Writing Parameterised Tests

In unit 3, we wrote several tests for a single component of production code: the function called 'how_many_shared_lists'.

These tests were all separated, so they looked something like this:

```
def test_with_one_shared_list():

    my_list = [42]

    list_a = [1,2,my_list]
    list_b = [4,5,my_list]

    expected_result = 1
    actual_result = mem.how_many_shared_lists(list_a, list_b)

    assert expected_result==actual_result

def test_with_no_shared_lists():

    my_list = [42]
    your_list = [42]

    list_a = [1,2,my_list]
    list_b = [4,5,your_list]

    expected_result = 0
    actual_result = mem.how_many_shared_lists(list_a, list_b)

    assert expected_result==actual_result
```

We can use a pytest decorator to provide a single test function with a list of argument tuples, so that the function will be repeatedly executed, once for each argument tuple.

The format of this decoration, and function arguments, is as follows:

```
argument_string = 'input1, input2, expected_result'
@pytest.mark.parametrize(argument_string, list_of_argument_tuples)
def test_function(input1, input2, expected_result):
    # use input1 and input2 to get an actual_result

    # compare with expected result
```

This leads to the following re-factor of the original test functions:

```
import pytest

class Store:

    my_list = [42]
    your_list = [42]
    tests = []
    tests.append(
        (
            [1,2,my_list], # list_a
            [3,4,my_list], # list_b
            1              # expected_result
        )
    )

    tests.append(
        (
            [1,2,my_list],  # list_a
            [3,4,your_list],# list_b
            0               # expected_result
        )
    )

@pytest.mark.parametrize("list_a, list_b, expected_result",
                         Store.tests)
def test_for_shared_lists(list_a, list_b, expected_result):

    actual_result = mem.how_many_shared_lists(list_a, list_b)
    print(list_a, list_b)
    assert actual_result == expected_result
```

We have chosen to enclose the list of argument tuples in a class named 'Store', so that they are easier to locate.

One advantage of using this form of tests (as opposed to iterating through these parameters within a single test) is that each parameterized test is considered as an individual test: The above block results in two unit tests:

This level of granularity is useful for logging and reporting of tests: the specific parameters associated with a failing test are automatically included in the error message.

In addition, since these three versions are considered separate independent tests, they can be run in parallel (on appropriate architectures)

Concept check: Parameterizing Unit Tests

Below are three test functions that check to see that how_many_shared_lists raises a ValueError when either list contains anything other than a list or an integer. Use the pytest.mark.parametrize decorator to write these three tests as a single test function.

```
def test_invalid_input1():

    shared_item = {'a':1}

    list_a = [1,2,shared_item]
    list_b = [4,5,shared_item]

    try:
        mem.how_many_shared_lists(list_a, list_b)
        assert False
    except ValueError:
        pass
    except Exception:
        assert False

def test_invalid_input2():

    shared_item = [42]

    list_a = (1,2,shared_item)
    list_b = (4,5,shared_item)

    try:
        mem.how_many_shared_lists(list_a, list_b)
        assert False
    except ValueError:
        pass
    except Exception:
        assert False

def test_invalid_input3():

    shared_item = [42]

    list_a = (True,False,shared_item)
    list_b = (True,False,shared_item)

    try:
        mem.how_many_shared_lists(list_a, list_b)
        assert False
    except ValueError:
        pass
    except Exception:
        assert False
```