FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG
PHILOSOPHISCHE FAKULTÄT
UND FACHBEREICH THEOLOGIE

Unittest in Python

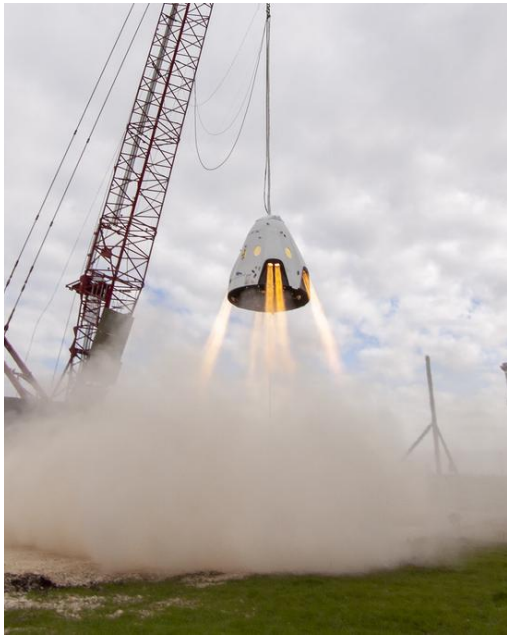
Unittest and Continuous Integration

https://github.com/martialblog/python-unittest-presentation

What is a Unit-Test?

Why bother testing?

- Does your code work as expected
- Testing helps you improve your design
- Testing takes fear
- Software Engineering vs. Regular Engineering
    - Test stuff or it explodes!



Microsoft Research Study 2008[1]:
[...] pre-release defect density of the four products decreased between 40% and 90% relative to similar projects that did not use the TDD practice. [...]

How to do it?

- Manual execution

- Disadvantages
  - Work intensive
  - Not repeatable
  - Could be wrong

```
def sieve_of_eratosthenes(n):
    """
    Gets you all the primes from 1 to n
    """

    A = [True] * n
    A[0] = False
    A[1] = False

    for i, isprime in enumerate(A):

        if isprime:
            yield i

            for x in range(i * i, n, i):
                A[x] = False
```

```
> python3 -i sieve/sieve.py
>>> list( sieve_of_eratosthenes(30) )
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
```

How to do it?

- Automatizing it
- Functions and Expectations

- Disadvantages
  - Work intensive

```python
import sieve

def main():

    primes = list(sieve.sieve_of_eratosthenes(30))
    print(primes)
    print('Should be [2, 3, 5, 7, 11, 13, 17, 19, 23, 29]')


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

```
> python3 -i sieve/main.py
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
Should be [2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
```

How to do it?

- Behold the mighty *assert()*
- Assertion of the state of the program

- Disadvantages
  - Stops code execution

```python
import sieve

def main():

    primes = list(sieve.sieve_of_eratosthenes(30))
    assert(primes == [2, 3, 5, 7, 11, 13, 17, 19, 23])


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

```
> python3 -i sieve/main.py
Traceback (most recent call last):
  File "main.py", line 45, in <module>
    main()
  File "main.py", line 37, in main
    test_assert_ok()
  File "main.py", line 18, in test_assert_ok
    assert(primes == [2, 3, 5, 7, 11, 13, 17, 19, 23])
AssertionError
```

How to do it - right.

- The **Unittest Module**
- Using the *Arrange-Act-Assert* Pattern
- Advantages
  - Automated
  - Reliable
  - Scalable
  - Informative
  - Focused*

Unit-Tests exist in almost all languages (Java, JavaScript, Ruby, PHP, …)

A simple Unittest

```
import unittest
import sieve

class SieveTest(unittest.TestCase):

    def test_sieve_of_eratosthenes(self):

        expect_return_value = [2, 3, 5, 7, 11]
        actual_return_value = list(sieve.sieve_of_eratosthenes(12))
        self.assertEqual(expect_return_value, actual_return_value)

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

```
> python3 -m unittest -v sieve_test.py
test_sieve_of_eratosthenes (sieve_test.SieveTest) ... ok

----------------------------------------------------------------------
Ran 1 test in 0.000s
```

The Testing Jargon

- What's a "**Unit**" anyways?
- The smallest testable parts of a program
  - E.g. function calls

- World's quickest primer into good code design[2]
  - Classes/Functions should be small
  - The single responsibility principle
  - Dependency inversion and decoupling

The Testing Jargon

- Fixtures — A well known and fixed environment in which the tests are running.
- Mocking — Isolating tests by simulating complex dependencies.
- Injections — Inserting fixtures/dependencies where you need them
- Coverage — *Code executed* versus *Code available*. Percentage of how much tests cover.
- Branches — VCS concept. Basically, development independent from your main code.
- Builds — Version of an application. Often used to describe compilation processes.
- Test-Driven Dev. — Development practice in which test are written first, then functionality.
- Regression Test — Testing for unwanted defects created in changes.
- Integration Test — All the program's components are tested as whole.
- Acceptance Test — Evaluate the program's compliance with the requirements.
- Continuous Integration — Development practice that focuses on constant integration of changes.

Python Testing Libraries

Unittest (Standard Library)

- Build-in standard for structured tests, works out of the box
- Where you should start when testing[2]
- Based on Java's xUnit. Sometimes not that *pythonic*

py.test

- Small, scalable, pythonic tests (less boilerplate)
- Extensible with plugins
- Supports unittests
- *Author*'s favorite

nose2

- Build on top of unittest2
- Extensible with plugins
- Supports unittests

Python Testing Libraries - Unittest

- Unittest TestCase wrapper class
- Supports simple test discovery
  - *module/tests* folder with *__init__.py*
- Predefined *assert* helpers (List)
- *setUp* and *tearDown* before and after each test
- Skip decorator (*new in 3.1*)
  - @unittest.skip
  - @unittest.skipIf

```
import unittest

class TestFoo(unittest.TestCase):

  def setUp(self):
     self.Instance = MyCoolClass()

  def tearDown(self):
    del self.Instance

  def test_some_method(self):
    return_value = self.Instance.some_method('Zardoz speaks to you')
    self.assertTrue(return_value)

> python3 -m unittest discover
```

Python Testing Libraries - nose2

- nose2 runs Unittest tests
- Less boilerplate more functionality
- Uses build-in *assert()* function
  - nose.tools provides more
- Provides lots of Plugins
  - Dropping into the debugger
  - XML output
  - Test coverage reporting
  - Selective tests with attributes

```
import nose2

def set_up():
  return MyCoolClass()

def test_some_method():
  Instance = set_up()
  return_value = Instance.some_method('More human than human')
  assert(return_value == True)

> nose2
```

Python Testing Libraries - py.test

- py.test runs Unittest tests
- Less boilerplate more functionality
- Uses build-in *assert()* function
- Really cool fixture injection
  - @pytest.fixture(scope="module")
- Provides lots of Plugins
  - XML output
  - Custom Markers
- Clean filesystem tests with tmpdir
- Simple capturing of the stdout/stderr output

```
import pytest

@pytest.fixture
def instance(request):
  return MyCoolClass()

def test_some_method(instance):
  return_value = instance.some_method('Open the pod bay doors')
  assert(return_value == True)

> py.test
```

Testing! Hooray!

- Knowing what the code does
- Better code quality[6]
- Doing the Refactor dance easily
- Increased productivity[6]
- Better cooperation with other developers
    - Automated Testing and CI

Limits of Unit-Testing

- 100% Coverage does not mean your tests are correct
- You can't write a test you didn't think of
- Testing takes time[6]
- You cannot test every input/path
- Tests don’t tell you about design errors

Continuous Integration

Jenkins[3] **-** https://jenkins.io/

- Open source CI tool written in Java
- Easy setup and integrates into almost everything
- Extensible with plugins

Travis CI - https://travis-ci.com/

- Web-based service for CI
- Free for open source projects
- Ridiculously easy to use

Literature

[1] Nachiappan Nagappan (2008): *Realizing quality improvement through test driven development.* [Link].

[2] Martin,Robert C. (2008): *Clean Code: A Handbook of Agile Software Craftsmanship*.

[3] Smart, John Ferguson (2011): *Jenkins: The Definitive Guide: Continuous Integration for the Masses.*

[4] Beazley, David (2013): *Python Cookbook*. Page: 565-590.

[5] Batchelder, Ned (2014)*: Getting Started Testing.* [Link].

[6] George, B., Williams, L. (2011): *An Initial Investigation of Test Driven Development in Industry*.

Markus Opolka

https://github.com/martialblog

markus.opolka@fau.de