Testing

Nikolai Krug

LMU Munich

25.06.2025, Publish Your Own Python Package workshop, Aachen

Outline

Part 1: Motivation of testing and overview

Part 2: Live coding examples with pytest

Sources/further references:

- Martin Ritter's slides from LMU collaborative software development lecture
- Chapter on testing from Henry Schneider's *software engineering in scientific computing*
- PyTest training from Florian Bruhin

When to test your code?

- Imagine you found a bug in your code
- There are large variety of possible causes you have in mind
- You might even have checked all these possible problems during development (but not written persistent tests)
  $\rightarrow$ any later change in the code could reintroduce these problems
  $\rightarrow$ now you have to investigate all these possible causes manually again

Conclusion:

- Write **persistent** tests **during development**
- In a way that they can be **run automatically**

"Code without tests is bad code. It doesn't matter how well written it is; it doesn't matter how pretty or object-oriented or well-encapsulated it is. With tests, we can change the behavior of our code quickly and verifiably. Without them, we really don't know if our code is getting better or worse."
– Michael Feathers, Working Effectively with Legacy Code

Test levels/types

Unit testing to test single units of the program

- usually a single class or function
- harder to implement for complex functions/classes with dependencies

Integration testing to test multiple components software which depend on each other.

- run test scripts and check output
- easier to setup (also for coupled components)
- might be harder to interpret (where does the error come from?)

System testing to test the whole software with respect to the requirements

- usually test input data and output data
- might compare statistical distributions (i.e. resolutions)
- even harder to find cause of error

Operational acceptance testing is where you give it to the user for them to break it

- usually not done in a formal way in science.
- just wait for bug reports :D

Unit tests

Simple example: point class

- works, doesn't crash
- phi is correct

```
class Point2D:
    def __init__(self, x, y):
        self.x = y
        self.y = x

    def phi(self):
        return math.atan2(self.x, self.y)
```

Unit tests

Simple example: point class

- works, doesn't crash
- phi is correct
- x,y is flipped

```
class Point2D:
    def __init__(self, x, y):
        self.x = y
        self.y = x

    def phi(self):
        return math.atan2(self.x, self.y)
```

Unit tests

Simple example: point class

- works, doesn't crash
- phi is correct
- x,y is flipped
- Fixing x,y flip **will break** calculation of phi

```
class Point2D:
    def __init__(self, x, y):
        self.x = y
        self.y = x

    def phi(self):
        return math.atan2(self.x, self.y)
```

Unit tests

Simple example: point class

- works, doesn't crash
- phi is correct
- x,y is flipped
- Fixing x,y flip **will break** calculation of phi

```
class Point2D:
    def __init__(self, x, y):
        self.x = y
        self.y = x

    def phi(self):
        return math.atan2(self.x, self.y)
```

→ Unit tests are precisely meant to find and prevent these issues
→ write small functions that test that the values are what we expect

What to test for?

This is the hardest part

- understand the correct behavior
- which input values cause problems?

Coverage analysis can be helpful

- there are tools to verify how much of your program is tested

Unit tests don't guarantee an error-free program

- even "100%" test coverage in your project doesn't guarantee error-free
- if you find a bug, add a unit test to make sure it doesn't reappear

Test Driven Development

Often tests are written after the software is designed

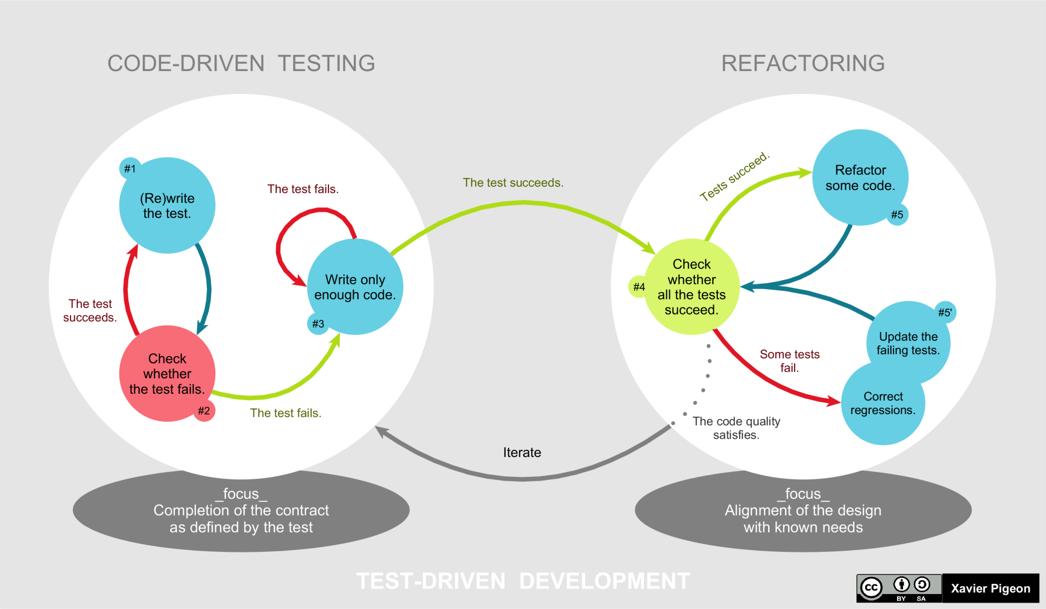
- test coverage is typically low
- you have to understand what to test after you developed it
- writing tests might be huge effort (no testable units)

→**Test Driven Development**

- make testing part of the development
- write tests **before** implementing code
  1. specify what the code should do
  2. write tests to test for the specification
  3. implement the specification

"There is a big difference between mentally knowing about coupling and feeling the pain of coupling... But when we actually write tests, we feel concrete pain. The concrete pain is not because testing is difficult, it's because we need to change our design."
– Micheal Feathers, the deep synergy between testability and good design

Test Driven Development



In simpler terms

For every new feature

1) Write a failing test
2) Write code until it passes
3) Clean up / refactor

Test Driven Development

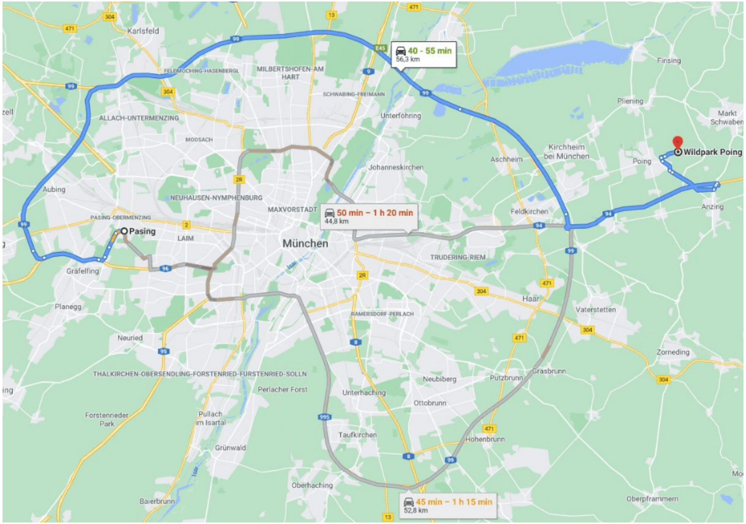
Advantages of TDD

- leads to more robust and correct code
- leads to less monolithic code with less dependencies (you need to write tests)
- helps in maintainability
  - rerun tests after change to ensure software still works (regression testing)
  - tests as "documentation"
- large test coverage helps localize problems

Disadvantages of TDD

- it takes more time. Maybe.

The shortest road is not always the best



. . . and not even the fastest one

Whitebox/Blackbox Testing

For writing tests it makes a difference whether your "know" the internal workings or not

Whitebox testing

- full access to the source
- can design tests by looking at the implementation
- disadvantage: tests might break when you change the implementation

Blackbox testing

- don't look inside, just test the public interfaces
- derive tests from requirements

In practice usually a mixture of both ("Graybox testing")
→better to have tests that you may need to modify/delete later than no tests at all

Testing in Python

Python comes with two distinct unit test frameworks

- doctest – Test interactive Python examples
  This allows to write simple unit tests directly in the the docstring of functions or in text files
- unittest – Unit testing framework
  Normal unit test framework to write test cases/suites with and without fixtures
  - allows more complex testing but has more overhead
  - more similar to other languages

Often used extensions:

- **pytest** – "Helps you write better programs"
  - Makes it easy to write small tests (minimal boilerplate)
  - Scales to support complex functional testing as well
  - Supports both unittests and doctests
- coverage.py – Measure code coverage
  → integrated with pytest using `pytest --cov` (need `pytest-cov` installed)

Testing in Python

Python comes with two distinct unit test frameworks

- doctest – Test interactive Python examples
  This allows to write simple unit tests directly in the the docstring of functions or in text files
- unittest – Unit testing framework
  Normal unit test framework to write test cases/suites with and without fixtures
  - allows more complex testing but has more overhead
  - more similar to other languages

Often used extensions:

- **pytest** – "Helps you write better programs" ← **start with this if unsure what to choose**
  - Makes it easy to write small tests (minimal boilerplate)
  - Scales to support complex functional testing as well
  - Supports both unittests and doctests
- coverage.py – Measure code coverage
  → integrated with pytest using `pytest --cov` (need `pytest-cov` installed)

Doctest

```python
"""example.py"""

def atan2(y, x):
    """Return the arctangent of x,y

    >>> atan2(0, 0)
    0.0
    >>> atan2(1, 0)
    1.5707963267948966
    >>> atan2(0, -1)
    3.141592653589793
    >>> atan2(-1, 0)
    1.5707963267948966
    """
    import math
    return math.atan2(y, x)
```

```
$ python -m doctest example.py
*********************************************
File "example.py", line 12, in example.atan2
Failed example:
    atan2(-1, 0)
Expected:
    1.5707963267948966
Got:
    -1.5707963267948966
*********************************************
1 items had failures:
   1 of   4 in example.atan2
***Test Failed*** 1 failures.
```

Will run all code examples with `>>>` and compare against output

Unittest

```python
"""tests.py"""

import unittest
from example import atan2
from math import pi

class TestAtan2(unittest.TestCase):
    def test_zeroone(self):
        self.assertEqual(atan2(0, 1), 0)

    def test_onezero(self):
        self.assertEqual(atan2(1, 0), pi/2)

    def test_oneminus(self):
        self.assertEqual(atan2(0, -1), pi)

    def test_minuszero(self):
        self.assertEqual(atan2(-1, 0), pi/2)
```

```
$ python -m unittest tests
F...
=============================================
FAIL: test_minuszero (tests.TestAtan2)
---------------------------------------------
Traceback (most recent call last):
  File "tests.py", line 16, in test_minuszero
    self.assertEqual(atan2(-1, 0), pi/2)
AssertionError:
  -1.5707963267948966 != 1.5707963267948966

---------------------------------------------
Ran 4 tests in 0.001s

FAILED (failures=1)
```

Will run all tests it can find

pytest

```python
from example import atan2
from math import pi


def test_zeroone():
    assert atan2(0, 1) == 0


def test_onezero():
    assert atan2(1, 0) == pi/2


def test_oneminus():
    assert atan2(0, -1) == pi


def test_minuszero():
    assert atan2(-1, 0) == pi/2
```

```
$ python -m pytest tests.py
[...]
=============== FAILURES ================
____________ test_minuszero _____________

    def test_minuszero():
>       assert atan2(-1, 0) == pi/2
E       assert -1.5707963267948966 == (3.
141592653589793 / 2)
E        +  where -1.5707963267948966 =
atan2(-1, 0)

test_example.py:20: AssertionError
======== short test summary info ========
FAILED test_example.py::test_minuszero
====== 1 failed, 3 passed in 0.22s ======
```

Will run all functions starting with `test_`

→ Less boilerplate for simple cases

Unit-test frameworks

Unit-test frameworks help with the overhead involved in

- Creating single **test cases**
- Organizing test cases
- Supporting **test fixtures**: common setup and cleanup for all test cases in a test suite
- Providing a **test runner** to execute all or some of the tests and provide the outcome

There are different approaches

- `unittest` follows a more classical, class based approach
- `pytest` provides a more pythonic interface
    - less boilerplate but more implicit behavior
    - also supports doctest/unittest
    - not part of standard python (install with `pip`)

Fixtures and mocking/monkeypatching

Not everything can be tested that easily

Fixtures

- code to be run before/after a test to prepare objects/data/files
- need to properly cleanup, tests should succeed independent of their order

```
import pytest
from unittest.mock import Mock

@pytest.fixture
def dbobject():
    return Mock(**{"query.return_value": 3})

def test_query(dbobject):
    assert dbobject.query("foo") == 3
    dbobject.query.assert_called_once_with("foo")
```

Mocking

- setup objects that imitate interfaces (e.g. database connection)
- inspect how the mock is called
- and define what it should return
- alternative: monkeypatching → monkeypatch fixture in pytest

Parametrized Tests

- some tests might need to be run repeatedly with different input
- can be automated to run different variants of the same test

Summary

Test Driven Development

- improves code quality
- and design
- simplifies changing software

$\rightarrow$ pytest-tutorial

Pytest plugins

Haven't covered pytest plugins - some worth looking into:

- `pytest-xdist`
  → since tests are independent we can run them in parallel
- `pytest-regression`
  → automatically store and possibly regenerate expected values
  → great for testing that larger blocks of values stay the same
- `pytest-mock`
  → integration of unittest.mock into `pytest` (e.g. inspect if function was called)
- `pytest-hypothesis`
  → test properties that hold for arbitrary inputs by inputting random values (fuzzing)
  → useful for parsing code or finding security vulnerabilities

Some recommendations

Reality is not perfect - strict rules/recipes don't always work, but some tips:

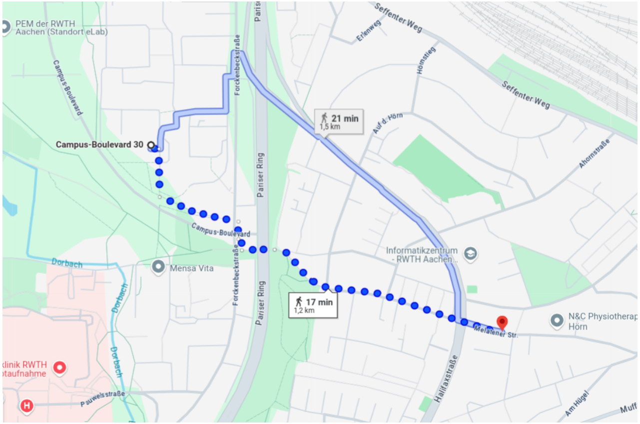
- Need to write some code to try out what you are currently developing?
  → write it as a test
- Found a bug and fixed it?
  → write a test to ensure it doesn't come back
- I don't always write tests before implementation
- But i try to introduce tests as separate commits
- Wrote your test after the code and want to make sure it actually fails without?
  → can use git to move in/reorder history
  (e.g. cherry-pick the commit that introduces the test or rebase)
- I don't have experience testing GUI applications
  → good strategy is probably to focus on testing logic/backend
  → try to seperate the logic as much as possible from the GUI

Validation and other forms of testing

- Linters and Type checkers provide some forms of automated testing
- what's called *Validation* usually much higher level, e.g. physics validation
  $\rightarrow$ look at physics results and compare between different versions of code
  $\rightarrow$ often involves humans looking at plots, but the plots can be produced automatically
- How Henry Schneider puts it in his tutorial:
  - **Verification** (what we discussed so far)
    the code is meeting the requirements you set (is this code correct?)
  - **Validation**
    the requirements you set made sense in the first place (is this the correct code?)

A/B Testing

A/B testing: (typically randomized) experiment between two setups
→ similar to validation: asks if the requirements are actually what we/the users want



(in this case the shortest road was actually at least as fast)