TDD. Unit Testing with Pyton

About me

Trayan Iliev

– CEO of **IPT – Intellectual Products & Technologies**
http://www.iproduct.org

– Oracle® certified programmer 15+ Y

– end-to-end reactive fullstack apps with Java, ES6+, TypeScript, Angular, React and Vue.js

– 12+ years IT trainer: Spring, Java EE, Node.js, Express, GraphQL, SOA, REST, DDD & Reactive Microservices

– Voxxed Days, jPrime, Java2Days, jProfessionals, BGOUG, BGJUG, DEV.BG speaker

– Organizer RoboLearn hackathons and IoT enthusiast

Where to Find The Code and Materials?

https://github.com/iproduct/intro-python

Test Driven Development (TDD)

Тестване на софтуера

Софтуерното тестване е процес на изследване на софтуера, с цел получаване на информация за качеството на продукта или услугата, която се изпитва. Софтуерното тестване може да осигури обективен, независим поглед, който да даде възможност на клиента да разбере рисковете при реализацията на софтуера. Техниките за тестване включват (но не са ограничени до) изпълнение на програмата с намерение да се открият софтуерни бъгове (грешки или други дефекти). Процесът на софтуерно тестване е неразделна част от софтуерното инженерство и осигуряване на качеството на софтуера.

[Wikipedia]

Източник : Wikipedia - http://en.wikipedia.org/wiki/Software_testing

Видове тестване

- Статично и динамично тестване
- White-Box testing – тества вътрешната структура и работа на софтуера (API testing, Code coverage, Fault injection – Stress testing, Mutation testing)
- Black-box testing – тества функционалността без да се интересува от вътрешната реализация

- Grey-box testing – използва познания за структурите от данни и алгоритмите при разработката на тестове
- Визуално тестване – записват се всички действия на тестващия с цел лесно да се възпроизведе проблема

Нива на тестване

- Unit testing – компонентно тестване, при което се тества функционалността на специфична секция от кода (обикновено метод – като минимум конструкторите)
- Integration testing – проверява дали интерфейсите между компонентите са реализирани според спецификацията им
- System testing – тества се напълно интегрираната система за да се определи дали реализацията съответства на изискванията
- Acceptance testing – тестване на системата от крайните й потребители

Специфични цели при тестване

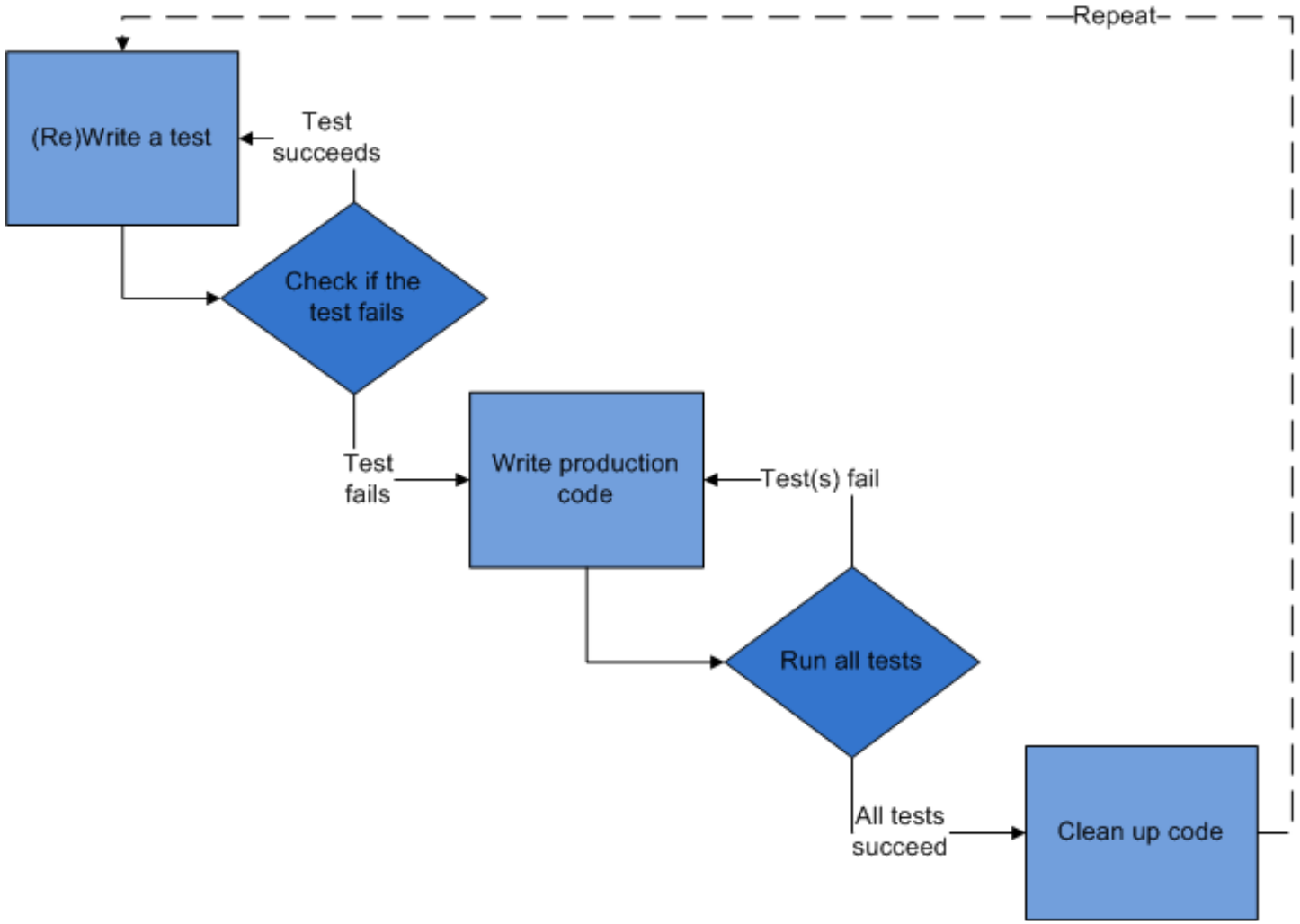
- Installation testing
- Compatibility testing
- Smoke and sanity testing
- Regression testing
- Acceptance testing
- Alpha testing
- Beta testing
- Functional vs non-functional testing
- Destructive testing
- Software performance testing
- Usability testing
- Accessibility
- Security testing
- Internationalization and localization
- Development testing

Източник : Wikipedia - http://en.wikipedia.org/wiki/Software_testing

Test Driven Development (TDD) с JUnit 4

- **Test-Driven Development (TDD)** е техника, при която разработката на софтуер се насочва чрез писане на тестове.
- Първоначално е развита от Kent Beck (в края на 90-те).
- Основната идея е да се повтарят последователно следните пет стъпки:
  1. Пишем автоматичен тест (Unit test) за следващата **малка** част нова функционалност като си представяме че кодът вече съществува;
  2. Пишем празни методи (Stubs), така че кодът да се компилира;
  3. Пускаме теста – той трябва да пропадне, иначе тестът не е добър;
  4. Пишем **минималното** количество функционален код така, че тестът да успее – ако тестът не минава успешно, значи кодът не е добър;
  5. Променяме (Refactor) както стария, така и новия код, за да го структурираме по-добре.

Последователни етапи при TDD



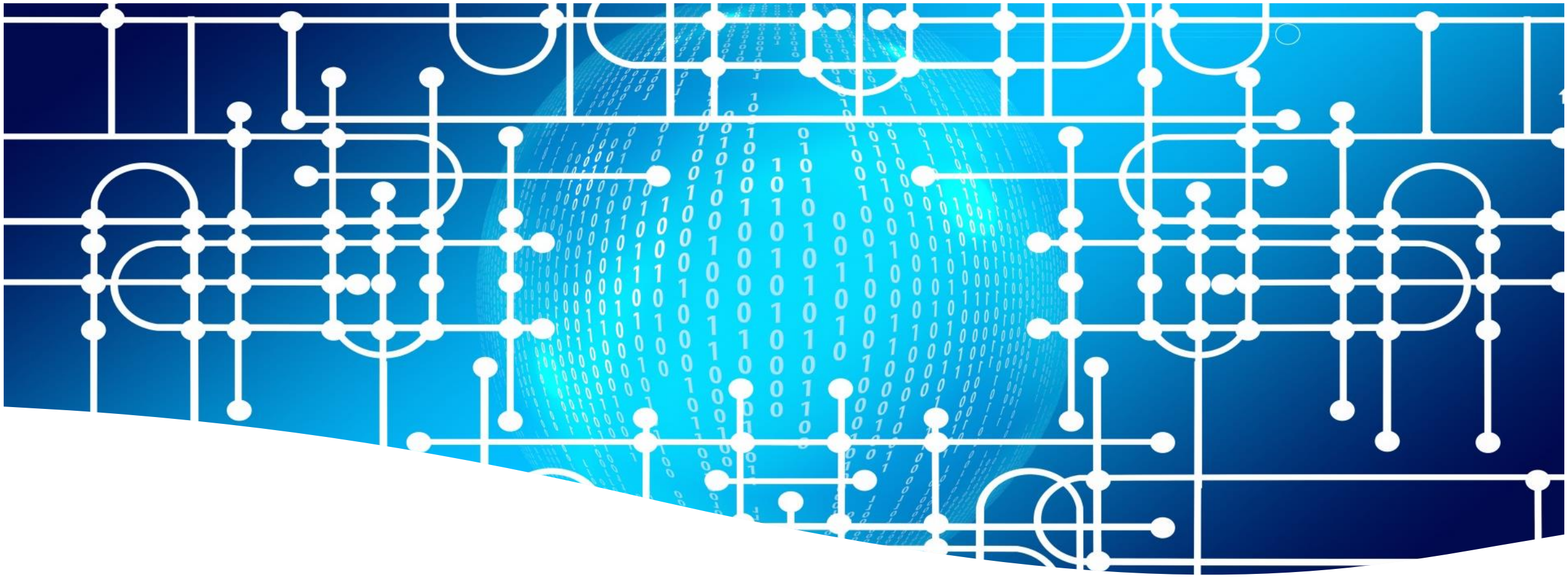
Източник: Wikipedia (http://en.wikipedia.org/wiki/File:Test-driven_development.PNG), Автор: Excirial, Лиценз: Creative Commons Attribution, ShareAlike 3.0 License / GNU Free Documentation License, Version 1.2

Agile Testing - TDD

Ето един добър начин за разработка на нова функционалност:

- Вижте какво имате да правите.
- Напишете UnitTest за желаната функционалност, като изберете най-малкия възможен инкремент, който ви хрумва.
- Стартирайте UnitTest-а. Ако е успешен сте готови; отидете на стъпка 1 или ако сте напълно готови си идете у дома.
- Решете текущия проблем: може би все още не сте написали новия метод. Може би методът не работи точно както трябва. Направете необходимите поправки. Отидете на стъпка 3.

Източник : http://c2.com/cgi/wiki?CodeUnitTestFirst



Unit testing frameworks: unittest

Main Concepts

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

Example Unit Test

```
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()
```

```
python -m unittest -v tests.test_str

test_isupper
(tests.test_str.TestStringMethods) ... ok

test_split (tests.test_str.TestStringMethods)
... ok

test_upper (tests.test_str.TestStringMethods)
... ok

----------------------------------------------------------------

Ran 3 tests in 0.002s
```

Command-Line Interface

- You can pass in a list with any combination of module names, and fully qualified class or method names:

```
python -m unittest test_module1 test_module2
python -m unittest test_module.TestClass
python -m unittest test_module.TestClass.test_method
```

- Test modules can be specified by file path as well:

```
python -m unittest tests/test_something.py
```

- When executed without arguments Test Discovery is started:

```
python -m unittest [discover] [-v]
python -m unittest discover -s project_directory -p "*_test.py“
python -m unittest discover -s root/namespace -t root
```

TestCase Common Assertions

Method	Checks that	New in
assertEqual(a, b)	a == b	
assertNotEqual(a, b)	a != b	
assertTrue(x)	bool(x) is True	
assertFalse(x)	bool(x) is False	
assertIs(a, b)	a is b	3.1
assertIsNot(a, b)	a is not b	3.1
assertIsNone(x)	x is None	3.1
assertIsNotNone(x)	x is not None	3.1
assertIn(a, b)	a in b	3.1
assertNotIn(a, b)	a not in b	3.1
assertIsInstance(a, b)	isinstance(a, b)	3.2
assertNotIsInstance(a, b)	not isinstance(a, b)	3.2

Checking for exceptions, warnings, log messages

Method	Checks that	New in
assertRaises(exc, fun, *args, **kwds)	fun(*args, **kwds) raises *exc*	
assertRaisesRegex(exc, r, fun, *args, **kwds)	fun(*args, **kwds) raises *exc* and the message matches regex *r*	3.1
assertWarns(warn, fun, *args, **kwds)	fun(*args, **kwds) raises *warn*	3.2
assertWarnsRegex(warn, r, fun, *args, **kwds)	fun(*args, **kwds) raises *warn* and the message matches regex *r*	3.2
assertLogs(logger, level)	The with block logs on *logger* with minimum *level*	3.4
assertNoLogs(logger, level)	The with block does not log on*logger* with minimum *level*	3.10

Problem 1: Write Unit test for the Long Number problem

1. Got to the Moodle of the course and find the "Exercise 1: Long Number" problem (from 21 February - 27 February section) using TDD
2. Find the solution of the problem –e.g. https://github.com/iproduct/intro-python/blob/master/08-exercises/long_nuber.py
3. Write a unit test for the find_solution function with the test cases and data given in Moodle of the course as examples.
4. Refactor the solution to be easier to test, by extracting the solving algorithm in a separate function called find_solution that receives test case data, and returns as tuple the substitution interval and the final maximal result number after substitution.
5. Run the test with coverage information.

Organizing test code

```
import unittest

class WidgetTestCase(unittest.TestCase):
    def setUp(self):
        self.widget = Widget('The widget')

    def tearDown(self):
        self.widget.dispose()

    def test_default_widget_size(self):
        self.assertEqual(self.widget.size(), (50,50),
                         'incorrect default size')

    def test_widget_resize(self):
        self.widget.resize(100,150)
        self.assertEqual(self.widget.size(), (100,150),
                         'wrong size after resize')
```

Testing Async IO

```
import unittest
from unittest import IsolatedAsyncioTestCase
import aiohttp

events = []

class Test(IsolatedAsyncioTestCase):
    def setUp(self):
        events.append("setUp")

    async def asyncSetUp(self):
        self._client_session = aiohttp.ClientSession()
        events.append("asyncSetUp")

    async def test_response(self):
        events.append("test_response")
        async def get_request():
            async with self._client_session.get("http://python.org") as response:
                self.assertEqual(response.status, 200)
        self.addAsyncCleanup(self.on_cleanup)

    def tearDown(self):
        events.append("tearDown")

    async def asyncTearDown(self):
        await self._client_session.close()
        events.append("asyncTearDown")

    async def on_cleanup(self):
        events.append("cleanup")
        print(events)

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

Distinguishing test iterations using subTest

```
import unittest

class NumbersTest(unittest.TestCase):

    def test_even(self):
        """
        Test that numbers between 0 and 5 are all even.
        """
        for i in range(0, 6):
            with self.subTest(i=i):
                self.assertEqual(i % 2, 0)
```

```
======================================================================
FAIL: test_even (__main__.NumbersTest) (i=1)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "subtests.py", line 32, in test_even
    self.assertEqual(i % 2, 0)
AssertionError: 1 != 0

======================================================================
FAIL: test_even (__main__.NumbersTest) (i=3)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "subtests.py", line 32, in test_even
    self.assertEqual(i % 2, 0)
AssertionError: 1 != 0

======================================================================
FAIL: test_even (__main__.NumbersTest) (i=5)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "subtests.py", line 32, in test_even
    self.assertEqual(i % 2, 0)
AssertionError: 1 != 0
```

Skipping tests

```
class MyTestCase(unittest.TestCase):
    @unittest.skip("demonstrating skipping")
    def test_nothing(self):
        self.fail("shouldn't happen")

    @unittest.skipIf(aiohttp.__version__ < "3.9",
                     "not supported in this library version")
    def test_format(self):
        # Tests that work for only a certain version of the library.
        pass

    @unittest.skipUnless(sys.platform.startswith("win"), "requires Windows")
    def test_windows_support(self):
        # windows specific testing code
        pass

    def test_maybe_skipped(self):
        if not external_resource_available(): self.skipTest("external resource not available")
        # test code that depends on the external resource
        pass
```

```
Ran 4 tests in 0.002s

OK (skipped=3)

Skipped: not supported in this library version

Skipped: external resource not available

Skipped: demonstrating skipping
```

Expected failures

```
class ExpectedFailureTestCase(unittest.TestCase):
    @unittest.expectedFailure
    def test_fail(self):
        self.assertEqual(1, 0, "broken")
```

```
Expected failure: Traceback (most recent call last):
  File "C:\Python310\lib\unittest\case.py", line 59, in testPartExecutor
    yield
  File "C:\Python310\lib\unittest\case.py", line 591, in run
    self._callTestMethod(testMethod)
  File "C:\Python310\lib\unittest\case.py", line 549, in _callTestMethod
    method()
  File "D:\CoursePython\git\intro-python\12-testing\tests\test_skipping_failures.py", line 35, in test_fail
    self.assertEqual(1, 0, "broken")
  File "C:\Program Files\JetBrains\PyCharm
2020.3\plugins\python\helpers\pycharm\teamcity\diff_tools.py", line 32, in _patched_equals
    old(self, first, second, msg)
  File "C:\Python310\lib\unittest\case.py", line 845, in assertEqual
    assertion_func(first, second, msg=msg)
  File "C:\Python310\lib\unittest\case.py", line 838, in _baseAssertEqual
    raise self.failureException(msg)
AssertionError: 1 != 0 : broken
```

Problem 2: Write Unit test for the Generic JsonRepository class

1. Write a unit test for all entity CRUD methods (create, update, delete_by_id, find_all, find_by_id, save, and load) of the generic JsonRepository class: https://github.com/iproduct/intro-python/blob/master/02-classes-library/dao/json_repository.py
2. Write a unit test for file IO methods (save, load) of the generic JsonRepository class
3. Run the tests with coverage information.

Main Concepts

- unittest.mock is a library for testing in Python. It allows you to replace parts of your system under test with mock objects and make assertions about how they have been used.
- unittest.mock provides a core Mock class removing the need to create a host of stubs throughout your test suite. After performing an action, you can make assertions about which methods / attributes were used and arguments they were called with. You can also specify return values and set needed attributes in the normal way.
- Additionally, mock provides a patch() decorator that handles patching module and class level attributes within the scope of a test, along with sentinel for creating unique objects.
- Mock is designed for use with unittest and is based on 'action -> assertion' pattern instead of 'record -> replay' used by many mocking frameworks.

unittest.mock — mock object library

```
>>> from unittest.mock import MagicMock
>>> thing = ProductionClass()
>>> thing.method = MagicMock(return_value=3)
>>> thing.method(3, 4, 5, key='value')
3
>>> thing.method.assert_called_with(3, 4, 5, key='value')
```

```
Ran 4 tests in 0.002s

OK (skipped=3)

Skipped: not supported in this library version

Skipped: external resource not available

Skipped: demonstrating skipping
```

- side_effect allows you to perform side effects, including raising an exception when a mock is called:

```
>>> mock = Mock(side_effect=KeyError('foo'))
>>> mock()
Traceback (most recent call last):
 ...
KeyError: 'foo'
>>>
values = {'a': 1, 'b': 2, 'c': 3}
def side_effect(arg):
```

unittest.mock — mock object library - II

```
>>> values = {'a': 1, 'b': 2, 'c': 3}
>>> def side_effect(arg):
...     return values[arg]

>>> mock.side_effect = side_effect
>>> mock('a'), mock('b'), mock('c')
(1, 2, 3)
>>> mock.side_effect = [5, 4, 3, 2, 1]
>>> mock(), mock(), mock()
(5, 4, 3)
```

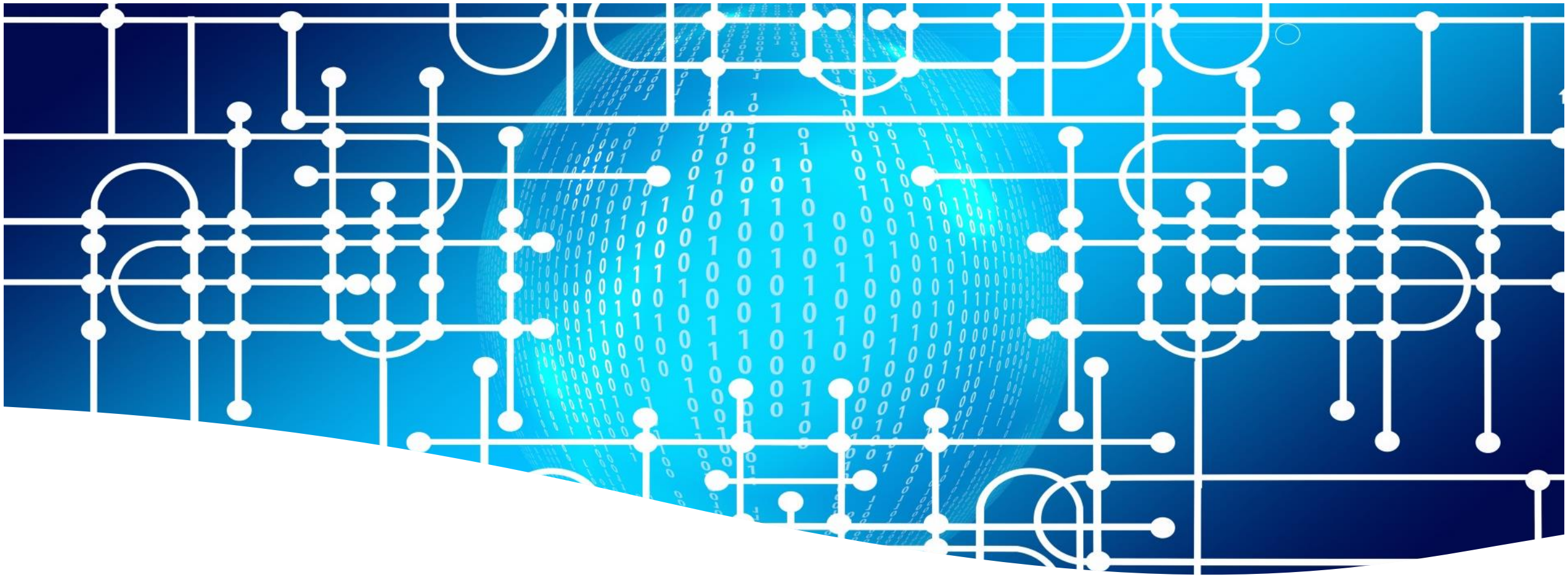
unittest.mock — using patch()

```
>>> from unittest.mock import patch
>>> @patch('module.ClassName2')
... @patch('module.ClassName1')
... def test(MockClass1, MockClass2):
...     module.ClassName1()
...     module.ClassName2()
...     assert MockClass1 is module.ClassName1
...     assert MockClass2 is module.ClassName2
...     assert MockClass1.called
...     assert MockClass2.called
...
>>> test()
```

unittest.mock — using patch()

```
>>> with patch.object(ProductionClass, 'method', return_value=None) as mock_method:
...     thing = ProductionClass()
...     thing.method(1, 2, 3)

mock_method.assert_called_once_with(1, 2, 3)
```



Unit testing frameworks: pytest

Main Concepts

- The pytest framework makes it easy to write small, readable tests, and can scale to support complex functional testing for applications and libraries.
- Run the following command in your command line:

```
pip install -U pytest
```

- Check that you installed the correct version:

```
$ pytest --version
```

```
pytest 7.1.1
```

Pytest Simple Example

```python
# content of test_pytest01.py
def inc(x):
    return x + 1

def test_answer():
    assert inc(3) == 5
```

```
(venv) D:\CoursePython\git\intro-python\12-testing>pytest tests/test_pytest01.py
============================================================================
test session starts
============================================================================
platform win32 -- Python 3.10.1, pytest-7.1.1, pluggy-1.0.0
rootdir: D:\CoursePython\git\intro-python\12-testing
collected 1 item

tests\test_pytest01.py F
[100%]
```

Pytest Simple Example

```
================================= FAILURES =================================
_____________________________________________________________________________
test_answer
_____________________________________________________________________________

    def test_answer():
>       assert inc(3) == 5
E       assert 4 == 5
E        +  where 4 = inc(3)

tests\test_pytest01.py:7: AssertionError
=========================================================================
short test summary info
=========================================================================
FAILED tests/test_pytest01.py::test_answer - assert 4 == 5
============================== 1 failed in 0.06s ==============================
```

Assert that a certain exception is raised

```
# content of test_pytest_exception02.py
import pytest

def f():
    raise SystemExit(1)

def test_mytest():
    with pytest.raises(SystemExit):
        f()
```

```
(venv) D:\CoursePython\git\intro-python\12-testing>pytest tests/test_pytest_exception02.py
============================= test session starts ==============================
platform win32 -- Python 3.10.1, pytest-7.1.1, pluggy-1.0.0
rootdir: D:\CoursePython\git\intro-python\12-testing
collected 1 item

tests\test_pytest_exception02.py .
[100%]

============================== 1 passed in 0.03s ===============================
```

Group multiple tests in a class

```python
# content of test_class.py
class TestClass:
    def test_one(self):
        x = "this"
        assert "h" in x

    def test_two(self):
        x = "hello"
        assert hasattr(x, "check")
```

```
=============== test session starts ======================
collecting ... collected 2 items

test_pytest_class.py::TestClass::test_one
test_pytest_class.py::TestClass::test_two PASSED [ 50%]  FAILED [100%]
test_pytest_class.py:6 (TestClass.test_two)
self = <tests.test_pytest_class.TestClass object at 0x0000013E78258670>

    def test_two(self):
        x = "hello"
>       assert hasattr(x, "check")
E       AssertionError: assert False
E        +  where False = hasattr('hello', 'check')

test_pytest_class.py:9: AssertionError
```

Group multiple tests in a class

- Grouping tests in classes can be beneficial for the following reasons:
  - Test organization
  - Sharing fixtures for tests only in that particular class
  - Applying marks at the class level and having them implicitly apply to all tests

Request a unique temporary directory for tests

- ```
  # content of test_tmp_path.py
  def test_needsfiles(tmp_path):
      print(tmp_path)
      assert 0
  ```

```
============================= test session starts =============================
collecting ... collected 1 item

test_pytest_tempdir04.py::test_needsfiles FAILED
[100%]C:\Users\office27\AppData\Local\Temp\pytest-of-office27\pytest-0\test_needsfiles0

test_pytest_tempdir04.py:1 (test_needsfiles)
tmp_path = WindowsPath('C:/Users/office27/AppData/Local/Temp/pytest-of-office27/pytest-0/test_needsfiles0')

    def test_needsfiles(tmp_path):
        print(tmp_path)
>       assert 0
E       assert 0

test_pytest_tempdir04.py:4: AssertionError
```
```

Problem: Write a web client test using Requests and BS4 for http://python.org

Write unit test for the http://python.org web page (using Resquests and Beautiful Soup 4 (BS4) libraries) with following test cases:

1. Using GET method of HTTP protocol, fetch the http://python.org web page document and test that the status code is 200 OK and the content type is text/html.
2. Test that all the menu items in the <nav> section are visualized correctly.
3. Test that the alt attribute text of the image in the <header> tag section of the page is "python™".
4. Test that the "small-widget" class <h2>s in the page contain "Get Started ", "Download" , "Docs", and "Jobs" texts

Problem 3: Refactor test for JsonRepository using pytest

1. Refactor the unit test for JsonRepository using pytest library for all CRUD and JSON File IO methods (create, update, delete_by_id, find_all, find_by_id, save, and load, save, load), using temporary directory from previuos slide for the json files created during the test: https://github.com/iproduct/intro-python/blob/master/02-classes-library/dao/json_repository.py
2. Write a unit tests for the additional find_by_title and find_by_author methods of BookRepository class.
3. Run the tests with coverage information.

Request a unique temporary directory for tests

```python
# content of test_class.py
class TestClass:
    def test_one(self):
        x = "this"
        assert "h" in x

    def test_two(self):
        x = "hello"
        assert hasattr(x, "check")
```

```
============== test session starts ======================
collecting ... collected 2 items

test_pytest_class.py::TestClass::test_one
test_pytest_class.py::TestClass::test_two PASSED [ 50%]  FAILED [100%]
test_pytest_class.py:6 (TestClass.test_two)
self = <tests.test_pytest_class.TestClass object at 0x0000013E78258670>

    def test_two(self):
        x = "hello"
>       assert hasattr(x, "check")
E       AssertionError: assert False
E        +  where False = hasattr('hello', 'check')

test_pytest_class.py:9: AssertionError
```

pytest-cov

- Installation with pip:

pip install pytest-cov

- Usage:

pytest --cov=myproj tests/
Would produce a report like:

```
---------- coverage: ... ----------
Name             Stmts   Miss  Cover
----------------------------------
myproj/__init__      2      0   100%
myproj/myproj      257     13    94%
myproj/feature4286  94      7    92%
----------------------------------
TOTAL              353     20    94%
```

Behavior Driven Development (BDD)

BDD Main Concepts

- Behavior-driven development is an extension of test-driven development, a development process that makes use of a simple DSL.
- These DSLs convert structured natural language statements into executable tests.
- The result is a closer relationship to acceptance criteria for a given function and the tests used to validate that functionality. As such it is a natural extension of TDD testing in general.
- Example:

```
Feature: showing off behave
  Scenario: run a simple test
     Given we have behave installed
      When we implement a test
      Then behave will test it for us!
```

BDD Focus

BDD focuses on:

- Where to start in the process
- What to test and what not to test
- How much to test in one go
- What to call the tests
- How to understand why a test fails

BDD Focus

- BDD focuses on obtaining a clear understanding of desired software behavior through discussion with stakeholders.
- It extends TDD by writing test cases in a natural language that non-programmers can read.
- Behavior-driven developers use their native language in combination with the ubiquitous language of domain-driven design to describe the purpose and benefit of their code.
- This allows the developers to focus on why the code should be created, rather than the technical details, and minimizes translation between the technical language in which the code is written and the domain language spoken by the business, users, stakeholders, project management, etc.

BDD Main Goals

- At its heart, BDD is about rethinking the approach to unit testing and acceptance testing in order to avoid issues that naturally arise.
- For example, BDD suggests that unit test names be whole sentences starting with a conditional verb ("should" in English for example) and should be written in order of business value.
- Acceptance tests should be written using the standard agile framework of a user story: "**Being a** [role/actor/stakeholder] **I want a** [feature/capability] **yielding a** [benefit]".
- Acceptance criteria should be written in terms of scenarios and implemented in classes: **Given** [initial context], **when** [event occurs], **then** [ensure some outcomes] .

Outside–in Process

- Establishing the goals of different stakeholders required for a vision to be implemented
- Drawing out features which will achieve those goals using feature injection
- Involving stakeholders in the implementation process through outside–in software development
- Using examples to describe the behavior of the application, or of units of code
- Automating those examples to provide quick feedback and regression testing
- Using 'should' when describing the behavior of software to help clarify responsibility and allow the software's functionality to be questioned
- Using 'ensure' when describing responsibilities of software to differentiate outcomes in the scope of the code in question from side-effects of other elements of code.
- Using mocks to stand-in for collaborating modules of code which have not yet been written

Python behave

- Installation with pip:

pip install -U behave

- Usage:

```
Feature: showing off behave

  Scenario: run a simple test
     Given we have behave installed
      When we implement a test
      Then behave will test it for us!
```

features/steps/tutorial.py

```
pip from behave import *

@given('we have behave installed')
def step_impl(context):
    pass

@when('we implement a test')
def step_impl(context):
    assert True is not False

@then('behave will test it for us!')
def step_impl(context):
    assert context.failed is False
```

Python behave

```
(venv) D:\CoursePython\git\intro-python\12-testing>behave
tests/features/tutorial.feature
Feature: showing off behave # tests/features/tutorial.feature:1

  Scenario: run a simple test        # tests/features/tutorial.feature:3
    Given we have behave installed   # tests/features/steps/tutorial.py:3
    When we implement a test         # tests/features/steps/tutorial.py:7
    Then behave will test it for us! # tests/features/steps/tutorial.py:11

1 feature passed, 0 failed, 0 skipped
1 scenario passed, 0 failed, 0 skipped
3 steps passed, 0 failed, 0 skipped, 0 undefined
Took 0m0.000s
```

Python behave - example 2

Scenario: Search for an account

  Given I search for a valid account "42"

  Then I will see the account details for "42"

```
pip from behave import *

@given('I search for a valid account "{account_id}"')
def step_impl(context, account_id):
    result = context.client.get(f'http://localhost:8000/accounts/{account_id}')
    context.response(result)

@then('I will see the account details for "{account_id}"')
def step_impl(context, account_id):
    assert context.response.status_code == 200
    assert getattr(context.response.json(), "id") == account_id
```

Thank's for Your Attention!

Trayan Iliev

IPT – Intellectual Products & Technologies

http://iproduct.org/

https://github.com/iproduct

https://twitter.com/trayaniliev

https://www.facebook.com/IPT.EACAD