Lab 1: Introducing testing

This week, we will introduce tests using the unittest module. In python we can write code that calls our functions and methods to check we get the correct result for given inputs.

Imagine we have a simple function like this (in a file `my_module.py`):

```
def my_function(input):
    return input * 3
```

A simple test might look something like this (in `tests.py`):

```
import unittest

from my_module import my_function

class TestMyFunction(unittest.TestCase):

    def test_with_zero(self):
        result = my_function(0)
        self.assertEqual(result, 0)

    def test_with_one(self):
        result = my_function(1)
        self.assertEqual(result, 3)

    def test_with_two(self):
        result = my_function(2)
        self.assertEqual(result, 6)

    def test_with_a_billion(self):
        result = my_function(1000000000)
        self.assertEqual(result, 3000000000)

    def test_with_string(self):
        with self.assertRaises(TypeError):
            result = my_function('should fail with TypeError')

if __name__ == '__main__':
    unittest.main()
```

Notice that the above code is far longer than the function it tests. This is often the case with tests. To a novice programmer, the value of tests is not always obvious. Especially in a simple case like this.

> The final test fails, which indicates that the function is not quite ready.

There are two major benefits of tests. One, which we will see this week, is that it is possible to write the tests **before** we write the actual code. This allows us to get detailed information about which tests pass and which fail. So we can see what we need to do to improve our code. It also allows us to know when the code is finished.

> If the tests pass, it should be **good enough**. Though, there may be a better way to do it, at least we know that our code works.
>
> It worth also noting here that if your test pass and your code doesn't work, then you probably need to update your testing code or add a new test.

The second major benefit, is that the test are always there as your code develops. This means you can (and should) run the test regularly as your codebase evolves to catch situations where you accidentally break some aspect of your code (known as regression testing).

Of course, its perfectly possible to write bad or incomplete tests, but if you take as much care over the test code as the functional code of your project, then missing tests can be discovered and added as you go.

> When a new bug is discovered, it is common to write a new test to catch the situation and make sure it never happens again.

A data model

We will store the state of each tile as a simple integer.

In our final implementation, we will store these in a 2-dimensional list. That is, a list where each element represents a single row of tiles. Each row will be another list, where each element is a tile. For empty tiles, we will use the special `None` value.

So, this is an example of a starting position for a game.

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

For now, consider a simplification of the game in which we only deal with one row. We will use a list to represent the row.

> There is no need to write any code yet, the code examples here are just for illustration

An empty row, looks like this.

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

If we add a 2 into the third column, we get this.

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

Moving left

To get started, we will only consider the player moving left. We will not consider merges yet, only the basic movement of tiles. So let's think about some test cases we might want to use to define correct behaviour.

If our row is empty, the move has no effect.

```
input = [None, None, None, None]
expected_output = [None, None, None, None]
```

If there is a single 2, then it will move to the beginning of the list.

```
input = [None, None, 2, None]
expected_output = [2, None, None, None]
```

With more numbers, they stack up on the left.

```
input = [None, 2, None, 4]
expected_output = [2, 4, None, None]
```

Three values are similar

```
input = [None, 2, 4, 2]
expected_output = [2, 4, 2, None]
```

And with four values, nothing happens.

```
input = [4, 2, 4, 2]
expected_output = [4, 2, 4, 2]
```

With these simple test cases, we can write a few tests.

Testing the simplest case

We will write a function into a file `core.py` and our tests into `tests.py`. You should create a folder for `lab_01` and place these files **inside** that folder.

In `core.py` create a function called `stack_left` to indicate that it doesn't handle merges. The function should take one argument `row` which will contain the input list of tiles (i.e. integers or `None`). For now, simply return a blank row, so all our test cases should fail except the first one.

```
"""Functions implementing the core behaviour of the 2048 tile grid"""

def stack_left(row):
    """Move the non-None items in one row to the left"""
    return [None, None, None, None]
```

In `tests.py` we need to import this function from the `core` module and also import `unittest` so we can write our first basic test.

```
"Tests for the core 2048 functions"

import unittest

import core

class TestStackLeft(unittest.TestCase):

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

if __name__ == '__main__':
    unittest.main()
```

Running the tests is handled by calling `unittest.main()`, so simply executing the `test.py` module will run all the tests.

```
$ python3 test.py
.
----------------------------------------------------------------------
Ran 1 test in 0.000s

OK
```

Our test passed!

> The `unittest` module will treat all methods beginning with the word `test` as unit tests. So remember that you **must** name your methods accordingly.

Let's add the second test case as a new method.

```
"Tests for the core 2048 functions"

import unittest

import core

class TestStackLeft(unittest.TestCase):

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

    def test_one_value(self):
        "A single non-None tile should be moved to the left"
        result = core.stack_left([None, None, 2, None])
        self.assertEqual(result, [2, None,  None, None])

if __name__ == '__main__':
    unittest.main()
```

Now, running the tests produces a more detailed output.

```
$ python3 test.py
.F
======================================================================
FAIL: test_one_value (__main__.TestStackLeft)
A single non-None tile should be moved to the left
----------------------------------------------------------------------
Traceback (most recent call last):
  File "test.py", line 17, in test_one_value
    self.assertEqual(result, [2, None,  None, None])
AssertionError: Lists differ: [None, None, None, None] != [2, None, None, None]

First differing element 0:
None
2

- [None, None, None, None]
?              ------

+ [2, None, None, None]
?  +++


----------------------------------------------------------------------
Ran 2 tests in 0.001s

FAILED (failures=1)
```

Study the output. It's clearly telling us that the 'Lists differ'. It even shows us the detail of the differences.

```
AssertionError: Lists differ: [None, None, None, None] != [2, None, None, None]
```

Our function returned a list of `None` values, `[None, None, None, None]` when the test specified we should be returning `[2, None, None, None]`.

This is good, this is exactly what we want to see.

A more complete test case

Let's add more tests and see if we can make them pass.

```
"Tests for the core 2048 functions"

import unittest

import core

class TestStackLeft(unittest.TestCase):

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

    def test_one_value(self):
        "A single non-None tile should be moved to the left"
        result = core.stack_left([None, None, 2, None])
        self.assertEqual(result, [2, None,  None, None])

    def test_two_values(self):
        "Two non-None tiles should retain their order"
        result = core.stack_left([None, 2, None, 4])
        self.assertEqual(result, [2, 4,  None, None])

    def test_three_values(self):
        "Three non-None tiles should retain their order"
        result = core.stack_left([None, 2, 4, 2])
        self.assertEqual(result, [2, 4, 2, None])

    def test_four_values(self):
        "All non-None tiles should not move"
        result = core.stack_left([4, 2, 4, 2])
        self.assertEqual(result, [4, 2, 4, 2])


if __name__ == '__main__':
    unittest.main()
```

Each test case covers a slightly different situation and, if we write a sensible function, these should be enough to test the basics.

> Note that if these tests pass, it doesn't guarantee that the code is bug free. All it does is test a few inputs to see if the expected outputs are given. It is possible to write a very dumb and complicated function that passes these tests but doesn't suit our needs.

If we run the tests, the output includes a long list of failures. Only the first test case passes because our function returns `[None, None, None, None]` every time, no matter what input is provided.

Challenges

Spend the rest of the session asking questions to make sure you understand what we are doing and writing a function that passes all the tests. It should convert the input list into a modified list in which all the `None` values are pushed to the end and the numbered tiles are moved to the left.

If you get the tests passing, well done. A solution will be provided next week. Start thinking about how to implement and test the next steps.

- merge tiles with the same number
- expand to a 2-dimensional grid

We will cover these next week.

Each week we will build on the code from the previous week. So the code from this week will be needed next week and its very important you understand what we are doing here. We will continue to develop both python modules, `core.py` will include the core functions to handle movement etc. and `tests.py` will include the tests for these functions.

A good approach might be to copy the whole folder each week and rename it (e.g. lab_01, lab_02, etc.) to that each week, you begin with a copy of the previous weeks code.