PYTHON MARATHON

TRAIN YOUR BRAIN

softserve | IT academy

Unit testing

softserve

Agenda

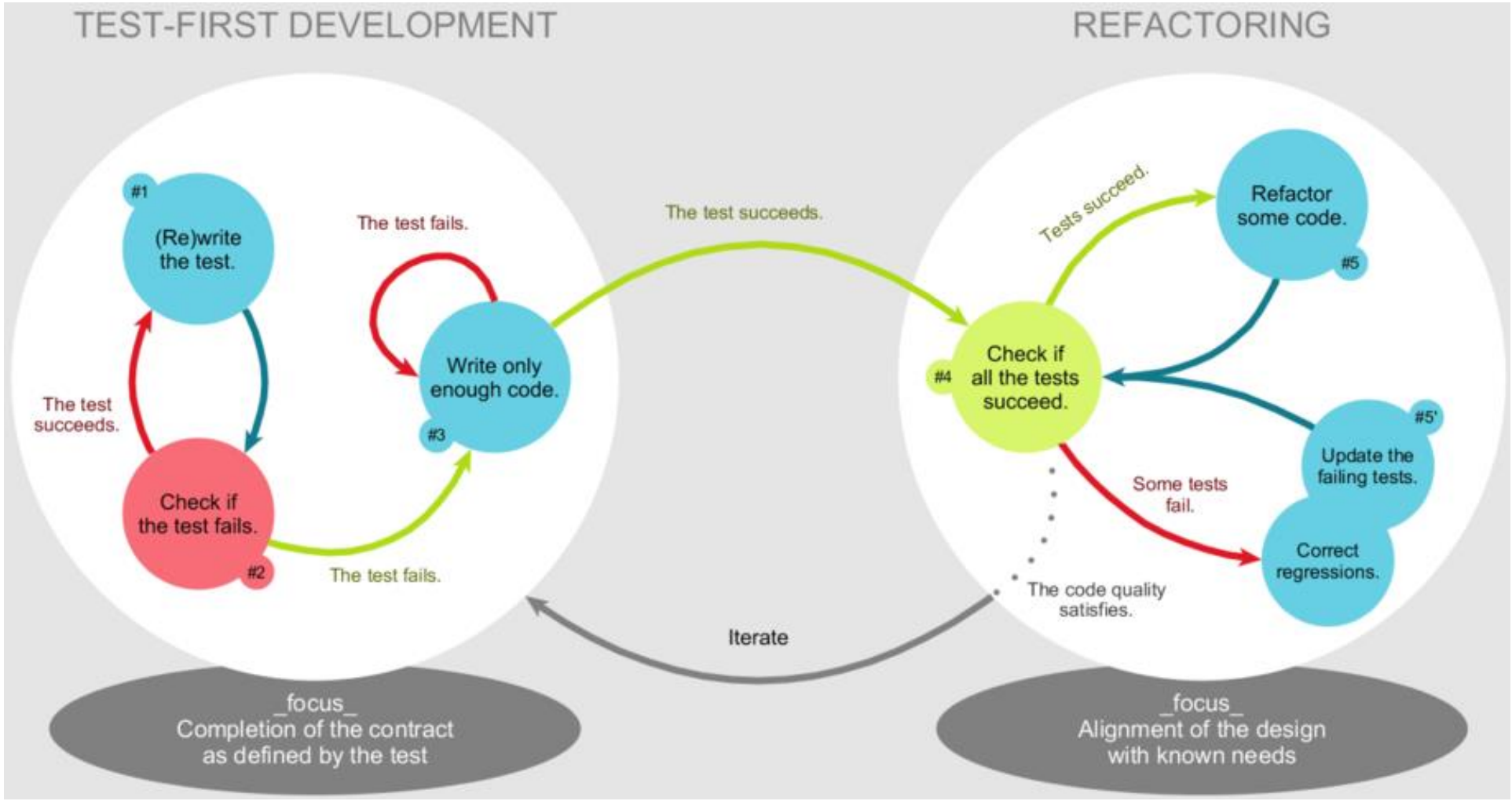
- Testing levels
- TDD
- Test structure
- Unittest libreary
- Pytest library
- Mock objects

softserve

Testing Levels

- ***End-to-end***: validate the flow through the entire system, from the ultimate source to the ultimate sink.
- ***Integration***: validate the integration between multiple components.
- ***Functional/Component***: A functional test is supposed to validate the behavior of a single process or daemon. In practice, the difference between integration and functional/component testing isn't very significant.
- ***Unit***: Test the isolated logic of a small piece of code (e.g., a single method of a python class). These tests are automated: a single invocation runs all unit tests for a product, complete with counts of passed/failed tests and information (what broke, traceback) on failed tests. Unit tests should have no external dependencies other than the specific code being tested. External resources should be mocked up when necessary.

TDD - Test Driven Development



softserve

General Rules of Testing Your Code

- A testing unit should focus on one tiny bit of functionality
- Each test unit must be fully independent
- Make tests that run fast (heavier tests in a separate test suite)
- Learn your tools and learn how to run a single test or a test case
- Run all tests before pushing code to a shared repository
- If you have to interrupt your work, write a broken unit test about what you want to develop next.
- The first step of debugging is to write a new test pointing the bug
- Use long and descriptive names for testing functions (BDD & BDD)
- The testing suite is a basis for fixing or modifying your code
- The testing code is as an introduction to new developers

softserve

See for details: http://docs.python-guide.org/en/latest/writing/tests/

Basic Test Structure - AAA

AAA (Arrange-Act-Assert) is a pattern for arranging and formatting code in Unit Test methods:

1) **Arrange** all necessary preconditions and inputs. (given)
2) **Act** on the object or method under test. (when)
3) **Assert** that the expected results have occurred. (then)

Benefits:

- Clearly separates what is being tested from the setup and verification steps.
- Clarifies and focuses attention on a historically successful and generally necessary set of test steps.
- Makes some Test Smells more obvious:
  - Assertions intermixed with "Act" code.
  - Test methods that try to test too many different things at once.

Unittest Library

unittest is the batteries-included test module in the Python standard library. Its API will be familiar to anyone who has used any of the JUnit/nUnit/CppUnit series of tools.

unittest supports some important concepts in an object-oriented way:

A **test fixture** represents the preparation needed to perform one or more tests, and any associated cleanup actions. This may involve, for example, creating temporary or proxy databases, directories, or starting a server process.

A **test case** is the individual unit of testing. It checks for a specific response to a particular set of inputs. unittest provides a base class, TestCase, which may be used to create new test cases.

A **test suite** is a collection of test cases, test suites, or both. It is used to aggregate tests that should be executed together.

A **test runner** is a component which orchestrates the execution of tests and provides the outcome to the user. The runner may use a graphical interface, a textual interface, or return a special value to indicate the results of executing the tests.

Unittest Library

A testcase is create by subclassing *unittest.TestCase*. All individual tests must be defined with methods whose names start with the letters ***test***. This naming convention informs the *test runner* about which methods represent tests.

```
import unittest

class TestStringMethods(unittest.TestCase):

    def test_upper(self):
        self.assertEqual('foo'.upper(), 'FOO')

    def test_isupper(self):
        self.assertTrue('FOO'.isupper())
        self.assertFalse('Foo'.isupper())

    def test_split(self):
        s = 'hello world'
        self.assertEqual(s.split(), ['hello', 'world'])
        # check that s.split fails when the separator is not a string
        with self.assertRaises(TypeError):
            s.split(2)

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

unittest.main() provides a command-line interface to the test script

```
test_isupper (__main__.TestStringMethods) ... ok
test_split (__main__.TestStringMethods) ... ok
test_upper (__main__.TestStringMethods) ... ok

----------------------------------------------------------------------
Ran 3 tests in 0.001s

OK
```

softserve

Unittest Library
Basic Assertions

Assertion	Operation
`assertTrue(a, M)` `assertFalse(a, M)`	a = True a = False
`assertEqual(a, b, M)` `assertNotEqual(a, b, M)`	a = b a ≠ b
`assertIs (a, b, M)` `assertIsNot (a, b, M)`	a is b a is not b
`assertIsNone(a, M)` `assertIsNotNone(a, M)`	a = None a ≠ None
`AssertIsInstance(a, b, M)` `AssertIsNotInstance(a, b, M)`	isinstance(a, b) not isinstance(a, b)

softserve

Unittest Library
Comparative Assertions

Assertion	Operation
`assertGreater(a, b, M)` `assertLess(a, b, M)`	a > b a < b
`assertGreaterEqual(a, b, M)` `assertLessEqual(a, b, M)`	a ≥ b a ≤ b
`assertAlmostEqual(a, b, n, M)` `assertNotAlmostEqual(a, b, n, M)`	round(a-b, n) = 0 round)a-b, n) ≠ 0
`assertItemsEqual(a, b, M)`	sort(a) = sort(b); sort(a) ≠ sort(b)

softserve

Unittest Library
Assertions for Collections

Assert	Operation
`assertIn(a, b, M)` `assertNotIn(a, b, M)`	a in b a not in b
`assertDictContainsSubset(a, b, M)`	a has b
`assertDictEqual(a, b, M)`	a = b
`assertListEqual(a, b, M)`	a = b
`assertSetEqual(a, b, M)`	a = b
`assertSequenceEqual(a, b, M)`	a = b
`assertTupleEqual(a, b, M)`	a = b
`assertMultilineEqual(a, b, M)`	a = b

softserve

Unittest Library

Working environment for the testing code is called a ***test fixture***

A new TestCase instance is created as a unique test fixture used to execute each individual test method.

setUp method called to prepare the test fixture. This is called immediately before calling the test method.

tearDown method called immediately after the test method has been called and the result recorded. This is called even if the test method raised an exception.

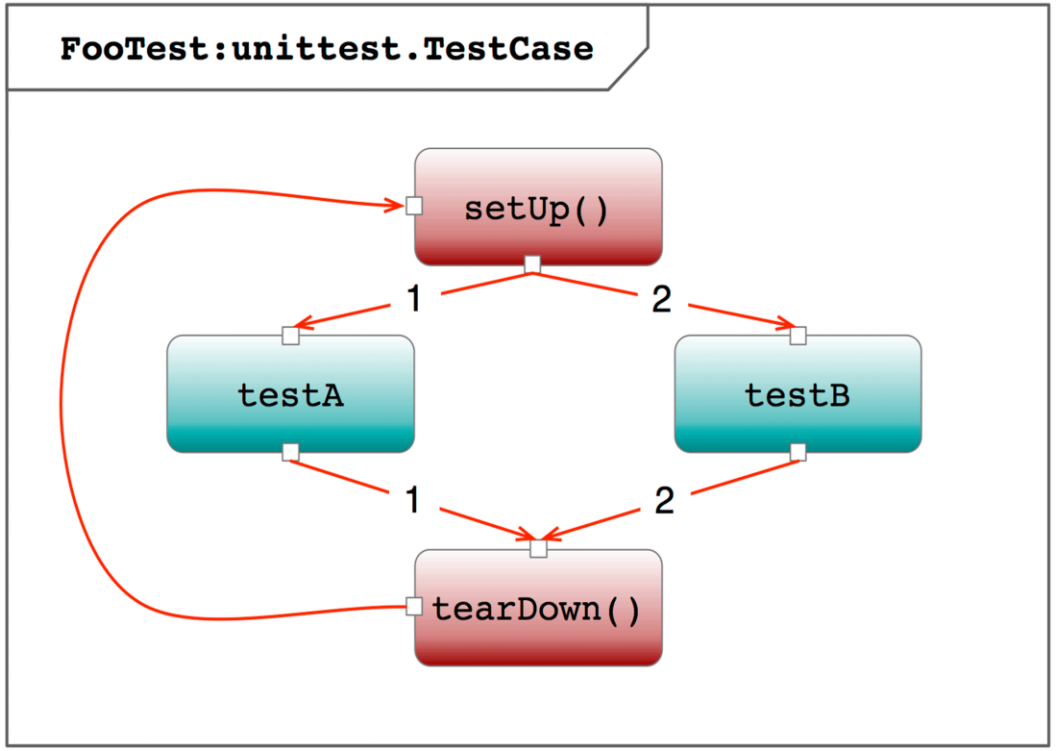
setUpClass and ***tearDownClass*** A class methods called before and after tests in an individual class are run. This methods is called with the class as the only argument and must be decorated as a *@classmethod*

setUpModule and ***tearDownModule*** should be implemented as functions. If an exception is raised in a setUpModule then none of the tests in the module will be run and the

tearDownModule will not be run

Unittest Library

```python
import unittest

class FooTest(unittest.TestCase):
    """Sample test case"""

    # preparing to test
    def setUp(self):
        """ Setting up for the test """
        print("FooTest:setUp_:begin")
        # do something...
        print("FooTest:setUp_:end")

    # ending the test
    def tearDown(self):
        """Cleaning up after the test"""
        print("FooTest:tearDown_:begin")
        # do something...
        print("FooTest:tearDown_:end")

    # test routine A
    def testA(self):
        """Test routine A"""
        print("FooTest:testA")

    # test routine B
    def testB(self):
        """Test routine B"""
        print("FooTest:testB")
```



softserve

Mock Objects

Mock objects are simulated objects that mimic the behavior of real objects in controlled ways. A computer programmer typically creates a mock object to test the behavior of some other object

A Python mock object contains data about its usage that you can inspect such as:

- If you called a method
- How you called the method
- How often you called the method

The Python mock object library is unittest.mock provides a class called Mock which you will use to imitate real objects in your codebase. Mock offers incredible flexibility and insightful data

Mock Objects

```
>>> mock = Mock(side_effect=Exception)
>>> mock()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/usr/local/Cellar/python/3.6.5/Frameworks/Python.
    return _mock_self._mock_call(*args, **kwargs)
  File "/usr/local/Cellar/python/3.6.5/Frameworks/Python.
    raise effect
Exception

>>> mock = Mock(name='Real Python Mock')
>>> mock
<Mock name='Real Python Mock' id='4434041432'>

>>> mock = Mock(return_value=True)
>>> mock()
True
```

Mock and MagicMock objects create all attributes and methods as you access them and store details of how they have been used. You can configure them, to specify return values or limit what attributes are available, and then make assertions about how they have been used. You can configure a Mock by specifying certain attributes when you initialize an object.

Mock instances store data on how you used them. For instance, you can see if you called a method, how you called the method

.assert_called() ensures you called the mocked method

Mock Objects

unittest.mock provides a powerful mechanism for mocking objects, called patch(), which looks up an object in a given module and replaces that object with a Mock.

patch() as a Decorator

```
class TestCalendar(unittest.TestCase):
    @patch('my_calendar.requests')
    def test_get_holidays_timeout(self, mock_requests):
            mock_requests.get.side_effect = Timeout
            with self.assertRaises(Timeout):
                get_holidays()
                mock_requests.get.assert_called_once()
```

patch() as a Context Manager

```
class TestCalendar(unittest.TestCase):
    def test_get_holidays_timeout(self):
        with patch('my_calendar.requests') as mock_requests:
            mock_requests.get.side_effect = Timeout
            with self.assertRaises(Timeout):
                get_holidays()
                mock_requests.get.assert_called_once()
```

softserve

Pytest library

Pytest is a Python library for testing Python applications. It is an alternative to unittest.

Pytest is installed with the command *$ pip install pytest*

Running pytest with command *$ pytest <test_file.py>*

If no arguments, pytest looks at the current working directory (or some other preconfigured directory) and all subdirectories for test files and runs the test code it finds. If with arguments running all test files in the current directory.

Pytest expects our tests to be located in files whose names begin with ***test_*** or end with ***_test.py***
Function names begin with ***test_*** or ***Test_*** for classes

Pytest library

You can use the standard python assert statement to verify test expectations.

In order to write assertions about raised exceptions, you can use pytest.raises as a context manager

```python
def sum(a, b):
    return a + b


def test_sum():
    assert sum(1, 2) == 3


class TestSum:
    def test_sum(self):
        assert sum(3, 4) == 7
```

```python
def raise_error():
    raise ValueError


def test_raise_error():
    with pytest.raises(ValueError):
        raise_error()
```

softserve

Pytest library

Software test fixtures initialize test functions. They provide a fixed baseline so that tests execute reliably and produce consistent, repeatable, results. Initialization may setup services, state, or other operating environments. These are accessed by test functions through arguments; for each fixture used by a test function there is typically a parameter (named after the fixture) in the test function's definition.

```
@pytest.fixture(scope='module')
def setup():
    return [5, 2]


def sum(a, b):
    return a + b


def test_sum(setup):
    assert sum(setup[0], setup[1]) == 7
```

A scope="module" parameter to the @pytest.fixture invocation to cause the decorated setup fixture function to only be invoked once per test module (the default is to invoke once per test function). Multiple test functions in a test module will thus each receive the same setup fixture instance, thus saving time. Possible values for scope are: **function**, **class**, **module**, **package** or **session**.

softserve

Unit Testing Metrics: coverage.py

Coverage.py is a tool for measuring code coverage of Python programs. It monitors your program, noting which parts of the code have been executed, then analyzes the source to identify code that could have been executed but was not.

Coverage measurement is typically used to gauge the effectiveness of tests. It can show which parts of your product code are being exercised by tests, and which are not.

pip install coverage

Some test runners provide coverage integration to make it easy to use coverage.py while running tests. For example, pytest has the pytest-cov plugin.

```
$ coverage report -m
Name                      Stmts   Miss  Cover   Missing
-------------------------------------------------------
my_program.py                20      4    80%   33-35, 39
my_other_module.py           56      6    89%   17-23
-------------------------------------------------------
TOTAL                        76     10    87%
```

softserve

Links

https://docs.python.org/3/library/unittest.html#class-and-module-fixtures

https://www.drdobbs.com/testing/unit-testing-with-python/240165163?pgno=2

https://docs.python-guide.org/writing/tests/

https://en.wikipedia.org/wiki/Test-driven_development

https://devpractice.ru/unit-testing-in-python-part-1/

https://docs.pytest.org/en/stable/contents.html

http://zetcode.com/python/pytest/

https://realpython.com/python-mock-library/

https://coverage.readthedocs.io/en/latest/

softserve


TO THE FUTURE
softserve