Chapter 12 - Testing Object-Oriented Programs

April 4, 2021

0.1 Why Testing

- testing might be more important in python then `c++` because of dynamic typing
- automated tests automatically run certain inputs through other programs or parts of programs

Four Principles of Maintainable Code: - ensures that code is working the way the developer thinks it should - ensures that code continues working when we make changes - ensure that developer understood the requirements - ensure that the code we are writting has a maintainable interface

0.2 Test-driven development

- this is the *write tests first* before code ideology
- test-driven development takes *untested code is broken code* concept a step further by saying that only unwritten code should be untested
- we dont write any code untill we have the test that will prove it works

- the first goal of test driven metholodgy is to ensure that the tests reall get written
- secondly, writing tests first forces us to consider exactly how the code will be used
  - it tells us what methods objects need to have and how attributes will be accessed
  - it helps us break up the initial problem into smaller, testable problems and then to recombine the tested solutions into larger, also testes solutions

0.3 Unit Tests

- this is pythons built in test libary
- `unit tests` focus on testing the least amount of code possible in any one test
- the most important tool is the `TestCase` class
- this class provides a set of methods that allow us to compare values, set up tests and clean up when they have finished
- when we want to write a set of unit tests for a specific task, we create a subclass of `TestCase` and write individual methods to do the actual testing
- these methods must all start with the name `test`

[ ]:
```
import unittest

class CheckNumbers(unittest.TestCase):
    def test_int_float(self):
        self.assertEqual(1, 1.0)

if __name__ == "__main__":
```

```
    unittest.main()
```

- the code above simply subclasses the `TestCase` class and adds a method that calls the `TestCase.assertEqual` method
- as long as each method begins with `test`, we can write as many tests as we want
- each test howerver should only do one thing
- good testing requires keeping each test method as short as possible, testing a small unit of code with each test case

0.4 Assertion Methods

- the general layout of a test case is to set certain variables to know values, run one or more functions, methods or processes and then *prove* that correct expected results were returned or calculated by using `TestCase` assertion methods

- the most common assertion methods `assertEqual` or `assertNotEqual` will not test boolean values
- the `assertRaises` method can be used to ensure that specific function call raises a specific exception or, optionally, it can be used as a context manager to wrap inline code
- the test passes if the code inside the `with` statement raises the proper exception, otherwise it fails

- the context manager allows us to write the code the way we would normally write it (by calling functions or executing core directly), rather than having to wrap the function call in another function call

Methods	**Description**

`assertGreater`
`assertGreaterEqual`
`assertLess`
`assertLessEqual`
	Accept two comparable objects and ensure the named inequality holds.

`assertIn` `assertNotIn`	Ensure an element is (or is not) an element in a container object.
`assertIsNone` `assertIsNotNone`	Ensure an element is (or is not) the exact `None` value (but not another falsey value).
`assertSameElements`	Ensure two container objects have the same elements, ignoring the order.

`assertSequenceEqualassertDictEqual`
`assertSetEqual`
`assertListEqual`
`assertTupleEqual`
	Ensure two containers have the same elements in the same order. If there's a failure, show a code difference comparing the two lists to see where they differ. The last four methods also test the type of the list.

0.5 Reducing Boilerplate and cleaning up

- we can use `setUp` method on the `TestCase` class to perform initialization for each test
- the `setUp` or `tearDown` methods do not need to be called inside of our methods, but they always run

0.6 Organizing and running tests

- we should divide our test `classes` into modules and packages that keep them organized
- if we name each test module starting with `test`, there is an easy way to find and run them all
- pythons `discover` module looks for any modules in the current folder or `subfolders` with names that start with `test`
- if it finds any `TestCase` objects in these modules, the tests are executed
- to use this feature name your file `test_<something>.py`
- then run `python3-munittestdiscover`

0.7 Ignoring Broken Tests

python has the following decorators to skip: - `expectedFailure()` - simply tess test runner to ignore this test if it fails - `skip(reason)` - does not even run the test, requires string saying why test failed - `skipIf(condition, reason)` - uses a basic comparision operators shuch as `==` - `skipUnless(condition, reason)`

[5]:
```python
import unittest
import sys

class SkipTests(unittest.TestCase):
    @unittest.expectedFailure
    def test_fails(self):
        self.assertEqual(False, True)

    @unittest.skip("Test is useless")
    def test_skip(self):
        self.assertEqual(False, True)

    @unittest.skipIf(sys.version_info.minor == 4, "broken on 3.4")
    def test_skipif(self):
        self.assertEqual(False, True)

    @unittest.skipUnless(
    sys.platform.startswith("linux"), "broken unless on linux"
    )
    def test_skipunless(self):
        self.assertEqual(False, True)

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

```
E
======================================================================
ERROR: C:\Users\Vicktree\AppData\Roaming\jupyter\runtime\kernel-63948ab3-2fef-45
f6-964b-3dd7e7c54dca (unittest.loader._FailedTest)
----------------------------------------------------------------------
AttributeError: module '__main__' has no attribute 'C:\Users\Vicktree\AppData\Ro
```

```
aming\jupyter\runtime\kernel-63948ab3-2fef-45f6-964b-3dd7e7c54dca'

----------------------------------------------------------------------
Ran 1 test in 0.001s

FAILED (errors=1)
```

```
An exception has occurred, use %tb to see the full traceback.

SystemExit: True
```

```
c:\users\vicktree\appdata\local\programs\python\python39\lib\site-
packages\IPython\core\interactiveshell.py:3445: UserWarning: To exit: use
'exit', 'quit', or Ctrl-D.
  warn("To exit: use 'exit', 'quit', or Ctrl-D.", stacklevel=1)
```

0.8 Testing with Pytest

- `unittest` requires alot of boilerplate code and is an example of overusing OOP
- `Pytest` does not require test cases to be classes
- it takes advantage of the fact that python functions are objects and allows any properly named function to behave like a test
- instead of providing a bunch of custom methods for asserting equality, it uses the `assert` statement to verify results

- when we run pytest, it starts in the current folder and searches for any modules or subpackages with names beginning with the character `test_`
- if any functions in the module also start with the `test` they will be executed
- if there area any classes in the module whoses name starts with `Test` any methods on that class that start with `test_` will also be executed
- pytest suppresses output from `print` statements if the test is successful

[7]:
```python
# notice how simple a test is to write

def test_int_float():
    assert 1 == 1.0

# we could also use a class

class TestNumbers:
    def test_int_float(self):
        assert 1 == 1.0

    def test_int_str(self):
        assert 1 == "1"
```

0.9 One way to do setup and cleanup

- pytest provides more flexability for the `setup` and `teardown` methods
- we can use `setup_method` and `teardown_method`
- the one difference between the pytest startup/teardown is that both methods accept an argument: the function object representing the method being called

- we have additional `setup_class` and `teardown_class` methods are expected to be class methods and they accept a single argument representing the class in question
- these methods only run when the class is initaited rather than on each test run

- we also have the `setup_module` and `teardown_module` functions, which are run immediately before and after all tests (in functions or classes) in that module
- these are useful for `one time` setup, such as creating a socket or database connection that will be used by all tests in the module
- be careful with this one as it can introudce dependencies between tests if the object stores state that is not cleaned up

[9]:
```
def setup_module(module):
    print(f'setting up MODULE {module.__name__}')


def teardown_module(module):
    print(f'tearing down Modulle {module.__name__}')


class BaseTest:
    def setup_class(cls):
        print("setting up CLASS {cls.__name__}")

    def teardown_class(cls):
        print(f'tearing down CLASS {cls.__name__}')

    def setup_method(self, method):
        print(f'tearing down METHOD {method.__name__}')

class TestCase1(BaseTest):
    def test_method_1(self):
        print(f'Running METHOD 1-1')

    def test_method_2(self):
        print(f'RUNNING METHOD 1-2')


class TestClass2(BaseTest):
    def test_method_1(self):
        print(f'RUNNING METHOD 2-1')

    def test_method_2(self):
```

```python
        print(f'RUNNING METHOD 2-2')
```

- the purpose of BaseTest class is to extract four methods that are otherwise identical to the test classes, and use inheritance to reduce the amount of duplicate code
- from the point of view of pytest the two subclasses have not only two test methods each, but also have two setup and two teardown methods

0.10 A completely different way to seup up variables

- one of the most common uses for the various setup and teardown functions is to ensure certain class or module variables are available with a known value before each test method is run
- pytest offers a completely different way of doing this, using what is known as fixtures
- fixtures are basically named variables that are predefined in a test configuration file
- this allows us to seprate configuration from the execution of tests, and allows fixtures to be used across multiple classes and modules
- to use them, we add parameters to our test function
- the names of the parametes are used to look up specific arguments in specially named functions

[10]:
```python
import pytest
from stats import StatsList

@pytest.fixture
def valid_stats():
    return StatsList([1, 2, 2, 3, 3, 4])

def test_mean(valid_stats):
    assert valid_stats.mean() == 2.5
```

```
---------------------------------------------------------------------------
ModuleNotFoundError                       Traceback (most recent call last)
<ipython-input-10-691cc52be896> in <module>
----> 1 import pytest
      2 from stats import StatsList
      3
      4 @pytest.fixture
      5 def valid_stats():

ModuleNotFoundError: No module named 'pytest'
```

- each of the three test methods accepts a parameter named valid_stats
- the parameter is created by calling the valid_stats function which was decorated with @pytest.fixture

- fixtures can do alot more than return basic variables
- a request object can be passed into the fixture factory provide useful infomation

- `module`, `cls` and `function` attribute allow us to see exactly which test is requesting the fixture
- the `config` attribute allows us to check command-line arguments and a great deal of other configuration data

- the fixture creates a new empty temporary directory for files to be created in
- it yeilds this for use in the test, but removes that directry using `shutil.rmtree` which recursively removes a directory and anything inside it

[11]:
```python
import pytest
import tempfile
import shutil
import os.path

@pytest.fixture
def temp_dir(request):
    dir = tempfile.mkdtemp()
    print(dir)
    yeild dir
    shutil.rmtree(dir)

def test_osfiles(temp_dir):
    os.mkdir(os.path.join(temp_dir, "a"))
    dir_contents = os.listdir(temp_dir)
    assert len(dir_contents) == 2
    assert "a" in dir_contents
    assert "b" in dir_contents
```

```
  File "<ipython-input-11-b29b2be3385e>", line 10
    yeild dir
          ^

SyntaxError: invalid syntax
```

- we can pass a `scope` parameter to create a fixture that lasts longer than one test
- this is useful when setting up an expensive operation that can be reused multiple tests
- temember that the resource reuse does not break the unit nature of tests
- the scope can be one of the string `class`, `module`, `package` or `session`
- it determines how long the arguments will be cached
- the `session` caches it for the duration of the `pytest` run but the `module` only caches it for test in that module

[ ]:
```python
@pytest.fixture(scope="session")
def echoserver():
    print("loading server")
    p = subprocess.Popen(["python3", "echo_server.py"])
    time.sleep(1)
```

```
    yield p
    p.terminate()
```

0.11 Skipping tests with Pytest

- `pytest.skip` function can skip a test
- it has a single argument, which is a string describing why it was skipped
- if called insite the function, the function is skipped
- if called on the module level, all the tests in that module will be skipped
- if we call it inside a fixture, all tests that call that funcarg will be skipped

[12]:

```
import sys
import pytest

def test_simple_skip():
    if sys.platform != "fakeos":
        pytest.skip("Test works only on fakeOS")
    fakeos.do_something_fake()
    assert fakeos.did_not_happen

# mark.skipif behaves simmilar to expectedFailure()
# if xfail is not supplied, it will be expected to fail under all situations
@pytest.mark.skipif("sys.version_info <= (3.0)")
    def test_python3():
        assert b"hello".decode() == "hello"
```

```
---------------------------------------------------------------------------
ModuleNotFoundError                       Traceback (most recent call last)
<ipython-input-12-362c9ccc587a> in <module>
      1 import sys
----> 2 import pytest
      3
      4 def test_simple_skip():
      5     if sys.platform != "fakeos":

ModuleNotFoundError: No module named 'pytest'
```

0.12 Imitating Expensive Objects

- imagine we need to use an `API`
- we can use `Mock()` objects in out test to replace the roublesome methods with an object we can introspect
- we create a `Mock` object for the `set` method and make sure that it is never called

[13]:
```python
from flight_status_redis import FlightStatusTracker
from unittest.mock import Mock
import pytest

@pytest.fixture
def tracker():
    return FlightStatusTracker()

def test_mock_method(tracker):
    tracker.redis.set = Mock()

    with pytest.raises(ValueError) as ex:
        tracker.change_status("AC101", "lost")
    assert ex.value.args[0] == "Lost is not a valid status"
    assert tracker.redis.set.call_count  = 0
```

```
  File "<ipython-input-13-a39b5d75f8e3>", line 15
    assert tracker.redis.set.call_count  = 0
                                         ^
SyntaxError: invalid syntax
```

- temporarily setting a library function to a specifc value is one of the few valid use cases for monkey-patching
- the mock library provides a patch context manager that allows us to replace attributes on existing libraries with mock objects
- when the context manager exits, the orginal attribute is automatically restored so as not to impact other test cases

[ ]:
```python
import datetime
from unittest.mock import patch

def test_patch(tracker):
    tracker.redis.set = Mock()
    fake_now = datetime.datetime(2015, 4, 1)
    with patch("datetime.datetime") as dt:
        dt.now.return_value = fake_now
        tracker.change_status("AC102", "on time")
    dt.now.assert_called_once_with()
    tracker.redis.set.assert_called_once_with(
        "flightno:AC102", "2015-04-01T00:00:00|ON TIME"
    )
```

- we first construct a value called `fake_now`, which we set as the return value of the `datetime.dsatetime.now` function
- we have to construct the object before we patch `datetime.datetime`, because otherwise we'd be calling the patched now function before we constructed it

- the `with` statement invites the patch to replace the `datetime.datetime` module with a mock object, which is returned as `dt` value
- the neat thing about a mock object is that you anytime you access an attribute or method on that object, it return another mock object
- this when we accessed `dt.now`, it gives us a new mock object
- we set the `return_Value` of that object to our `fake_now` object
- when ever the `datetime.datetime.now` function is called, it will return our object instead of a new mock object
- when the context manager is exited, the orginal `datetime.datetime.now()` functionality is restored

- if we find ourselves mocking out multiple elements in a given unit test, we should rethink it

0.13 How much testing is enough

- how much of our code is actually being tested is easy to verify, we can just use the `coverage` coverage

- `coverage run coverage_unittest.py`
  - generates a `.coverage` file
- `coverage report`
  - shows the coverage
- `coverage html`
  - shows us the coverage in the html with more info
- we can use the coverage module with `pytest` as well