Testing in Python

François Durand (Nokia Bell Labs)

Python Academy, Lincs, 29 January 2020

GitHub Repo of the Talk

`https://github.com/francois-durand/python_academy_on_testing`

Preliminary Example

From file `my_plus`:

```
def my_plus(x, y):
    return x + y

if __name__ == '__main__':
    assert my_plus(3, 4) == 7
    assert my_plus('a', 'b') == 'ab'
```

Experiment it:

- Run this code.
- What happens if we modify the function with `y + x`?

Benefits of Testing

- Detect quickly when you break something, and what.
- Detect quickly when another developer's contribution breaks something.
- Detect quickly when a change in the environment (OS, database management system, file system) breaks something.
- Incite to think about the inputs and the outputs of a function, and about the API in general. If one see that it is difficult to write tests for some code, it may lead to refactoring in order to make it cleaner and clearer.
- Test-Driven Development (TDD) / Behavior Driven Development (BDD).

Testing Solutions in Python: Overview

- **Unittest**: the standard library of Python. Heavy to use. No real advantage compared to...
- **Pytest**: the package that everybody uses.
- **Doctest**: tests that are included in the doctests.

A typical solution (and I recommend it):

- To write some tests in **doctest** syntax anyway.
- To write some tests in **pytest** syntax optionally for the tests that you don’t want to see in the documentation, or for more advanced tests.
- To use the **pytest runner** to run all the tests: indeed, it is also able to run tests in unittest or doctest format.

Plan

Doctest

Pytest

Running the tests

Conclusion

Plan

Doctest

Pytest

Running the tests

Conclusion

Basic Example

From file `my_get.py`:

```python
def my_get(lst, index, default=None):
    """Get an element in a list by position, with a default value.

    >>> my_get(['a', 'b', 'c'], 2, 'z')
    'c'
    >>> my_get(['a', 'b', 'c'], 42, 'z')
    'z'
    """
    try:
        return lst[index]
    except IndexError:
        return default
```

- Syntax: like interactive Python.
- Can be mixed with regular documentation.
- In PyCharm: “Run...”, then choose Doctest.
- What happens if we change a result?

Ellipsis

From file `MyClass`:

```
class MyClass:
    """A nice class.

    >>> my_object = MyClass()
    >>> my_object  # doctest: +ELLIPSIS
    <...MyClass object at ...>
    """
    pass
```

- What happens without the magic comment?

Skip

From file `my_randint`:

```python
from random import randint

def my_randint(a, b):
    """Random integer.

    Returns
    -------
    int
        A random integer between `a` and `b` (both included).

    Examples
    --------
        >>> my_randint(0, 0)
        0
        >>> my_randint(0, 10)  # doctest: +SKIP
        7
    """
    return randint(a, b)
```

- What happens without the magic comment?

Normalize Spaces

From file `my_np_array`:

```python
import numpy as np

def my_np_array(lst):
    """Convert list to numpy array.

    >>> my_np_array([1, 2, 3642])  # doctest: +NORMALIZE_WHITESPACE
    array([ 1, 2, 3642])
    """
    return np.array(lst)
```

- What happens without the magic comment?
- Here another solution would be to put the exact output (which is deterministic). But it is not always convenient or even possible, so the magic comment may be an option.

Blank Lines

From file `print_some_text`:

```
def print_some_text():
    """Print some text.

    >>> print_some_text()
    This is a fake text.
    <BLANKLINE>
    It features a blank line.
    """
    print("This is a fake text.")
    print()
    print("It features a blank line.")
```

- What happens without the mention `<BLANKLINE>`?

Errors

From file `my_division`:

```
def my_division(x, y):
    """Divide.

    >>> my_division(42, 0)
    Traceback (most recent call last):
    ZeroDivisionError: division by zero
    """
    return x / y
```

- What happens if we remove the expected result?
- For an error, only the first and last lines are necessary in the expected output.

Plan

Doctest

Pytest

Running the tests

Conclusion

Running Example

Remember the function in the file `my_get.py`:

```python
def my_get(lst, index, default=None):
    """Get an element in a list by position, with a default value."""
    try:
        return lst[index]
    except IndexError:
        return default
```

“Manual” Testing Without Pytest

From file `test_my_get_manual.py`:

```python
from python_academy_on_testing.my_get import my_get

my_beautiful_list = ['a', 'b', 'c']
element = my_get(lst=my_beautiful_list, index=2, default='z')
assert element == 'c'
```

- Run this file.
- What happens if we change the result?

Testing With Pytest

From file `test_my_get_pytest.py`:

```python
from python_academy_on_testing.my_get import my_get

def test_get():
    my_beautiful_list = ['a', 'b', 'c']
    element = my_get(lst=my_beautiful_list, index=2, default='z')
    assert element == 'c'
```

- Syntax: functions `test*` in files `test*`.
- In PyCharm: “Run...”, then choose pytest.
- What happens if we change a result?
- Advantages:
  - Clearer message in case of failed test.
  - As we will see later, pytest can automatically run all the tests in the project, has a system of “fixtures”, is configurable, etc.

Errors

From file `test_my_get_pytest_error.py`:

```python
import pytest
from python_academy_on_testing.my_get import my_get

def test():
    with pytest.raises(TypeError):
        my_beautiful_list = ['a', 'b', 'c']
        element = my_get(lst=my_beautiful_list, index='some string', default='z')
```

- Here you need to import pytest.
- If you want more information about the error (for example, to check the error message), you can catch it with the keyword `as`. Cf. function `test_error_message` in the file.

Fixtures

From file `test_my_get_pytest_fixture.py`:

```python
from pytest import fixture
from python_academy_on_testing.my_get import my_get

@fixture()
def my_beautiful_list():
    return ['a', 'b', 'c']

def test_get(my_beautiful_list):
    element = my_get(lst=my_beautiful_list, index=2, default='z')
    assert element == 'c'

def test_missing_element(my_beautiful_list):
    element = my_get(lst=my_beautiful_list, index=42, default='z')
    assert element == 'z'
```

- Here `my_beautiful_list` is available for all the tests.
- But it is not a shared variable. Cf. `test_removed_element` and `test_the_element_is_still_here` in the file.
- Some tests may not use the fixture. Cf. `test_tuple` in the file.

Yield Fixtures

From file `test_my_get_pytest_yield_fixture.py`:

```python
import os
from pytest import yield_fixture
from python_academy_on_testing.my_get import my_get

@yield_fixture()
def my_fid():
    # Setup: done before every test using this fixture. Example: connect to a database, a file...
    fid = open(os.path.join(os.path.dirname(__file__), 'my_file.txt'))

    # The ``return`` is replaced by a ``yield``.
    yield fid

    # Teardown: done after every test using this fixture. Example: disconnect from the database, the file...
    fid.close()

def test_get(my_fid):
    my_beautiful_list = []
    for line in my_fid:
        assert line[0] in {'a', 'b', 'c'}
        my_beautiful_list.append(line[0])
    assert my_get(lst=my_beautiful_list, index=2, default='z') == 'c'
    assert my_get(lst=my_beautiful_list, index=42, default='z') == 'z'
```

A Few Words on Unittest

From file `test_my_get_unittest.py`:

```python
import unittest
from python_academy_on_testing.my_get import my_get


class TestMyGet(unittest.TestCase):

    def test(self):
        my_beautiful_list = ['a', 'b', 'c']
        element = my_get(lst=my_beautiful_list, index=2, default='z')
        self.assertEqual(element, 'c')
```

- The basic syntax is heavier: you need to derive from class `TestCase`.
- The syntax for setup and teardown is also heavier.
- If you define a setup and teardown, you have not choice but to use them for all the tests.

Do not use unittest, use pytest.

Plan

Doctest

Pytest

Running the tests

Conclusion

Running All the Tests

In a shell: `py.test` (without specifying a specific file to test).

In PyCharm (instructions from `https://my-toy-package.readthedocs.io/`):

- Menu Run → Edit Configurations.
- Add a new configuration by clicking the + button → Python tests → pytest.
- Give a name to the configuration, e.g. All tests.
- Ignore the warning and validate.

Default behavior: runs all the functions “test*” included in files “test*”.

Setting Pytest Options

Available solutions:

- In the command line,
- In PyCharm's test configuration, in the field Additional Arguments.
- In a configuration file `pytest.ini` or, better `tox.ini` (we will see why in a few slides). Cf. example in the GitHub Repo.
  - Add a section `[pytest]`.
  - Use the keywords `addopts =` (add options) and put the options here.

Pytest Options

I recommend to always use:

- `--doctest-modules`: runs also the doctests.
- `--showlocals`: show local variables on failed tests.
- `--capture=no`: actually print the `print` instructions of the code.
- `--exitfirst` (or `-x`): the first failure stops the tests.
- `--failed-first`: begin by running the tests that failed last time.

Other useful options:

- `-vv`: very verbose (display more information).
- `--ignore=PATH`: ignore the tests in this path.
- `-k EXPRESSION`: run only the tests containing this string.

Tox and Travis CI

- Tox combines:
  - Automated testing,
  - Virtual environments.
- It runs your tests in several environments: e.g. Python 3.6, 3.7 and 3.8. Or with/without numba, etc.
- It can be used directly as a command-line tool, or via a continuous integration framework such as Travis CI.
- Configuration: file `tox.ini`. Its syntax is compatible with the one of `pytest.ini`, but more general.

To get up and running quickly with these tools, cf. `https://my-toy-package.readthedocs.io/`.

Plan

Doctest

Pytest

Running the tests

Conclusion

Take-Aways

- If you are new to testing, use doctest!
- Use the pytest runner anyway, even if you write only doctests.
- When to write tests?
  - When you just tested your new function or class in a notebook, why not seize the opportunity to make a doctest out of it by a simple copy-paste?
  - When writing the documentation of a function or class.
  - When a bug is detected, write the corresponding test (which fails) then solve it.
  - When you want to improve your coverage.
  - ...

References (1)

Sam & Max website:

- `http://sametmax.com/un-gros-guide-bien-gras-sur-les-tests-unitaires-en-python-partie-1` and the following articles (there are 5 parts).
- `http://sametmax.com/des-astuces-avec-pytest/`.
- `http://sametmax.com/parametres-sympas-pour-pytest/`.
- `http://sametmax.com/parametres-par-defaut-pour-la-commande-py-test/`.
- `http://sametmax.com/se-simplifier-les-tests-python-avec-pytest/`.

Warning: this site contains explicit language and images.

References (2)

Official documentation:

- `https://docs.pytest.org/`.
- `https://docs.python.org/3/library/doctest.html`.

My Toy Package contains instruction to configure a package with all the convenient development tools, including testing: `https://my-toy-package.readthedocs.io/`.

Thanks For Your Attention!

NOKIA