Lab 2: Merging left

This week, we will add some more code to build on what we created last week. Before you begin, copy the code from last week into a new folder `lab_02` and keep the original safe.

Last week we looked at a function called `stack_left` in which we moved all the non-empty tiles to the left as the first basic stage in implementing our 2048 game. We wrote some tests to define the behaviour we wanted from our function and you were left with a challenge to implement the function and make the tests pass.

This week, we will work on a new function (`merge_left`) that adds the ability to merge to the left. But first, lets look at the solution to last week's challenge.

> Hopefully you were able to attempt this challenge and perhaps you even succeeded. If so, well done. If you didn't get it working, don't worry, we will go through a few approaches now.

Solution

There are many ways to implement a function like this. Some are better than others in terms of speed or clarity of code. I will show you three ways, with my current preferred solution presented last.

A traditional approach would be to loop over the input list and write the data out to an output list like this.

```
def stack_left(row):
    """Move the non-None items in one row to the left"""
    result = [None, None, None, None]
    stack_index = 0
    for i in range(len(row)):
        if row[i] is not None:
            result[stack_index] = row[i]
            stack_index += 1
    return result
```

The function starts by creating a list of `None` values (`result`) which will be filled with data as needed and returned. It then initialises a variable `stacked_index` to zero. This represents the index (the position) in the `result` list which should be written next. So the idea is that we loop over the input list and write the non-None values out to the result list at the correct position, starting at zero.

The code loops over the input and checks each value to see if it has a numbered tile (i.e. whether it is not `None`). For each numbered tile, the value is copied into the `result` list at the `stack_index` position and `stack_index` is immediately incremented, ready for the next non-None value.

Finally, after all the input data has been checked and non-None values copied over, the `result` list is returned.

> Paste this code into your `core.py` module and run the tests. You should see that they all pass.

```
$ python3 test.py
.....
----------------------------------------------------------------------
Ran 5 tests in 0.000s

OK
```

However, you may have noticed some room for improvement. We can do better than this. A more *pythonic* version of the above might look like this:

```
def stack_left(row):
    """Move the non-None items in one row to the left"""
    result = [None, None, None, None]
    stack_index = 0
    for tile in row:
        if tile:
            result[stack_index] = tile
            stack_index += 1
    return result
```

This is similar but a bit neater and easier to read. The main improvement is that we have removed the counter index (`i`).

> Try it, the tests should still pass.

But if we think about the problem as a `sorting` problem, we might come up with an even simpler solution. Essentially, we want to sort the row such that all `None` elements are pushed to the end and everything else stays in the same order.

```
def stack_left(row):
    """move the non-None items in one row to the left"""
    return sorted(row, key=lambda tile: tile is None)
```

> Again, the tests should still pass.

In the above code, we are passing the input list `row` into the built-in `sorted` function with a custom `key` argument. The `key` argument is set to a function that returns `True` is the element is `None` and `False` otherwise. So the `sorted` function will push all values that return `True` to the end (since `True` or 1 is

greater than `False` or 0). This keeps our code shorter and simpler and so this is my current preferred approach.

> If you have a different approach that passes the tests, let me know.

Merging

OK, so we have a function that will stack the elements to the left. The next thing we will create is a function (we will call it `merge_left`) that will take the stacked elements and merge any pairs from right to left. The left tile of a pair always doubles and the right tile is set to `None`. For example, a pair of `2`'s would become a `4` and a `None`.

> We are not implementing the whole move here, only the merge. We will take the output of the `stack_left` function as an input, so our new function will always be dealing with values pre-stacked to the left. This makes testing much simpler and keeps our code clean.

We can clarify what we want by defining a few new tests. The first test is familiar. If there are no tiles to merge then a merge has no effect.

Add the following new `unittest.TestCase` class to your `tests.py` module.

```
class TestMergeLeft(unittest.TestCase):

    def test_empty(self):
        """An empty row is unaffected by a merge"""
        result = core.merge_left([None, None, None, None])
        self.assertEqual(result, [None, None, None, None])
```

Now, add the following function to your `core.py` module.

```
def merge_left(stacked_row):
    return [None, None, None, None]
```

As last week, we have started with a placeholder function to fail the tests. However, if you run the tests you should find all the tests pass (because we are not testing much yet).

```
$ python3 test.py
......
----------------------------------------------------------------------
Ran 6 tests in 0.000s

OK
```

Here are some more tests, check what each is doing and add them to your `TestMergeLeft` class.

> When there are no merges to make, our function should have no effect.

```
    def test_one_value(self):
        """An single value is unaffected by a merge"""
        result = core.merge_left([2, None, None, None])
        self.assertEqual(result, [2, None, None, None])

    def test_all_different(self):
        """Different values are unaffected by a merge"""
        result = core.merge_left([2, 4, 8, 16])
        self.assertEqual(result, [2, 4, 8, 16])
```

> For single merges, the impact of additional tiles should be clear. In particular, see the `test_value_after_pair` case. When there are three similar tiles, the left-most pair is merged and the third tile is left unchanged.

```
    def test_one_pair(self):
        """A single pair is simple"""
        result = core.merge_left([2, 2, None, None])
        self.assertEqual(result, [4, None, None, None])

    def test_value_before_pair(self):
        """Simple merge with an additional tile to the left"""
        result = core.merge_left([4, 2, 2, None])
        self.assertEqual(result, [4, 4, None, None])

    def test_value_after_pair(self):
        """Simple merge with an additional tile to the right"""
        result = core.merge_left([2, 2, 2, None])
        self.assertEqual(result, [4, None, 2, None])

    def test_larger_numbers(self):
        """A large pair with two tiles to the left"""
        result = core.merge_left([64, 128, 256, 256])
        self.assertEqual(result, [64, 128, 512, None])
```

> and finally, the case of two pairs to merge.

```
    def test_two_pairs(self):
        """Two pairs leaves a gap"""
        result = core.merge_left([2, 2, 2, 2])
        self.assertEqual(result, [4, None, 4, None])
```

Notice (e.g. in `test_value_after_pair` and `test_two_pairs`) that after the merge, we still haven't completed a full move. Don't worry, we will get there by the end of this lab session.

> The full move actually requires us to call `stack_left` one final time after the call to `merge_left`. Ask if you're not sure about this.

Getting the tests to pass

Running the new tests understandably results in a lot of failures.

> Spend a few minutes thinking about how to write this function so the tests pass. Again, there are a number of approaches which could be applied here.

There are only three possible pairs in a four-element list. Our code needs to check each pair in turn. Before we check a pair, we first need to check for a `None` value, which means we can return (because there will be

no further values).

A simple version goes like this. We check all possible pairs one at a time and handle the merges directly.

```
def merge_left(stacked_row):
    """Merge similar non-None items to the left"""

    # If the first pair match, merge them
    if stacked_row[0] and stacked_row[0] == stacked_row[1]:
        stacked_row[0] *=2
        stacked_row[1] = None

    # If the second pair match, merge them
    if stacked_row[1] and stacked_row[1] == stacked_row[2]:
        stacked_row[1] *=2
        stacked_row[2] = None

    # Finally check the final pair
    if stacked_row[2] and stacked_row[2] == stacked_row[3]:
        stacked_row[2] *=2
        stacked_row[3] = None

    # return the result
    return stacked_row
```

A more algorithmic approach is to loop over the three pairs and modify the data accordingly. This leads to a shorter method and is slightly more efficient and maintainable.

```
def merge_left(stacked_row):
    """Merge similar non-None items to the left"""
    for i in range(3):
        if stacked_row[i] and stacked_row[i] == stacked_row[i+1]:
            stacked_row[i] *= 2
            stacked_row[i + 1] = None
    return stacked_row
```

> The beauty of testing, is that you can try multiple variants to see if they pass the tests.
>
> To do this, simply rename the working method (e.g. to `old_merge_left`) and add a new method called `merge_left`.
>
> If your new method passes the tests then you can safely delete the old working code.
>
> If you can't get the tests to pass, delete the new code and revert the name back.
>
> This is particularly a good approach if using a version control system such as git.

Completing the move

With our `merge_left` function passing tests, we can finally implement a simple `move_left` function. The function will complete a full move to the left on one row of data.

Add the following function to your `core.py` module

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

We think this function works, but can we be sure?

Write a new `unittest.TestCase` class in the `tests.py` module to confirm that the new function does its job properly.

```
class TestMoveLeft(unittest.TestCase):

    def test_empty(self):
        """An empty row is unaffected by a merge"""
        result = core.move_left([None, None, None, None])
        self.assertEqual(result, [None, None, None, None])

    def test_value_after_pair(self):
        """The additional tile should be stacked into the merged pair"""
        result = core.move_left([2, 2, 2, None])
        self.assertEqual(result, [4, 2, None, None])

    def test_two_pairs(self):
        """Two pairs are both merged and stacked"""
        result = core.move_left([2, 2, 2, 2])
        self.assertEqual(result, [4, 4, None, None])
```

You should see that all the tests pass.

Challenges

Now we have the main logic for moving left in place. We have a function that converts a four-element list into a new four-element list according to the rules of the 2048 game.

Think about the next step. We need to upgrade our `move_left` function to take a 4 × 4 grid of tile data.

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