Table of contents (../toc.ipynb)

Pytest

Software rots with time and to keep the software in a healthy state tests are vital.

Pytest is a testing library for Python code and software testing (https://en.wikipedia.org/wiki/Software_testing) is very important to ensure:

- quality of software,
- maintainability,
- flexibility.

Imagine a large piece of software which is not tested and there is just one super guru developer who knows how it works and how to add a feature. This is a nightmare for everyone, the company, the dev team, the managers. It will be very hard to change or extend the software.

With tests, you are much more confident to change and improve the implementation.

Alternatives

Pytest is just one option of test frameworks for Python code. You can also look for

- doctest,
- unittest,
- nose.

Using pytest is just my preference, the important thing is that you care about software testing!

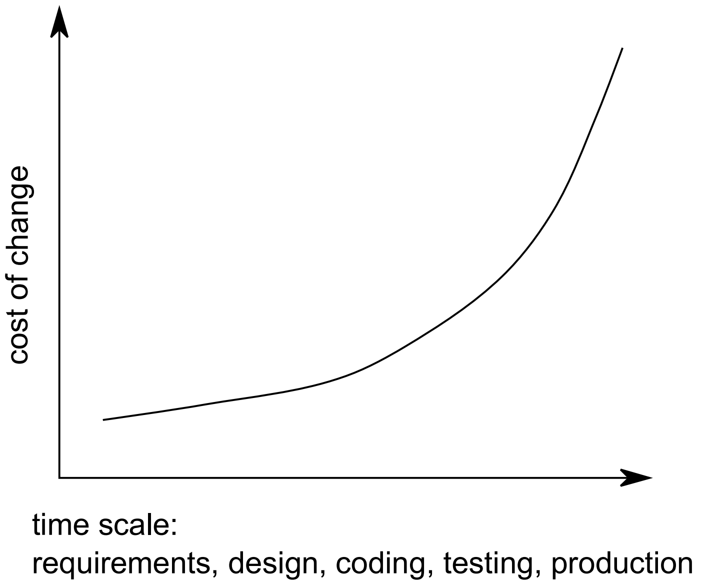
A bit of background

As said, an untested software becomes rigid and is actually not soft anymore. If you have no tests, developers will be afraid to apply changes to the code and hence the software quality decreases over time.

If you skip testing and prefer fast implementation you are accepting something which is called **technical debt (https://en.wikipedia.org/wiki/Technical_debt)**.

It is known from many examples in practice that high technical debt during implementation will cause higher costs later than taking care about software quality from the first day.

The function cost of change over time is roughly exponential as shown in next figure. Hence it makes sense to detect and resolve errors as early as possible.



A test is not a proof

There is one common misunderstanding in testing software. With a test, you can **only say that you could not find an error**, but **you can not say that the software is correct** (free of errors).

There might still be errors which another test could discover.

The only concept which proof correctness is formal verification (https://en.wikipedia.org/wiki/Formal_verification), which is a mathematical procedure to check is software fulfills specification.

But testing is better than no test

Despite the inability of tests to proof correctness, testing is state of the art in software development.

There are different layers of tests, beginning from

- unittesting,
- to integration testing,
- to system testing.

There is much more theory to discover in testing software but we will move now to the practical part and learn how to use pytest.

Processes for software tests

Looking from the process view on software testing there are different concepts at which stage developers write software tests.

- Never. This concept should be avoided of course but you will be surprised how often teams develop software without tests in practice.
- Test driven development (TDD) (https://en.wikipedia.org/wiki/Test-driven_development). This is the extreme counter concept to *never* where fast repetition cycles of: add test; run tests to see that new test fails; write code; run tests; refactor; repeat are conducted.
- V-Model (https://en.wikipedia.org/wiki/V-Model). This classical procedure form product engineering can be summarized as: code, write tests, verify and validate the implementation. It is in between *never* and *TDD*.

Apart from the method *never*, TDD and V-Model can be applied. However, the V-Model takes care about test at a later stage when usually time to market is close and removal of errors high. Hence, continuous testing and a now or never mindset should be fostered in V-Model from my point of view.

Running the first pytest

Pytest is mainly a command line tool and collects and executes all files and function with `test_` prefix. Hence, we need a test file before we can use pytest.

The file `test_simple_test` contains:

```
def test_a_simple_test():
    assert(1 == 1)
```

Once you have installed pytest as package in your python environment you can run pytest in terminal with `pytest`.

Here in jupyter we will use the `!` as prefix to tell jupyter to run commands in command line.

So let us try and run pytest for the specific file `test_simple_test.py`.

In [1]:

```
! pytest - -verbose test_simple_test.py
```

```
============================= test session starts ==============================
==
platform linux -- Python 3.7.6, pytest-5.2.4, py-1.8.1, pluggy-0.13.1 -- /home
/rhs2rng/PycharmProjects/py-algorithms-4-automotive-engineering/_venv/bin/pyth
on
cachedir: .pytest_cache
rootdir: /home/rhs2rng/PycharmProjects/py-algorithms-4-automotive-engineering,
inifile: pytest.ini
plugins: cov-2.8.1
collected 1 item

test_simple_test.py::test_a_simple_test PASSED                           [10
0%]

============================== 1 passed in 0.01s ==============================
==
```

The output shows that one file was collected and that the test passed.

However, this was quite clear because the test will always pass due to `assert(1 == 1)`.

Testing a function

We will test now the function

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

which is contained in the file `my_source.py`

The respective test is

```python
def test_add_one():
    assert add_func(2, 5) == 7
```

in `test_my_source.py` file.

In [2]:

```
! pytest - -verbose test_my_source.py: : test_add_one
```

```
============================= test session starts ==============================
==
platform linux -- Python 3.7.6, pytest-5.2.4, py-1.8.1, pluggy-0.13.1 -- /home
/rhs2rng/PycharmProjects/py-algorithms-4-automotive-engineering/_venv/bin/pyth
on
cachedir: .pytest_cache
rootdir: /home/rhs2rng/PycharmProjects/py-algorithms-4-automotive-engineering,
inifile: pytest.ini
plugins: cov-2.8.1
collected 1 item

test_my_source.py::test_add_one PASSED                                     [10
0%]

============================== 1 passed in 0.01s ===============================
==
```

The syntax `pytest --verbose test_my_source.py::test_add_one` told pytest to run one specific test function in one specific file.

Now let us have a look at a failing test. The test function

```python
def test_add_fail():
    assert add_func(5, 5) == 9
```

in `test_my_source.py` file should fail.

In [3]:

```
! pytest - -verbose test_my_source.py
```

```
============================ test session starts ============================
==
platform linux -- Python 3.7.6, pytest-5.2.4, py-1.8.1, pluggy-0.13.1 -- /home
/rhs2rng/PycharmProjects/py-algorithms-4-automotive-engineering/_venv/bin/pyth
on
cachedir: .pytest_cache
rootdir: /home/rhs2rng/PycharmProjects/py-algorithms-4-automotive-engineering,
inifile: pytest.ini
plugins: cov-2.8.1
collected 2 items

test_my_source.py::test_add_one PASSED                                  [ 5
0%]
test_my_source.py::test_add_fail FAILED                                 [10
0%]

================================== FAILURES =================================
==
________________________________ test_add_fail ______________________________
__

    def test_add_fail():
>       assert add_func(5, 5) == 9
E       assert 10 == 9
E         -10
E         +9

test_my_source.py:9: AssertionError
========================= 1 failed, 1 passed in 0.02s =======================
==
```

We called now all tests of `test_my_source.py` at once with `pytest --verbose test_my_source.py`. The first test still passes, the second test fails as expected.

You can also group tests within classes and the syntax to call a specific test in a class of a file becomes

```
pytest test_mod.py::TestClass::test_method.
```

Types of assertions

Next to assertions of integers there are assertions for floats, types, warnings, strings, error codes, exceptions, and so forth. Please find in pytest documentation many basic examples https://docs.pytest.org/en/latest/assert.html#assert (https://docs.pytest.org/en/latest/assert.html#assert).

If you want to compare floats, please remember to use relative or absolute tolerance instead of `assert(3.14 == 3.14)`. You can user either the pytest `approx` function https://docs.pytest.org/en/latest/reference.html#pytest-approx (https://docs.pytest.org/en/latest/reference.html#pytest-approx) or numpy `np.allclose` (see numpy notebook of this course).

We will take a look at some float comparisons and skip the other types of assertions.

A plain float comparison of two numpy arrays will cause a failure.

```
In [1]: import numpy as np

        np.array([0.1, 0.2]) + np.array([0.2, 0.4]) == np.array([0.3, 0.6])
```

```
Out[1]: array([False, False])
```

The correct way to compare floats is to use an approximate comparison. The default precision is $1e-6$.

In [5]:

```
from pytest import approx

np.array([0.1, 0.2]) + np.array([0.2, 0.4]) == approx(np.array([0.3, 0.6]))
```

Out[5]:

```
True
```

But the precision can also be passed to the comparer.

In [6]:

```
np.array([0.1, 0.2]) + np.array([0.2, 0.4]
                                ) == approx(np.array([0.3, 0.6]), abs=1e-3)
```

Out[6]:

```
True
```

Other pytest features

There are much more features in pytest which help to write good tests. You can for instance

- parametrize tests (https://docs.pytest.org/en/latest/example/parametrize.html) to test code with a set of stimuli data,
- work with markers and fixtures (https://docs.pytest.org/en/latest/example/markers.html) to select tests through their names or through labels, or to skip test which take long time to conduct,
- check code coverage with `pytest-cov` (https://pytest-cov.readthedocs.io/en/latest/) to see if your tests covered all lines of the code under test.

Next, we will briefly try `pytest-cov`.

In [7]:

```
! pytest - -cov-report term-missing - -cov
```

```
============================= test session starts ==============================
==

platform linux -- Python 3.7.6, pytest-5.2.4, py-1.8.1, pluggy-0.13.1

rootdir: /home/rhs2rng/PycharmProjects/py-algorithms-4-automotive-engineering,
inifile: pytest.ini

plugins: cov-2.8.1

collected 3 items



test_my_source.py .F                                                        [ 6
6%]

test_simple_test.py .                                                       [10
0%]



=================================== FAILURES ===================================
==

________________________________ test_add_fail _________________________________
__



    def test_add_fail():

>       assert add_func(5, 5) == 9

E       assert 10 == 9

E        +  where 10 = add_func(5, 5)
```

This report is appended with pytest coverage information at the bottom. You can see in a table how many statements are covered by tests.

The coverage of `my_source.py` is at 44%, hence there is some not tested code in the file. You can see in the `Missing` column that line 8 and lines 20-23 are not tested.

Add to this minimal report, you can also export nicely rendered html reports where the uncovered lines of code are marked in red color.

Books on software testing

There are many books on software testing in general and for each programming language out there. Please find here a short list.

- Test-Driven Python Development [Govindaraj2015] (../references.bib).
- Chapter 9 of Clean Code [Martin2008] (../references.bib).
- Python Testing with pytest [Okken2018] (../references.bib).

Exercise: pytest (10 minutes)

- Write a pytest file which checks the implementation of a function we used at beginning of this course

```python
def euclid(p, q):
    """Return the euclidean distance.
    Args:
        p (list): p vector
        q (list): q vector

    Returns:
        euclidean distance
    """
    dist = 0
    for p_i, q_i in zip(p, q):
        dist += (q_i - p_i) ** 2
    return dist ** 0.5
```

Solution

Please find one possible solution in `solution_pytest.py` (solution_pytest.py) file.

In [8]:

```
! pytest - -verbose solution_pytest.py
```

```
============================= test session starts ============================
==

platform linux -- Python 3.7.6, pytest-5.2.4, py-1.8.1, pluggy-0.13.1 -- /home
/rhs2rng/PycharmProjects/py-algorithms-4-automotive-engineering/_venv/bin/pyth
on

cachedir: .pytest_cache

rootdir: /home/rhs2rng/PycharmProjects/py-algorithms-4-automotive-engineering,
inifile: pytest.ini

plugins: cov-2.8.1

collected 1 item



solution_pytest.py::test_euclid PASSED                                     [10
0%]



============================== 1 passed in 0.00s =============================
==
```