Software testing in Python

Janusz Jabłonowski, MIMUW

Testing only causes problems

Why test at all?

- (Some) managers (at various levels) do not like testing
  - “Cases are going up in the U.S. because we are testing far more than any other country, and ever expanding. With smaller testing we would show fewer cases!”
    D. Trump, (probably CEO level manager, already fired now, but he does not admit it)
  - “When you test, you create cases” Same expert as above
  - (I am not 100% sure that it was about programming, have to check it)
- My code is the best in the universe, why should I test?
- I should know what my program is doing

Testing solves all problems with code quality

- “Testing proves that your code work as it’s supposed to in response to all the input types it’s designed to receive.”
  “Python Crash Course”, Eric Matthes, no starch press, p. 215, second printing.

So, what are tests good for?

- People make mistakes
  - “To err is human, to repent divine, to persist devilish.” Benjamin Franklin
- Test show errors in our code
  - Which makes them extremely useful!
- Lack of errors during testing proves nothing
  - Still they are extremely useful!
- Creating good tests is difficult
  - The art of testing
- Who should create tests
  - The programmer is not interested in proving that their code is buggy
  - Still some testing should be done by the programmer
- TDT - Test driven development
  - Start with tests, not with code!

Problems with tests

They **do not** prove program correctness

- Not those parameters
  - How to choose the right ones?
- No complete coverage
  - programs are complex
- Not this interleave
  - testing concurrent programs is a way bigger problem than with the sequential ones
- Randomness in program behaviour
  - e.g. uninitialized variables
- Users are different
  - ‘I would never enter an empty string in this field, it does not make any sense!”

So, are they worth the effort?

The final conclusion

- They are difficult to do
- They do not prove the correctness
- They cost (creating, maintaining, CPU time, program size on disk)

Still in doubts?

Then, for example, think about the lock in your entry door at your home and compare with the list above.

(Or the password to your bank account.)

Some hints

This list is far from being complete (none is complete)

- Test usual cases first
- Look for corner cases
- Use random data in testing too

Various types of testing

Who organizes test

- Manual
- (Semi)automatic

What is being tested

- Unit tests
- Integration tests

Terminology

- *Unit test* - a single test, of a single aspect of a function being tested
- *Test case* - a collection of unit tests, for example for one function
- *Full (test) coverage* - a set of unit tests during which all control flows through a function body had been tested

Tools for testing (presented on these slides or during tutorials)

- Plain code (with `if` and `print`)
- `assert`
- doctest
- unittest
- pytest

Plain code

- Lots of problems
- Mixed code and test
- Difficult to switch off
- Difficult to actually run the test

The assert command

- Slightly better
- Can be turned out by the -o option when invoking the Python program

doctest

- Very simply and extremely useful
- Allows for tests as specification
- Tests are written within docstrings
- They mimics interactive console syntax

doctest example

A sample function

```
def value(polynomial, x):
    """
    :param polynomial: the polynomial that is being calculated
    :param x: the value for which the polynomial is to be calculated
    :return: the value of the polynomial for x polynomial(x)
    """
    res, power = 0, 1
    for coeff in polynomial:
        res += power*coeff
        power *= x
    return res
```

doctest example

A sample function with tests (the rest of the function body is skipped)

```
def value(polynomial, x):
    """
    >>> value([], 1)
    0
    >>> value([1,2,1], 2)
    9
    """
```

doctest example

Code invoking doctest

```
if __name__ == "__main__":
    import doctest
    doctest.testmod()
```

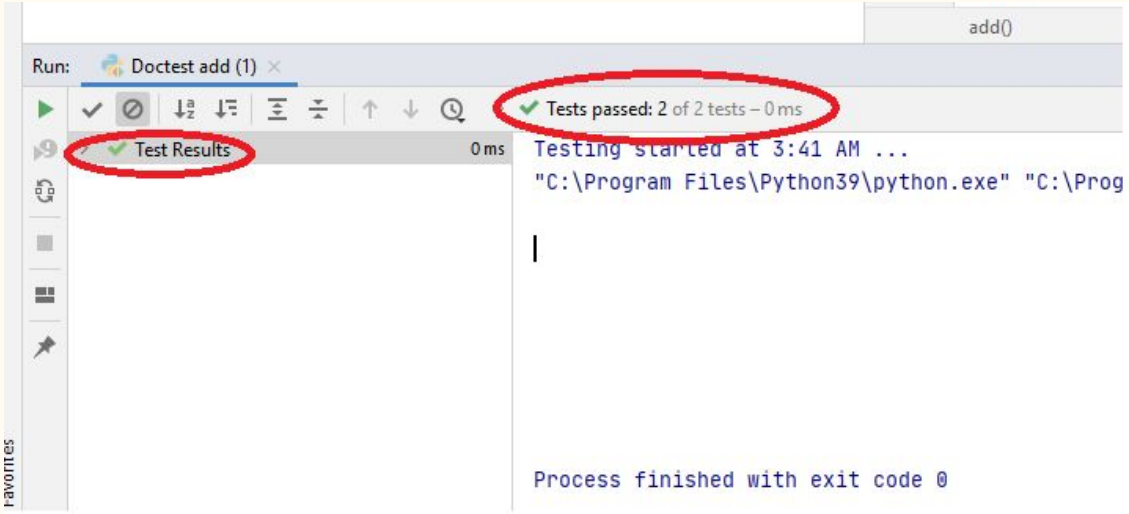
doctest: entire (simpler) example

```
def add(x,y):
    """
    >>> add(1,2)
    3
    >>> add(-2,2)
    0
    """
    return x+y

if __name__ == "__main__":
    import doctest
    doctest.testmod()
```

Invoking doctest

- IDE
  - some IDEs support doctest
  - e.g. Pycharm displays test results above the run pane



Invoking doctest

- Command line

```
M:\Zajęcia\Narzędzia\Python>python addition.py

M:\Zajęcia\Narzędzia\Python>
```

- Nothing written because all test were successful
  - ti is not the brightest idea of this (really very, very good) solution
- Doctest may be forced to output more text (see next slide)

Invoking doctest

```
M:\Zajęcia\Narzędzia\Python>python addition.py -v
Trying:
    add(1,2)
Expecting:
    3
ok
Trying:
    add(-2,2)
Expecting:
    0
ok
1 items had no tests:
    __main__
1 items passed all tests:
   2 tests in __main__.add
2 tests in 2 items.
2 passed and 0 failed.
Test passed.
```

More about doctest

- Any expression can be given after >>>
- Any expected (even multiline) test as result
- But beware that spaces matter here!

```
"""
>>> [ add(1,2), add(-2,2), add(0,0), add(13123,232)]
[3, 0, 0, 13355]
>>> [ add(x,y) for x,y in [(1,2), (-2,2), (0,0), (13123,232)]]
[3, 0, 0, 13355]
"""
```

More about doctest

- Even exceptions may be checked
- Use `...` to skip unimportant part of the result

```
def add1(x):
    """
    >>> add1(2)
    3
    >>> add1('a')
    Traceback (most recent call last):
    ...
    ValueError: x must be a number
    """
```

(see next slide)

More about doctest

Body of add1

```
import numbers
if isinstance(x, numbers.Number): # test for being a number
    return x+1
else:
    raise ValueError("x must be a number")
```

More about doctest - another version of `add1`

```
def add1(x):
    """
    >>> [x for x in [1, 2.0, 3j, complex(4,5)]]
    [1, 2.0, 3j, (4+5j)]
    >>> add1('a')
    Traceback (most recent call last):
    ...
    ValueError: x must be a number
    """
    if type(x) == int or type(x) == float or type(x) == complex:
        return x+1
    else:
        raise ValueError("x must be a number")
```

doctest - failed tests

```
def add2(x):
    """
    >>> add2(1)
    1
    """
    return x + 2
**********************************************************************
File "[...]", line 28, in __main__.add2
Failed example:
    add2(1)
Expected:
    1
Got:
    3
**********************************************************************
1 items had failures:
   1 of   1 in __main__.add2
***Test Failed*** 1 failures.
```

doctest - what is checked

- doctest analyses all docstrings of
  - module
  - functions
  - classes
  - methods
- Nothing else (e.g. the second triple quotations literal in a function is not analysed)
- But the following syntax enables for analysing contents of any text file

```
doctest.testfile("example.txt")
```

doctest - syntax of tests

- The form of a test
    a. Series of lines starting with `'>>> '` or `'... '` with Python code
    b. Series of lines with the expected output (ended by a `'>>> '` or empty line or end of the text being analysed (end of a docstring, end of file)
- The starting column of code does not matter
- The output starts of the last column of the `'>>> '` string
- (see the example on the next slide)

Doctest examples

```
def add1(x):
    """
>>> x = 1
  >>> x += 1
    >>> add1(x)
    3
    """
    return x+1
```

Is fine but with (spaces before 3 added, other lines skipped)

```
    >>> add1(x)
     3
```

is treated as failed test, and the following is an error in test itself (“inconsistent leading whitespace”)

```
    >>> add1(x)
  3
```

The `unitest` module

- Very popular
- Similar to other languages
- Tests are separated from code
- Test files should be named test_*.py
- (cont...)

The `unitest` module

- It uses class syntax - tests are methods of a class inheriting from `unittest.TestCase`
- Just structure the test files as follows:

```
import unittest
class <any_name>(unittest.TestCase):
<a list of indented test functions with the self parameter>
```

- The file with test is just a Python program!
- An advantage: you can run all tests in your project at any time
- Supported in IDEs (e.g. Pycharm RMB>Go to>Test Ctrl/Shift/T)

The test methods

- These methods names should start with test_
- It allows unittest for automatic finding and calling them
- May contain any code and call to a method from unittest
- They do not need to return any results

Simple module for testing ...

```
def add1(x):
    if type(x) == int or type(x) == float or type(x) == complex:
         return x+1
    else:
        raise ValueError("x must be a number")
```

... and a simple test file for it

```
import import unittest
from arithmetic.adding import add1

class Test(unittest.TestCase):
    def test_add1(self):
        self.assertEqual(add1(1), 2)
```

- And the result

```
.
----------------------------------------------------------------------
Ran 1 test in 0.000s

OK
```

Preparing tests

- Sometimes test require some environment (like initializing some variables, reading data files, creating objects)
- It can be done in each testing method separately
- But if there is a common part then it can be mode to the `setup` method (function)
- It gets executed before each test
- Variables defined within `self.` in its body are visible in the testing methods

Preparing tests - fixtures

- Similarly a tearDown() method cleans after a test case suit
- A test fixture - an environment for a test case suit
- A TestCase instance represents such a test fixture
- The setUp, tearDown, and __init__ methods are called once per test

Fixture example

- For a function

```
def find(tree, value):
    if not tree:
        return False
    elif value == tree[0]:
        return True
    else:
        return find(tree[1 if value < tree[0] else 2], value)
```

- a complex data structure is needed
- (cont...)

Fixture example

```
import unittest

class Test(unittest.TestCase):
    def setUp(self):
        self.tree1 = []
        self.tree2 = [5, [3, [1, [], []], [2, [], []]], [8, [6, [], [7, [], []]],[]]]

    def test_find1(self):
            from BST_functions import find
            self.assertTrue(find(self.tree2, 8))

    def test_find2(self):
            from BST_functions import find
            self.assertFalse(find(self.tree2, 4))
```

Available methods in unittest

(Some of the) available methods (note the camelCase name syntax)

- assertEqual(val1, vl2)
- assertNotEqual(val1, val2)
- assertTrue(val)
- assertFalse(val)
- assertIn(item, list)
- assertNotIn(item, list)
- assertRaises(exception)

Executing

- Command line version 3:

```
python -m unittest discover
```

- or equivalently

```
python -m unittest
```

- runs all test in the current directory

Executing

- IDE support
- Command line version 1:

```
python -m unittest test_adding.py
```

- Command line version 2

```
python test_adding.py
```

- But requires in the test module

```
if __name__ == "__main__":
    unittest.main()
```

Thank you for your attention!