built-in zip function within a list comprehension.

> The `zip` function returns an iterator which aggregates the first item from each row, collected together in a tuple and then the second and third and so on.

In this case, we are converting the tuples returned by `zip` (the columns of our grid) into a nested list.

> An alternative implementation is something like this:
>
> ```
> def transpose(grid):
>     """flip the grid diagonally"""
>     result = [list(r) for r in grid]
>     for row in range(4):
>         for col in range(4):
>             result[col][row] = grid[row][col]
>     return result
> ```
>
> This would also pass the test, but is obviously more complicated.

Now we can implement `move_up` which is similar to `move_right`.

```
def move_up(grid):
    """move up by transposing the grid and moving left"""
    grid = transpose(grid)
    grid = move_left(grid)
    return transpose(grid)
```

Moving down

To make a move in the downward direction is now easy. All we need to do is transpose and then call `move_right`.

```
def move_down(grid):
    """move down by transposing the grid and moving right"""
    grid = transpose(grid)
    grid = move_right(grid)
    return transpose(grid)
```

So, we now have a full set of functions that will allow us to make any of the moves we need. The next step is to build in the higher level game dynamics.

> Of course, you could add a few tests to confirm that moves in each direction were correctly applied.

Challenges

Think about what else we need before we can actually play a game of 2048.

- Randomised starting tiles (two 2's)
- Additional random tiles (either a 2 or a 4 is added each turn)
- user interaction to select the next move
- A scoring system
- Ability to detect game over (no more legal moves)
- Restart

Create a class `Game` in a new module `game.py`.

- Initialise the game with an instance variable `self.grid` set to an empty game grid and `self.score` set to zero.
- Add a `__str__` method to show the current game state.
- Try to set two random tiles to `2`
- start an infinite loop in which you:
    - print the game state to the terminal
    - get user input (using the `input` built-in function)
    - respond to specific commands (e.g. "W", "A", "S", "D") to make a move.