Testing

February 19, 2026

1 Unit testing

1.0.1 What is a unit test

A test that checks the functionality of a program component in isolation.

- A function
- A class
- An interface of a class

1.0.2 Why we test

- To check that the code does what it is supposed to
- To catch type errors in condtionally accesed code
- To make the code testable
- To make bugs shallow
- To document code behaviour
- To know when our job is done
- To communicate requirements
- To get quality measures (coverage)
- To be able to bisect checkins
- To get tracability in the software development process
  - Is a requirement in many fields, like medicine and security

1.0.3 How we test

It is usually not viable to have 100% code coverage, and at any rate, it is more important to cover core algorithms thoroughly than to cover every line of code.

- Test for a typical case
- Test for edge cases
- Test for interesting combinations
- Test that error conditions that you catch raise the right exception

For legacy code, a test that simply loads the code is better than no test at all!

1.0.4 When we test

- While developing
  - Often with a subset of the test suite
  - Should not take more than a few seconds!

- On checkin
  - Always the whole regular test suite
  - Includes all merges and rebases
- Nightly
  - The whole regular test suite
  - May run long running tests in addition

1.0.5 An important note

In interpreted languages, we don't have a compile step, so tests are necessary to ascertain that the entire code base is without syntax errors. Python allows conditional imports, so just starting the application is not a guarantee that the code gets byte compiled.

1.0.6 Test desiderata

Kent Beck, the inventor of Extreme Programming has formulated a number **Test Desiderata**. This is a fancy name for the desired properties of the test suite and the individual tests.

We group them in 4 groups:

Fast to get feedback

- Isolated - run in any order, run any subset
- Run in parallell
- Minimal data

Predictive of deployment success

- Inspire confidence - tested code will do what it should
- Sensitive to behaviour - detect non-functional behaviour
- Sensitive to execution qualities - performance, security systems

Support ongoing changes

- Composable
- Documents intent
- Durable
- Organized
- Positive design pressure
- Necessary (guide choices)

Minimize cost of ownership

- Automated
- Easy to read/write/update
- Diagnosable
- Deterministic
- Insensitive to code structure

1.1 Mocks

In the Python world we talk about mocks as anything that replaces a component in the context of a test.

This is not the terminology that the Sofware Methology community uses. They instead talk about *Test Doubles*.

https://en.wikipedia.org/wiki/Test_double

They then divide the Test Doubles into different categories:

- **Stub** — provides static input.
- **Mock** — verifies output via expectations defined before the test runs (differs from Mock object).
- **Spy** — supports setting the output of a call before a test runs and verifying input parameters after the test runs.
- **Fake** — a relatively full-function implementation that is better suited to testing than the production version; e.g. an in-memory database instead of a database server.
- **Dummy value** — a value that is required for the tested interface but on which the test case does not depend, for instance returning a None value.

1.1.1 *Preliminaries to make pytest work in Jypyter Lab*

The following is setup to make pytest work inside Jupyter Lab.

We need to *pip install ipytest* to get the module.

https://pypi.org/project/ipytest/

[1]:
```
import ipytest
ipytest.autoconfig()

__file__ = "Testing.ipynb"
```

1.2 Parametrized tests

[2]:
```
%%ipytest -qq --disable-pytest-warnings
import pytest

@pytest.mark.parametrize("test_input,expected", [("3+5", 8), ("2+4", 6),␣
 ↪("6*9", 42)])
def test_eval(test_input, expected):
    assert eval(test_input) == expected
```

```
..F
[100%]
============================================ FAILURES
============================================
```

```
_____________________________________ test_eval[6*9-42]
_____________________________________

test_input = '6*9', expected = 42

    @pytest.mark.parametrize("test_input,ex
pected",
[("3+5", 8),
("2+4", 6),
("6*9",
42)])
    def test_eval(test_input,
expected):
>       assert eval(test_input) ==
expected
E       AssertionError: assert 54 == 42
E        +  where 54 = eval('6*9')

/tmp/ipykernel_1493445/3911971697.py:5: AssertionError
=========================================== short test summary info
=========================================
FAILED t_1176832633d441d891f1a8abbee98e3d.py::test_eval[6*9-42] -
AssertionError: assert 54 == 42
```

[3]:
```
%%ipytest -q
import pytest

@pytest.mark.parametrize(
    "test_input,expected",
    [("3+5", 8), ("2+4", 6), pytest.param("6*9", 42, marks=pytest.mark.xfail)],
)
def test_eval(test_input, expected):
    assert eval(test_input) == expected
```

```
..x
[100%]
```

[4]:
```
%%ipytest -s -qq --disable-pytest-warnings
#%%ipytest -qq --disable-pytest-warnings

import pytest


@pytest.mark.parametrize("x", [0, 1])
@pytest.mark.parametrize("y", [2, 3])
def test_foo(x, y):
```

```
    print(x, y)
```

```
0 2
.1 2
.0 3
.1 3
.
```

1.2.1 Fixtures

If you want setup or teardown code to be run before/after tests, pytest has the concetp of fixtures.

The scope for which a fixture is shared is one of *function* (default), *class*, *module*, *package* (experimental) or *session*.

[5]:
```
%%ipytest -qq --disable-pytest-warnings
import pytest

@pytest.fixture
def smtp_connection():
    import smtplib
    print('SMTP SETUP')
    smtp_connection = smtplib.SMTP("smtp.gmail.com", 587, timeout=5)
    yield smtp_connection  # provide the fixture value
    print('SMTP TEARDOWN')
    smtp_connection.close()

def test_ehlo(smtp_connection):
    response, msg = smtp_connection.ehlo()
    assert response == 250
    #assert False  # for demo purposes
```

```
.

[100%]
```

1.2.2 Property based testing

When paramerized tests are not enough, you can go to property based testing, which will let you run a large amount of tests with randomly selected values for the parameters you specify. The tests will record failing tests and minimze them.

The Python library for property based tests is **Hypothesis**

https://hypothesis.readthedocs.io/en/latest/

Here we have made some fairly comprehensive unit tests. We'd probably be ready to trust that this code is correct.

[6]:
```
%%ipytest -qq --disable-pytest-warnings
import pytest
```

```
# The function under test
from typing import List
def max_product(lst: List[int]) -> int:
    '''Return the largest product of two numbers from a list.'''
    if len(lst) < 2:
        raise ValueError()
    b1, b2 = sorted(lst, reverse=True)[:2]
    return b1 * b2
#----------------------------------------------
def test1():
    assert max_product([1, 2, 5]) == 10

def test2():
    assert max_product([5, 1, 2]) == 10

def test3():
    assert max_product([-2, 3, 0, -1, 2, 5]) == 15

def test_fail():
    with pytest.raises(ValueError):
        max_product([1])
```

```
....
[100%]
```

Let's see what hypothesis shows:

[7]:
```
%%ipytest -qq --disable-pytest-warnings
import pytest
from hypothesis import given
from hypothesis.strategies import lists, integers

@given(lists(integers(), min_size=2))
def test_fuzz(lst):
    assert max_product(lst) >= lst[0] * lst[1]
```

```
F
[100%]
============================================= FAILURES
=============================================
_____________________________________________ test_fuzz
_____________________________________________

    @given(lists(integers(),
min_size=2))
>   def test_fuzz(lst):
```

```
/tmp/ipykernel_1493445/1554593529.py:6:
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
- - - - - - - - - -

lst = [-1, -1, 0]

    @given(lists(integers(),
min_size=2))
    def test_fuzz(lst):
>       assert max_product(lst) >= lst[0] *
lst[1]
E       assert 0 >= (-1 * -1)
E        +  where 0 = max_product([-1, -1, 0])
E       Falsifying example: test_fuzz(
E           lst=[-1, -1, 0],
E       )

/tmp/ipykernel_1493445/1554593529.py:7: AssertionError
=================================== short test summary info
====================================
FAILED t_1176832633d441d891f1a8abbee98e3d.py::test_fuzz - assert 0 >= (-1 * -1)
```

1.2.3 Design by Contract

For systems with high reliability requirements, like space applications, medical equipment etc., You may want to apply something called *Design by Contract* It is an idea that comes from languages like *Eiffel* and *ADA*. You use use assertions about *preconditions* and *postconditions* for function calls while the code is in development, debugging and testing. Since the assertions are code, you can do much more than checking types.

The assertions are very suitable for automated integration testing.

The most useful package in Python for Design by Contract is *dpcontracts*.

https://github.com/deadpixi/contracts

In production, you run *python -O*, which will disable the assertions. (python -OO will remove both assertions and docstrings.)

[8]:
```
from typing import List, Any
from dpcontracts import require, ensure, invariant, PostconditionError

@require('lst must not be empty', lambda args: len(args.lst) > 0)
@ensure('result is tail of list', lambda args, result: args.lst == [args.
 ↪lst[0]] + result)
def tail(lst: List[Any]) -> List[Any]:
    return lst[1:]

tail([1])
```

[8]: []

You can also apply *invariants* to a class. This requires the condtions to always hold true for the class.

[9]:
```
@invariant("inner list can never be empty", lambda self: len(self.lst) > 0)
@invariant("inner list must consist only of integers",
           lambda self: all(isinstance(x, int) for x in self.lst))
class NonemptyList:
    @require("initial list must be a list", lambda args: isinstance(args.
 ↪initial, list))
    @require("initial list cannot be empty", lambda args: len(args.initial) > 0)
    @ensure("the list instance variable is equal to the given argument",
            lambda args, result: args.self.lst == args.initial)
    @ensure("the list instance variable is not an alias to the given argument",
            lambda args, result: args.self.lst is not args.initial)
    def __init__(self, initial):
        self.lst = initial[:]

    def get(self, i):
        return self.lst[i]

    def pop(self):
        self.lst.pop()

    def as_string(self):
        # Build up a string representation using the `get` method,
        # to illustrate methods calling methods with invariants.
        return ",".join(str(self.get(i)) for i in range(0, len(self.lst)))

nl = NonemptyList([1,2,3])
nl.pop()
nl.pop()
try:
    nl.pop()
except PostconditionError as e:
    print(e)
```

```
inner list can never be empty
```

[10]:
```
try:
    nl = NonemptyList(["a", "b", "c"])
except PostconditionError as e:
    print(e)
```

```
inner list must consist only of integers
```

1.2.4 Using tox

Tox is a Python package that will:

1. Take a configuration of one or more virtual environments you want to build for a project
   - Each environment may have a different Python interpreter or package config
2. Optionally build a package of your code
3. Build all or selected environments from scratch
4. Run test suite in each environment
5. Make a report
6. Cache environments, so future runs will only rebuild necessary parts

There used to be a plugin called *detox* that would allow work in parallel. This is now built in and is accessed with the *-p* flag.

```
# content of: tox.ini , put in same dir as setup.py
[tox]
envlist = py27,py36

[testenv]
# install pytest in the virtualenv where commands will be executed
deps = pytest
commands =
    # NOTE: you can run any command line tool here - not just tests
    pytest
```

https://tox.readthedocs.io/en/latest/

1.2.5 An article dispelling myths about test driven development

By *Rob Myers*

https://www.agileinstitute.com/articles/dispelling-myths-about-test-driven-development