



# Unit Testing in Python

---

James Brucker

# Python Testing Frameworks

We will cover these two:

- **unittest** - part of the Python library, similar to JUnit 3
- **DocTest** - test by example, part of the Python library

Other testing frameworks:

- **Py.Test** - very simple "assert" syntax.
  - can also run unittest style tests
- **Mock objects** - create "fake" external components
- <https://wiki.python.org/moin/PythonTestingToolsTaxonomy>

# unittest example

```
import unittest
```

class extends TestCase

```
class TestBuiltins(unittest.TestCase):
    """Test some python built-in methods"""
    def test_len(self):
        self.assertEqual(5, len("hello"))
        self.assertEqual(3, len(['a', 'b', 'c']))
        # edge case
        self.assertEqual(0, len(""))

    def test_str_upper(self):
        self.assertTrue("ABC".isupper())
        self.assertFalse("ABc".isupper())
        s = "" # edge case
        self.assertFalse(s.isupper())
```

# Run tests from the command line

Run all tests or just specific test.

```
cmd> python -m unittest test_module
```

```
cmd> python -m unittest module.TestClass
```

```
cmd> python -m unittest tests/test_module.py
```

# Other Ways to Run tests

1. Let the IDE run them for you.
2. Use a test script or build tool.
3. Add a "main" script to end of your Test class...

```
import unittest

class TestBuiltins(unittest.TestCase):
    """Test some python built-in method"""
    def test_len(self):
        self.assertEqual(5, len("hello"))
        self.assertEqual(3, len(['a', 'b', 'c']))

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

# 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 some 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
```

# 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(1, math.sqrt(25))

    # sqrt of negative number is not allowed.
    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

The test summary prints:

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

How are "failure" and "error" different?

Failure means \_\_\_\_\_

Error means \_\_\_\_\_

# 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( 2*2, 4)
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 isinstance(a,b)
assertListEqual( list1, list2 ) # all elements equal
```

Many more!

See "unittest" in the Python Library docs.

# 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):  
        math.sqrt(-1)
```

# What to Name Your Tests?

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

```
def test_sqrt(self)
```

```
def test_sqrt_of_negative(self)
```

2. **Test class name** either starts with `Test` (Python style) or ends with `"Test"` (JUnit style, ok). 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 file ends with `_test` like `math_test.py` then Python's "test discovery" feature (used by Django) won't run the tests unless you write:

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

# Exercise: Test Driven Development

Write some tests for this function before you write the function body. Just return 0:

```
def average( lst ) :  
    """Return average of a list of numbers"""  
    return 0
```

# Exercise: Define Test Cases

1. **Typical case:** list contains a few numbers
2. **Edge cases:** a) list with only 1 number,  
b) list with many values all the same,  
c) list containing some 0 values (changes average).
3. **Illegal case:** empty list

*What should happen in this case??\*\**

TDD forces you to think about  
what the code *should do*.

*\*\*Hint: Python has a builtin `max(list)` function.*

# Write the Tests

10 minutes

File: test\_average.py

```
import unittest
from listutil import average
class TestAverage(unittest.TestCase):
    def test_average_singleton_list(self):
        self.assertEqual( 5, average([5]) )

    def test_list_with_many_values(self):
        # test average of many values

    def test_average_of_empty_list(self):
        # test that average([]) throws exception
```

# Run Your Tests

```
cmd> python -m unittest test_average.py
```

```
FFF
```

```
-----
```

```
Ran 3 tests in 0.001s
```

```
FAILED (failures=3)
```

The test should all fail.

## Exercise: Write `average` (1st)

Write the code for `average()` so it passes all tests.

Do you *feel* any difference while coding?

# Test involving Floating Point

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

This is normal.

To test a result which may have **rounding error**, use `assertAlmostEqual`

```
def test_with_limited_precision( self ):  
    self.assertAlmostEqual(  
        2.33333333, average([1,2,4]), places=8 )
```

# A Stack Example

- ❑ A **Stack** implements common stack data structure.
- ❑ You can `push()`, `pop()`, and `peek()` elements.
- ❑ Throws **StackException** if you do something stupid.

Stack
<pre>+ Stack( capacity ) + capacity( ): int + size( ): int + isEmpty( ): boolean + isFull( ): boolean + push( T ): void + pop( ): T + peek( ): T</pre>



# Stack Tests all Need a Stack

In each test we need to create a new stack (so the tests are independent).

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() )
```

# Use tearDown() to clean up after tests

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

```
class FileTest(unittest.TestCase):  
    def setUp(self):  
        # open file containing test data  
        self.file = open("testdata", "r")  
  
    def tearDown(self):  
        self.file.close()
```

# 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)
```

# Testing is Not So Easy!

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

Real tests are much 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>

Real Python - good explanation & how to run unit tests in IDE

<https://realpython.com/python-testing/>

Python Hitchhiker's Guide to Testing

<https://docs.python-guide.org/writing/tests/>

- Examples of common testing tools

*Python Cookbook, Chapter 14*

How to test many common situations, including I/O