Python in the enterprise

Unit test report

Dawid Gerstel & Emilia Zacharjasz

May 28, 2016

> No amount of experimentation can ever prove me right; a single experiment can prove me wrong.
>
> *Albert Einstein*

1 Introduction

The objective of this project was to get acquainted with unit testing in Python based on the unittest module.

The application obtained is in the `./application_to_be_tested` catalogue. There are two test files inside the same catalogue, namely, `test1_inputValidatorImpl.py` and `test2_Trintegratorimpl.py`.

2 Application functionality

First, we focused on the functionality of the application. Although, the project seems to be quite complicated (there are many files) – its core functionality is performing integration of an input function. The data flow consists of:

- reading the data from user,
- validating them,
- computing integral of the expression provided (in a given range and number of its divisions),
- displaying the output of the operation.

From the user perspective it is essential that the input is read correctly and the integral is calculated accurately enough. After we made sure we understand the application code we concluded that the following classes need to be tested:

- `TrIntegratorImpl`: which performs integration itself,
- `InputValidatorImpl`: which validates the user-provided input in various ways.

These classes comprise all essential and sensitive points of the application.

3 Performing unit tests

3.1 Input validation

The Class `InputVaidatorImp` contains all considered (by the author of code) forms of validation and exceptions in application. We decided to test the following aspects:

- Allowed input function characters (testing class `TestInputValidatorAllowedCharacters`):
  - Does program interpret "xx"?
  - Does program allow letters, e.g. "a"?
- Proper integration limits (class `TestInputValidatorIntegrationLimits`):
  - Can non-numerical values be passed as integration start or end?
  - Can the start value be greater than the end value? [1]
- Integration range divisions(class `TestInputValidatorSplitsAmountException`):
  - Can number of divisions equal 0?

[1] Obviously, mathematically it is proper, but the program assumes the start value is smaller than the end value.

– Additionally, for practice, we wanted to check if none exceptions are raised if the number of divisions is correct, e.g. 10.

Listing 1: `test1_inputValidatorImpl.py`

```
from TrIntegratorImpl import TrIntegratorImpl

from Integral import Integral

from InvalidFuncException import InvalidFuncException
from InvalidIntegrationPointException import InvalidIntegrationPointException
from InvalidIntegrationSplitsAmmountException import InvalidIntegrationSplitsAmmountException

from InputValidatorImpl import InputValidatorImpl

import unittest


class TestInputValidatorAllowedCharacters(unittest.TestCase):
  def setUp(self):
    self.inpValImpl = InputValidatorImpl()

  def testInputValidatorDoubleX(self):
     self.assertRaises(InvalidFuncException,  self.inpValImpl.validateIntegralFunc, "xx")

  def testInputValidatorLetters(self):
     self.assertRaises(InvalidFuncException,  self.inpValImpl.validateIntegralFunc, "a")

class TestInputValidatorIntegrationLimits(unittest.TestCase):
  def setUp(self):
    self.inpValImpl = InputValidatorImpl()

  def testInputValidatorIntegrationStart(self):
     self.assertRaises(InvalidIntegrationPointException, self.inpValImpl.validateIntegrationStart, "x")

  def testInputValidatorIntegrationEnd(self):
     self.assertRaises(InvalidIntegrationPointException, self.inpValImpl.validateIntegrationEnd, "3", "x")

  def testInputValidatorIntegrationRangePermutation(self):
     self.assertRaises(InvalidIntegrationPointException, self.inpValImpl.validateIntegrationEnd, "3", "2")

class TestInputValidatorSplitsAmountException(unittest.TestCase):
  def setUp(self):
    self.inpValImpl = InputValidatorImpl()

```

```
    def testInputValidatorSplitsAmountException(self):
        self.assertRaises(InvalidIntegrationSplitsAmmountException, self.inpValImpl.validateSplitsAmmount, "0")

    def testInputValidatorSplitsAmountExceptionWithCorrectInput(self):
        try:
            self.inpValImpl.validateSplitsAmmount("10")
        except InvalidIntegrationSplitsAmmountException:
            self.fail("InputValidatorImpl.validateSplitsAmmount('10') raised InvalidIntegrationSplitsAmmountException unexpectedly!")


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

3.2 Integration accuracy

In order to check only integration accuracy we decided that the integral expression and specifications be hard-coded. The integral we chose is $\int_0^1 2x+1dx = 2$. The listing below shows this unit test. Since the numerical integration has some inherent error we used `EPSILON` as a tolerance parameter of the difference between analytical and numerical results. In class `TestTrintegratorAccuracy` we exploited `unittest.TestCase.assertTrue` method to formulate this condition.

Listing 2: `test2_Trintegratorimpl.py`

```
from TrIntegratorImpl import TrIntegratorImpl

from Integral import Integral

import unittest

EPSILON = 1e-10

class TestTrintegratorAccuracy(unittest.TestCase):
    def setUp(self):
    # create Integral instance
        self.integral = Integral
        self.integral.func = "2*x+1"
        self.integral.integrationStart = 0
        self.integral.integrationEnd = 1
        self.integral.splitsAmmount = 101

        # create TrIntegratorImpl instance
        self.trint = TrIntegratorImpl()
        self.trint.init(self.integral)

    def test_calculate_integral_return_corr_res(self):
        self.assertTrue(self.trint.calculate() - 2 < EPSILON)

```

```

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

4 Testing results

The input validation tests gave one failure in its output. The application validation accepts "xx" and does not raise `InvalidFuncException`, although the program does not recognise this expression and stops immediately. The other six tests yielded positive and that means the program acts accordingly in those situations. It throws expected exceptions.

Listing 3: Input validation unit test output

```
$ python test1_inputValidatorImpl.py
F......
======================================================================
FAIL: testInputValidatorDoubleX (__main__.
    TestInputValidatorAllowedCharacters)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "test2_inputValidatorImpl.py", line 20, in
      testInputValidatorDoubleX
    self.assertRaises(InvalidFuncException,  self.inpValImpl.
        validateIntegralFunc, "xx")
AssertionError: InvalidFuncException not raised

----------------------------------------------------------------------
Ran 7 tests in 0.002s

FAILED (failures=1)
```

The accuracy testing showed the application calculated the given integral precisely enough (within the assumed tolerance). Of course, we did not perform any in-depth analysis of the integration precision. Our goal was merely to check if the integration result is reasonable.

Listing 4: Integration accuracy unit test output

```
$ python test2_Trintegratorimpl.py
.
----------------------------------------------------------------------
Ran 1 test in 0.004s

OK
```

5 Conclusion

We familiarised ourselves with the foundations of unit testing in Python. After carrying out the project we formed an opinion that `unittest` module is a useful and quite straightforward utility.

Because `unittest` allows for testing single functionalities, tester does not need to know the application fully. It is enough one understands a chosen class or function and then a satisfactory unit test may be implemented.