Unit Testing in Python

James Brucker

Python Unit Test Libraries

Doctest - testable examples in docstring comments

Unittest - the standard test library, based on JUnit

Pytest - simple yet powerful package for concise tests. Can execute doctests & unittests, too.

Libraries to Enhance Tests

Mock objects - "fake" objects for external components

- also called "test doubles"

Hamcrest - declarative rules of "intent" to help write readable, powerful matching rules for tests.

unittest example

```
import unittest

class TestBuiltins(unittest.TestCase):
    """Test some python built-in methods"""

    def test_len(self):
        self.assertEqual(5, len("hello"))
        self.assertEqual(7, len(" el lo "))
        self.assertEqual(0, len("")) # edge case

    def test_isupper(self):
        self.assertTrue( "ABC".isupper() )
        self.assertFalse( "ABc".isupper() )
```

class extends `TestCase`

Test method name must begin with **`test_`**

How to Write an "assert"

docstring will be shown on test output

expected result

actual result

```
    def test_len(self):
        """length of a string is number of chars"""
        self.assertEqual(5, len("hello"))


    def test_isupper(self):
        self.assertTrue( "ABC".isupper() )

        self.assertFalse( "ABc".isupper() )
```

should be True

should be False

Run tests from the command line

Run all tests or just specific tests.

```
cmd> python -m unittest test_module

cmd> python -m unittest tests/test_module.py

# print verbose test results
cmd> python -m unittest -v test_module

# auto-discovery: run all test_*.py files
cmd> python -m unittest

# print help
cmd> python -m unittest -h
```

Other Ways to Run tests

1. Use your IDE run the tests.
2. Use a test script or build tool.
3. Add a "`main`" block to your Test file...

```
import unittest

...

if __name__ == "__main__":
    unittest.main()    # or unittest.main(verbose=2)
```

Exercise: Try it Yourself

Test math.sqrt() and math.pow().

```
import unittest
import math

class MathTest(unittest.TestCase):
    def test_sqrt(self):
        self.assertEqual(5, math.sqrt(25))
        self.assertEqual(0, math.sqrt(0)) #edge case

    def test_pow(self):
        #TODO Write 1 or 2 tests of math.pow(x,n)
```

Exercise: Run Your Tests

Run on the command line:

```
cmd>  python -m unittest test_math
..
----------------------------------------------------
Ran 2 tests in 0.001s
```

Run with verbose (-v) output

```
cmd>  python -m unittest -v test_math.py
test_sqrt (test_math.MathTest) ... ok
test_pow (test_math.MathTest) ... ok
----------------------------------------------------
Ran 2 tests in 0.001s
```

Exercise: Write two Failing Tests

```
import unittest
import math

class MathTest(unittest.TestCase):
    # This answer is WRONG. Test should fail.
    def test_wrong_sqrt(self):
        self.assertEqual(10.0, math.sqrt(100.000001))

    # This is ILLEGAL.  Cannot sqrt a negative value.
    def test_sqrt_of_negative(self):
        self.assertEqual(-4, math.sqrt(-16))
```

Exercise: Run the Tests

Run on the command line:

```
cmd>  python -m unittest math_test.py
..EF
===================================================
ERROR: test_sqrt_of_negative (math_test.MathTest)
---------------------------------------------------
Traceback (most recent call last):
  File "test_math.py", line 10, in test_sqrt_negative
    self.assertEqual(4, math.sqrt(-16))
ValueError: math domain error
===================================================
FAIL: test_wrong_sqrt (test_math.MathTest)
Trackback (most recent call last):
AssertionError: 1 != 5.0
```

Test Results

At the end, unittest prints:

```
Ran 4 tests in 0.001s
FAILED (failures=1, errors=1)
```

How are "failure" and "error" different?

Failure means a test condition (assertion) failed

assertEquals(except, actual)

fail("it didn't work")

expected an exception, but exception not raised

Error means some code caused an error

Tests Outcomes

Success: passes all "assert"

Failure: fails an "assert" but code runs OK

Error: error while running test, such as exception raised

What Can You assert?

```
assertTrue( gcd(-3,-5) > 0 )
assertFalse( "hello".isupper() )
assertEqual( 9,  math.pow(3,2))
assertNotEqual( "a", "b")
assertIsNone(a)              # test "a is None"
assertIsNotNone(a)           # test "a is not None"
assertIn(a, list)            # test "a in list"
assertIsInstance(3, int)   # test 3 in an "int"
assertListEqual(list1, list2) # all elements equal
```

Many more!
See "unittest" in the Python Library docs.

Use the Correct assert

Use the 'assert' that matches what you want to test.

Good asserts (matches what you want to verify):

```
assertEqual( 5, math.sqrt(25))

assertGreater( math.pi, 3.14159)

assertNotIn('a', ['yes','no','maybe'])
```

Don't write this:

```
assertTrue(5 == math.sqrt(25))

assertIs(math.pi > 3.14159, True)

assertTrue( math.pi > 3.14159 )

assertFalse('a' in ['yes','no','maybe'])
```

Test involving Floating Point

Calculations using floating point often result in *rounding error* or *precision error*.

Try this:

```
>>> 2.0 - 1.1 == 0.9

True or False?
```

Tests for Floating Point

Use **`assertAlmostEqual`** to test a result which may have rounding error:

1) assertAlmostEqual(a, b, places=n) tests $| a - b | < 10^{-n}$

```
self.assertAlmostEqual(
        1.33333, 4.0/3.0, places=5)
```

2) assertAlmostEqual(a, b, delta=d) tests $| a - b | <= d$

```
# delta = allowed difference in values
self.assertAlmostEqual(
        0.33333, 1.0/3.0, delta=0.00001)
```

Skip a Test or Fail a Test

```
import unittest
class MyTest(unittest.TestCase):


    @unittest.skip("Not done yet")
    def test_add_fractions(self):
        pass


    def test_fraction_constructor(self):
        self.fail("Write this test!")
```

Test for Exception

What if your code should throw an exception?

```
def test_sqrt_of_negative( self ):
    """sqrt of a negative number should throw
       ValueError.
    """
    self.assert????( math.sqrt(-1) )
```

Test for Exception

`assertRaises` expects a block of code to raise an exception:

```
def test_sqrt_of_negative(self):
    with self.assertRaises(ValueError):
        x = math.sqrt(-1)
```

Exercise: use **`assertRaises`**

Add `assertRaises` expects to your sqrt test:

```
def test_sqrt_of_negative(self):
    with self.assertRaises(ValueError):
        result = math.sqrt(-1)
        result2 = math.log(-4) # not reached
```

Can we do this?

`assertRaises` with extra argument:

```
def test_sqrt_of_negative(self):
    self.assertRaises(ValueError, math.sqrt(-1))
```

This **doesn't work**.

A ValueError exception is thrown (the test fails).

Which Operation is Done 1st, 2nd, ..?

```
print("sqrt 5 + 1 is", 1 + math.sqrt(5))
```

Which operation is done first?

```
def test_sqrt_of_negative(self):
    self.assertRaises(ValueError, math.sqrt(-1))
```

Python evaluates math.sqrt(-1) before calling assertRaises.

So it raises an uncaught exception.

The Python Docs State:

assertRaises(*exception*, *callable*, **args*, ***kwargs*)

What is a *callable*?

Something that you can call. :-)

Example: a function, a lambda expression

Use a callable in assertRaises

`assertRaises` with callable:

```
def test_sqrt_of_negative(self):
    self.assertRaises(ValueError, math.sqrt, -1)
```

*args passed to the callable

Don't test multiple exceptions in one "assertRaises" block

The Cash class constructor should raise exception if
a) value (1st param) is negative
b) currency (2nd param) is an empty string

This test will fail to detect some errors. Why?

```
def test_cash_constructor(self):
    with self.assertRaises(ValueError):
        c1 = Cash(-1, "Baht")
        c2 = Cash(10, "")
```

What to Name Your Tests?

1. Test **methods** begin with `test_` and use snake case.

```
def test_sqrt(self)

def test_sqrt_of_negative_value(self)
```

2. Test **class name** either starts with Test (Python style) or ends with "Test" (JUnit style). Use CamelCase.

```
class TestMath(unittest.TestCase)

class MathTest(unittest.TestCase)
```

What to Name Your Tests?

3. **Test filename** should start with **`test_`** & use snake case

   `test_math.py`

   `test_list_util.py or test_listutil.py`

Note:

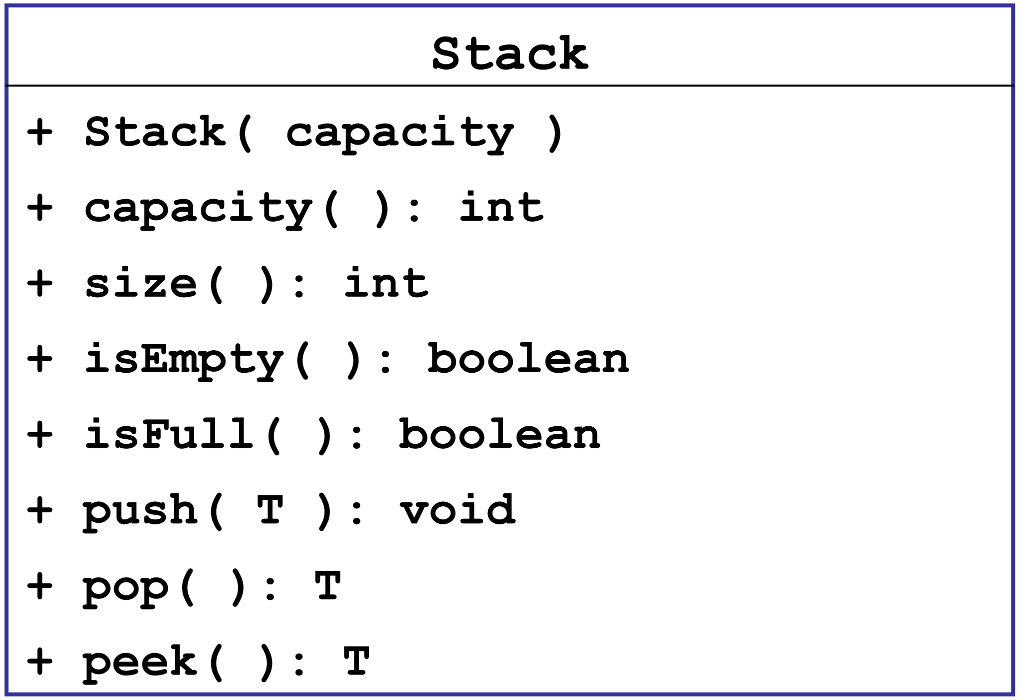
if test filename **ends** with _test like **`math_test.py`** then Python's "test discovery" feature won't discover the tests unless you use -p ("pattern"):

```
python -m unittest -p "*_test.py"
```

Test setUp for a Stack test

A Stack implements common stack data structure.

Throws StackException if you do something stupid.



```
Stack
+ Stack( capacity )
+ capacity( ): int
+ size( ): int
+ isEmpty( ): boolean
+ isFull( ): boolean
+ push( T ): void
+ pop( ): T
+ peek( ): T
```

Stack Tests all Need a Stack

In each test creates a new stack.

That's a lot of **duplicate code**.

How to eliminate duplicate code?

```
def test_new_stack_is_empty(self):
    stack = Stack(5)
    self.assertTrue( stack.isEmpty() )

def test_push_and_pop(self):
    stack = Stack(5)
    stack.push("foo")
    self.assertEqual("foo", stack.pop() )
    self.assertTrue( stack.isEmpty() )
```

Use setUp() to create test fixture

setUp() is called before **each** test.

```
import unittest

class StackTest(unittest.TestCase):
    # Create a new test fixture before each test
    def setUp(self):
        self.capacity = 5
        self.stack = Stack(capacity)

    def test_new_stack_is_empty(self):
        self.assertTrue( self.stack.isEmpty() )
        self.assertFalse( self.stack.isFull() )
        self.assertEqual( 0, self.stack.size() )
```

In unit testing, what is `setUp`() ?

What is the purpose of setUp?

* create a "test fixture" containing objects or whatever your tests need

* avoids redundant code in many tests

* TestCase invokes setUp before each test.

* "setUp" (or equivalent) is available in Unittest, Pytest, JUnit, and other xUnit frameworks.

How to clean up after each test ?

Example: you read test data from a file.

You should close the file after each test.

Example: your tests write data to a file.

You want to delete the file after each test.

Solution:

`tearDown(self)` is called after each test.

Use tearDown() to clean up after test

tearDown() is called after each test. Its not usually needed, since setUp will re-initialize a test fixture.

```
class FileTest(unittest.TestCase):

    def setUp(self):
        # open file containing test data
        self.file = open("testdata", "r")

    def tearDown(self):
        try:
            self.file.close()
        except Exception:
            pass
```

setUp Done Once Per Run

There is a method you can use to initialize the TestCase class before any tests are run.

This is done only once and its a class method.

Example: open a database or network connection one time before running any of the tests.

What is the method?

```
@classmethod
def setUpClass(cls):
    # perform initialization for this
    # test suite.
```

Doctest

Include runnable code inside Python DocStrings.

Provides example of how to use the code
and executable tests!

```
def average(lst):
    """Return the average of a list of numbers.

    >>> average([2, 4, 0, 4])
    2.5
    >>> average([5])
    5.0
    """
    return sum(lst)/len(lst)
```

doctest
comments

Running Doctest

Run doctest using command line:

```
cmd> python -m doctest -v listutil.py
2 tests in 5 items.
2 passed and 0 failed.
Test passed.
```

Or run doctest in the code:

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

Doctest for Exception

Use ". . ." to indicate omitted lines of output.

The test passes if the exception is raised.

```
def average(lst):
    """Return average of a list of numbers.
    >>> average([])
    Traceback (most recent call last):
     ...
    ValueError: list must not be empty
    """

    if len(lst) == 0:
        raise ValueError("list must not be empty")
    return sum(lst)/len(lst)
```

Testing is Not So Easy!

These examples are *trivial tests* to show the syntax.

Real tests are more thoughtful and demanding.

Designing good tests makes you **think** about what the code should do, and what may go wrong.

Good tests are often short... but many of them.

References

Python Official Docs - easy to read, many examples

`https://docs.python.org/3/library/unittest.html`

Good article & how to run unit tests in an IDE

`https://realpython.com/python-testing/`

Video shows how to use unittest

`https://youtu.be/6tNS--WetLI`