Unit Testing in Python

James Brucker
Python Testing Frameworks

We will cover these two:

- **unittest** - part of the Python library, similar to JUnit
- **doctest** - test by example, part of the Python library
Other Testing Frameworks

Pytest - simple "assert" syntax, easy-to-read output
- Pytest can also run unittest style tests

Mock objects - create "fake" external components
- also called "test doubles"

Good Overview

*Getting Started with Testing in Python* (RealPython)

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

```python
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() )
```

expected result: should be True

actual result: 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. Let your IDE run the tests.
2. Use a test script or **build tool**.
3. Add a "main" block to your Test class...

```python
import unittest

class TestBuiltins(unittest.TestCase):
    """Test some python built-in methods"""
    def test_len(self):
        self.assertEqual(5, len("hello"))

if __name__ == "__main__":
    unittest.main()
```
Exercise: Try it Yourself

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

```python
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
```
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?

```python
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 (specific):

```python
assertEqual( 5, math.sqrt(25))
assertGreater( math.pi, 3.14159)
assertNotIn('a', ['yes','no','maybe'])
```

**Don't** write this:

```python
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*.

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

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

# delta = allowed difference in values
self.assertAlmostEqual( 0.33333, 1.0/3.0, delta=0.5e-4)
```
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!")
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:

```python
def test_sqrt_of_negative(self):
    with self.assertRaises(ValueError):
        result = math.sqrt(-1)  # hidden
    with self.assertRaises(ValueError):
        math.log(-4)
```
Can we do this?

`assertRaises` with extra argument:

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

This *doesn't work*.

A `ValueError` exception is thrown (test fails).
Which Operation is Done 1st, 2nd, ..?

```python
print("sqrt 5 + 1 is", 1 + math.sqrt(5))
```
Which operation is done first?

```python
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.
What is a `callable`?

Something that you can call. :-)  
Example: a function, a lambda expression
Use a callable in `assertRaises`

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

*args passed to the callable
What to Name Your Tests?

1. **Test methods** begin with `test_` and use **snake case**.
   
   ```python
   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**.
   
   ```python
   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 (used by Django) **won't** discover 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 code. Just return 0:

```python
def average(lst):
    """Return average of a list of numbers""
    return 0
```
What is Test Driven Development

Write the tests before you write the code to test.

1. Write a test or a small number of tests.
2. Run tests. They all fail.
3. Write just enough code to make the tests pass...but not more code.
Repeat.
TDD: 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.

Use it as an example.
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
TDD: Run Your Tests

```
cmd> python -m unittest test_average.py
FFF
Ran 3 tests in 0.001s
FAILED (failures=3)
```

The tests should all fail.
TDD: Write `average(lst)`

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

Do you *feel* any difference while coding?
A **Stack** implements common stack data structure.

You can push(), pop(), and peek() elements.

Throws **StackException** if you do something stupid.

<table>
<thead>
<tr>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ Stack( capacity )</td>
</tr>
<tr>
<td>+ capacity( ) : int</td>
</tr>
<tr>
<td>+ size( ) : int</td>
</tr>
<tr>
<td>+ isEmpty( ) : boolean</td>
</tr>
<tr>
<td>+ isFull( ) : boolean</td>
</tr>
<tr>
<td>+ push( T ) : void</td>
</tr>
<tr>
<td>+ pop( ) : T</td>
</tr>
<tr>
<td>+ peek( ) : T</td>
</tr>
</tbody>
</table>
Stack Tests **all** Need a Stack

In each test we need to create a new stack.

That's a lot of duplicate code.

How to eliminate duplicate code?

```python
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.

```python
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 test

tearDown() is called after each test. It's 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):
        self.file.close()
**Doctest**

Include runnable code inside Python DocStrings.

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

```python
def average(lst):
    """Return the average of a list of numbers."
    return sum(lst)/len(lst)
```

doctest comments
Running Doctest

Run doctest using command line:

```sh
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:

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

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

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

Video shows how to use unittest

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

Extensive List of Testing Tools for all kinds of testing

https://wiki.python.org/moin/
  PythonTestingToolsTaxonomy