Scientific Software Development

Inga Ulusoy, Scientific Software Center, Interdisciplinary Center for Scientific Computing, Heidelberg University

March 2022

Unit 5: Testing, testing, testing,…

- Why you need testing
- Types of tests and test-driven development
- Unittest
- Pytest

We will continue working on our Python modules.

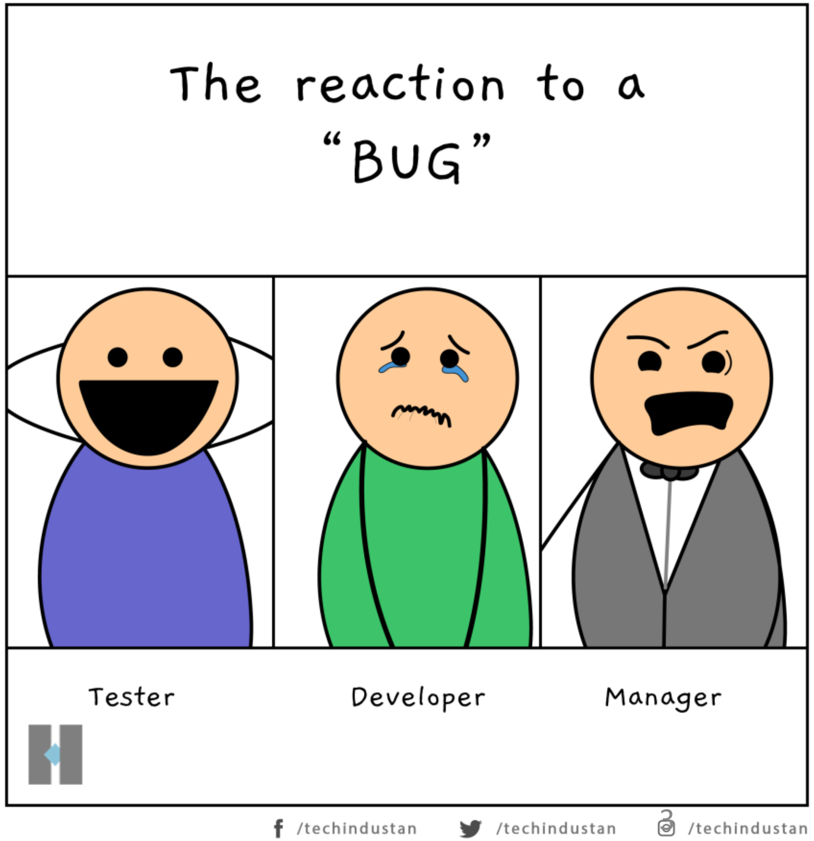
Why testing?

You should always, always test your implementation against a known result!!!!!!!!!!

This ensures

1. that you obtain "real" results.
2. that you find errors in your code that may not always strike.

Meaning that you obtain scientifically sound and reproducible results. As a scientist/scholar, you need to adhere to Scientific Best Practices and are responsible of and accountable for your work!!!



A bad example

SCIENTIFIC PUBLISHING

A Scientist's Nightmare: Software Problem Leads to Five Retractions

Science 314, 2006

Models and the software that implement them define both how science is done and what science is done.

Joppa, McInerny, Harper, Salido *et al.*, "Troubling Trends in Scientific Software Use", *Science* 340, 814 (2013)

Home-made data analysis software had flipped a minus sign leading to false analysis of the data
Result: Retraction of five papers (three were published in *Science)*
The first of those five papers was cited 365 times.

The story is about Geoffry Chang from the Scripps Institute, he is a biologist and reported crystal structures of proteins.
In addition to loss of own reputation, it also cost numerous other researchers a lot of time trying to reproduce and build upon the false results. Others could not get funding or publish papers for topics that contradicted Chang's papers.

Unit 5: Testing, testing, testing,…

- *Why you need testing*
- Types of tests and test-driven development
- Unittest
- Pytest

We will continue working on our Python modules.

Types of tests

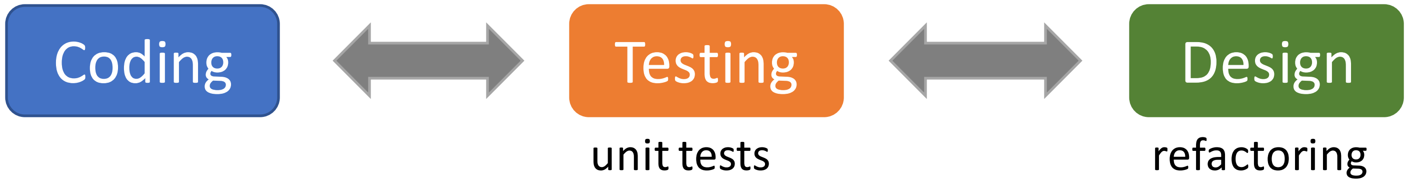
Unit testing
Focus on smallest unit of the program such as a particular function; check that it returns correct value/only accepts "reasonable" input

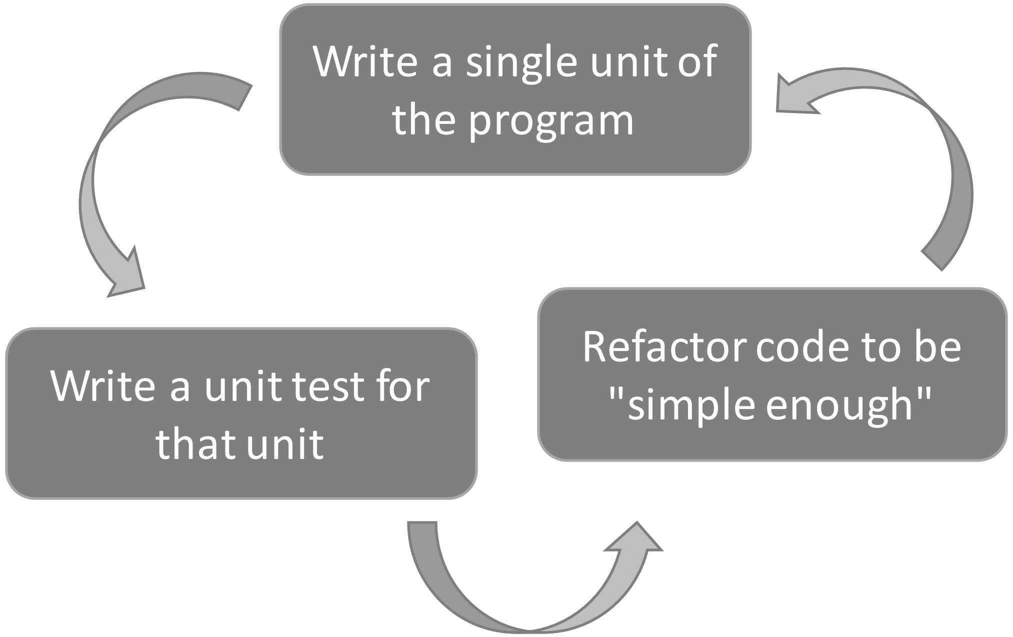
Integration testing
Verifies that unit-tested pieces work together and produce correct output

System testing
Verifies that program runs in different environments/with different compilers/language versions

Performance testing/End-to-end tests/Regression testing ...

Test-driven development





Advantages:

- Improved code design
- Fewer errors in the code
- Initial development takes longer, but saves time in the end phase before release/when adding new features

Pitfalls:

- Too many tests/forget to run all tests/too trivial or too large tests
- Poor maintenance of test suite

Which tests do I need?

- For now, we will use unit tests
- Tomorrow, we will automatize the testing
- There are two main unit test frameworks in python: `unittest` and `pytest`

unittest

contains all the essential functionality

pytest

contains all the essential functionality AND more compact style

Code coverage

- Quantifies how many lines of code/blocks/... are covered by tests – for example, code coverage of 80% means that 20% of the code are not covered by tests
- Good code coverage does not equal good tests!

Unit 5: Testing, testing, testing,...

- *Why you need testing*
- *Types of tests and test-driven development*
- Unittest
- Pytest

We will continue working on our Python modules.

Unittest

- We will start with a simple unit test example.

unittest

- Object-oriented
- TestCase base class
- test fixture: pre- and post-processing of tests
- test suite: collection of tests belonging together
- test runner: test execution and output

`unittest.TestCase`

```
self.assertEqual('foo'.upper(), 'FOO')
self.assertTrue('FOO'.isupper())
self.assertFalse('Foo'.isupper())
s = 'hello world'
self.assertEqual(s.split(), ['hello', 'world'])
```

Run the test:

`python -m unittest`

Unittest

Method	Checks that
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
assertIsNot(a, b)	a is not b
assertIsNone(x)	x is None
assertIsNotNone(x)	x is not None
assertIn(a, b)	a in b
assertNotIn(a, b)	a not in b
assertIsInstance(a, b)	isinstance(a, b)
assertNotIsInstance(a, b)	not isinstance(a, b)

https://docs.python.org/3/library/unittest.html

Unittest

Function to be tested in file `transform.py`:

```
import numpy as np


def area_circ(r_in):
    """Calculates the area of a circle with given radius.

    :Input: The radius of the circle (float, >=0).
    :Returns: The area of the circle (float)."""
    if r_in < 0:
        raise ValueError("The radius must be >= 0.")
    area_out = np.pi * r_in**2
    print("The area of a circle with radius r = {:3.2f}cm \
        is A = {:4.2f}cm2.".format(r_in, area_out))
    return area_out
```

Test class in file `test_transform.py`:

```
import unittest
import numpy as np
import transform as tf


class test_area_circ(unittest.TestCase):
    def test_area_circ(self):
        """Test the area values against a reference for r >= 0."""
        self.assertEqual(tf.area_circ(1), np.pi)
        self.assertEqual(tf.area_circ(0), 0)
        self.assertEqual(tf.area_circ(2.1), np.pi*2.1**2)

    def test_values(self):
        """Make sure value errors are recognized for area_circ."""
        self.assertRaises(ValueError, tf.area_circ, -5)
```

Unittest

- Write an example function and a test class testing the function using different assert methods. Run the unittest test runner by invoking `python –m unittest`.
- Try what happens if your test fails.

Unit 5: Testing, testing, testing,...

- *Why you need testing*
- *Types of tests and test-driven development*
- *Unittest*
- Pytest

We will continue working on our Python modules.

Pytest

- Pytest has all the unittest methods with a shorter syntax (no TestCase derived classes), plus additional modules.

pytest

```
import pytest
import numpy as np
import transform as tf


def test_area_circ():
    """Test the area values against a reference for r >= 0."""
    assert tf.area_circ(1) == np.pi, "should return pi"

    assert tf.area_circ(0) == 0
    assert tf.area_circ(2.1) == np.pi*2.1**2

def test_values():
    """Make sure value errors are recognized for area_circ."""
    with pytest.raises(ValueError):
        tf.area_circ(-5)
```

unittest

```
import unittest
import numpy as np
import transform as tf


class test_area_circ(unittest.TestCase):
    def test_area_circ(self):
        """Test the area values against a reference for r >= 0."""
        self.assertEqual(tf.area_circ(1), np.pi)
        self.assertEqual(tf.area_circ(0), 0)
        self.assertEqual(tf.area_circ(2.1), np.pi*2.1**2)

    def test_values(self):
        """Make sure value errors are recognized for area_circ."""
        self.assertRaises(ValueError, tf.area_circ, -5)
```

Pytest: Structuring your unit tests

- A test can be divided into four sections:

Arrange
prepare the environment for the test

Act
change of the state of system under test (function/method call)

Assert
check changed state and compare to expected behaviour

Cleanup
revert state to "clean slate" so that the next test can run

Pytest: Structuring your unit tests

Arrange

Assert

Act

Cleanup

```
import pytest
import numpy as np
import transform as tf


def test_area_circ():
    """Test the area values against a reference for r >= 0."""
    assert tf.area_circ(1) == np.pi, "should return pi"

    assert tf.area_circ(0) == 0
    assert tf.area_circ(2.1) == np.pi*2.1**2

def test_values():
    """Make sure value errors are recognized for area_circ."""
    with pytest.raises(ValueError):
        tf.area_circ(-5)
```

Pytest: Using markers

- Markers can be used to categorize tests – for example here a marker named circles

```
@pytest.marker.circles
```

Register your markers in pytest.ini (this is enforced to prevent you from accidentally mistyping a marker):

```
# content of pytest.ini
[pytest]
markers =
    circles: mark a test only applying to circles
    your_other_markers: your description
```

Run pytest with only the selected tests:

```
python –m pytest –m circles
```

Pytest: Using markers

- You may also skip tests by using

```
@pytest.marker.skip(reason="My reason to skip this test")
```

Fixtures – an excursion to decorators

- Decorator: A function that extends another function without modifying it

Pytest: Using fixtures

- Fixtures are used to **Arrange** the test
  - not just setup/teardown (explicit names, modular)
  - explicit declarations of dependencies
  - provide a baseline so that each test is reliable and consistent
- Separate dependencies from implementation
- Especially important for integration tests

Pytest: Using fixtures

- Fixtures are invoked as
  `@pytest.fixture()`
- Fixtures can inherit fixtures

```
@pytest.fixture()
def my_parent_fixture():
    …

@pytest.fixture()
def my_child_fixture(my_parent_fixture):
    …
```

scope of the fixture

- function: the default scope, the fixture is destroyed at the end of the test
- class: the fixture is destroyed during teardown of the last test in the class
- module: the fixture is destroyed during teardown of the last test in the module
- package: the fixture is destroyed during teardown of the last test in the package
- session: the fixture is destroyed at the end of the test session

- The scope of a fixture determines the order in which it is executed and how often it is executed:

```
@pytest.fixture(scope='module')
```

- higher-scoped fixture will be executed first, fixtures of same order will be executed based on dependencies
use `autouse=True` if all tests will use that fixture

Pytest: Using fixtures

- You can pass data from a test into a fixture using markers and request

```
@pytest.fixture
def myfixture(request):
    marker = request.node.get_closest_marker("mymark")

@pytest.mark.mymark(myval)
def mytest(myfixture):
    …
```

(replace the example names given in italics)

- You can have your fixture pass a generating function:

```
@pytest.fixture
def myfixture():
    def _my_func(input):
        return 2 + input
    return _my_func

def mytest(myfixture):
    value = myfixture(40)
```

Pytest: Using parameterization

```
@pytest.mark.circles
@pytest.mark.parametrize('myval, result',
                         [
                          (1, np.pi),
                          (0, 0),
                          (2.1, np.pi*2.1**2)
                         ])
def test_area_circ(myval, result):
    """Test the area values against a reference for r >= 0."""
    assert tf.area_circ(myval) == result
```

Pytest: Using parametrizing fixtures

```
@pytest.fixture(params=[1,2], ids=["one", "two"])
def myfixture(request):
    return request.param

def test_myfixture(myfixture):
    print(myfixture)
    pass
```

You can also define the params list elsewhere (ie., top of the module) and pass it to the fixture as a variable.

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

Pytest: Useful plugins

- *pytest-randomly:* enforces your tests to run in a random order (uncover stateful dependencies)
- *pytest-cov*: coverage report of your tests
- *pytest-sugar*: nicer appearance and shows failed tests instantaneously

Unit 5: Testing, testing, testing,…

- *Why you need testing*
- *Types of tests and test-driven development*
- *Unittest*
- *Pytest*

We will continue working on our Python modules.

Live lesson

- In the live lesson, we will write Pytest unit tests for the module that you and your team developed so far.

Live lesson - Demonstrations

- The following demonstrations will take place in the beginning of the live session:
    - How to use parametrization and fixtures in Pytest