Unit Testing

Software Engineering for Scientists

https://github.com/swe4s/lectures/tree/master/src/unit_test

Unit tests are functions that test the individual components of your code, which is in contrast to **functional tests** that consider the full execution paths that users are likely to encounter. Unit tests help us deal with the complexity of large programs by ensuring the correctness of the individual pieces and can uncover bugs as we make improvements and add features. While there is no set rule on what should be covered by a unit test, in general, every (non-trivial) function should have a unit test, and functions with several different behaviors should have a unit test for each behavior. It is also good practice to create a unit test when fixing a bug. Making your code easy to test will require your code to be modular (i.e., many small methods).

Unit tests with `unittest` (https://docs.python.org/3.6/library/unittest.html)

`math_lib.py`

```python
def add(x, y):
      return x + y

def sub(x, y):
      return x - y
```

`test_math_lib.py`

```python
import unittest
import math_lib

class TestCalc(unittest.TestCase):
    def test_add(self):
        self.assertEqual(math_lib.add(10,-5), 5)

    def test_sub(self):
        self.assertEqual(math_lib.sub(10,-5), 15)
```

```
$ python -m unittest test_math_lib.py
..
----------------------------------------------------------------------
Ran 2 test in 0.000s

OK
```

Each `.` after the test is executed corresponds to a successful test. An `F` indicates a failed test. For example, if `test_sub` had an expected value of `14`, then the output would be

```
$ python -m unittest x.py
.F
```

```
======================================================================
FAIL: test_sub (x.TestCalc)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/Users/rl/Box
Sync/Research/teaching/swe4s/lectures.git/unit_test/x.py", line 9, in
test_sub
    self.assertEqual(math_lib.sub(10,-5), 14)
AssertionError: 15 != 14

----------------------------------------------------------------------
Ran 2 tests in 0.000s
```

Each test method counts as one test. Each test method can have many `self.assert` calls, but will still count as just one test. If all asserts pass, then the test passes. If any assertion fails, then the test fails. Because of this, you must keep your tests small and independent of any other test. Each test should be specific to one behavior, otherwise, it will be difficult to determine what has failed.

Notes about `unittest`

- The class `TestCalc` could be named anything. Including the `Test` and `Calc` is convention so we know it is a test and what it is testing
- The testing class must inherit `unittest.TestCase`
- Test methods must begin with `test_`. Class methods can have other names, but they will not be counted as tests.
- Other asserts:

`assertEqual(a, b)`	`a == b`
`assertNotEqual(a, b)`	`a != b`
`assertTrue(x)`	`bool(x) is True`
`assertFalse(x)`	`bool(x) is False`
`assertIs(a, b)`	`a is b`
`assertIsNot(a, b)`	`a is not b`
`assertIsNone(x)`	`x is None`
`assertIsNotNone(x)`	`x is not None`
`assertIn(a, b)`	`a in b`
`assertNotIn(a, b)`	`a not in b`

Making tests easier to run

test_math_lib.py

```python
import unittest
import math_lib

class TestCalc(unittest.TestCase):
    def test_add(self):
        self.assertEqual(math_lib.add(10,-5), -15)

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

```
$ python test_math_lib.py
.
----------------------------------------------------------------------
Ran 1 test in 0.000s

OK
```

Testing error modes

In addition to testing for proper behavior, it is also important to test error handling and in Python, this means making sure that the right exceptions are generated when input parameters force the function into a bad state. There are two ways to test exceptions and both are slightly strange. One uses a context manager and has a `with` statement,

```
with self.assertRaises(ZeroDivisionError):
    math_lib.div(10, 0)
```

and the other does not but has a non-standard function call.

```
self.assertRaises(ZeroDivisionError, math_lib.div, 10, 0)
```

Either is fine, but I prefer the second.

Reducing repetitive code for test setup

In many cases, we want to run several tests against the same file or data structure. Since the tests are independent, this can lead to redundant data structure setup. For example, the methods `file_add` and `file_prod` both take a file of numbers as input and to test these functions we need to create test files. One option is to create and delete these files in each test, which is difficult to maintain.

```python
    def test_file_add_no_setup(self):
        test_file_name = 'test_file.txt'
        f = open(test_file_name, 'w')

        direct_sum = 0

        for i in range(100):
            rand_int = random.randint(1, 100)
            direct_sum += rand_int
            f.write(str(rand_int) + '\n')
        f.close()

        file_sum = math_lib.file_add(test_file_name)

        self.assertEqual(file_sum, direct_sum)

        os.remove(test_file_name)

    def test_file_prod_no_setup(self):
        test_file_name = 'test_file.txt'
        f = open(test_file_name, 'w')

        direct_prod = 1

        for i in range(100):
            rand_int = random.randint(1, 100)
            direct_prod += rand_int
            f.write(str(rand_int) + '\n')
        f.close()

        file_prod = math_lib.file_prod(test_file_name)

        self.assertEqual(file_prod, direct_prod)

        os.remove(test_file_name)
```

Alternatively, you can define a `setUp()` and `tearDown()` methods that will run before and after, respectively, every test in the test. To use these methods, you must also define instance variables to hold the relevant values.

```
    def setUp(self):
        self.test_file_name = 'setup_test_file.txt'
        f = open(self.test_file_name, 'w')

        self.direct_sum = 1
        self.direct_prod = 1

        for i in range(100):
            rand_int = random.randint(1, 100)
            self.direct_prod *= rand_int
            self.direct_sum += rand_int
            f.write(str(rand_int) + '\n')
        f.close()

    def tearDown(self):
        os.remove(self.test_file_name)
```

Now tests can access the instance variables `test_file_name`, `direct_sum`, and `direct_prob`. Since `setUp` and `tearDown` run before and after each test, these variables will hold different values (`test_file_name` will have the name string, but the contents of the file will be different).

```
    def test_file_add_setup(self):
        file_sum = math_lib.file_add(self.test_file_name)
        self.assertEqual(file_sum, self.direct_sum)

    def test_file_prod_setup(self):
        file_prod = math_lib.file_prod(self.test_file_name)
        self.assertEqual(file_prod, self.direct_prod)
```

It is important to note that running `setUp()` and `tearDown()` before and after every test could add an unreasonable amount of overhead. If this is the case, the class methods `setUpClass` and `tearDownClass` can be defined and will run once before every test and once after every test, respectively. The drawback is that the values of those variables and files will not change between tests.

```
    @classmethod
    def setUpClass(cls):
        cls.class_test_file_name = 'class_setup_test_file.txt'
        f = open(cls.class_test_file_name, 'w')

        cls.class_direct_sum = 1
        cls.class_direct_prod = 1

        for i in range(100):
            rand_int = random.randint(1, 100)
            cls.class_direct_prod *= rand_int
            cls.class_direct_sum += rand_int
            f.write(str(rand_int) + '\n')
        f.close()

    @classmethod
    def tearDownClass(cls):
        os.remove(cls.class_test_file_name)

    def test_file_add_class_setup(self):
        file_sum = math_lib.file_add(self.class_test_file_name)
        self.assertEqual(file_sum, self.class_direct_sum)

    def test_file_prod_class_setup(self):
        file_prod = math_lib.file_prod(self.class_test_file_name)
        self.assertEqual(file_prod, self.class_direct_prod
```

Inheritance and Creating Multiple Test Classes

When we construct our test class, we build it as a subclass of `unittest.TestCase`, this is done by placing the parent class, `unittest.TestCase` as an input argument to the child class `TestCalc`, as shown:

`test_libraries.py`

```
import unittest
import math_lib

class TestCalc(unittest.TestCase):
    def test_add(self):
        self.assertEqual(calc.add(10,-5), 5)

    def test_sub(self):
        self.assertEqual(calc.sub(10,-5), 15)
```

Then we run the child class, TestCalc, indirectly by running the main() constructor function of the parent class, unittest, inside our “main” function (the function `if __name__ == ‘__main__’:`)

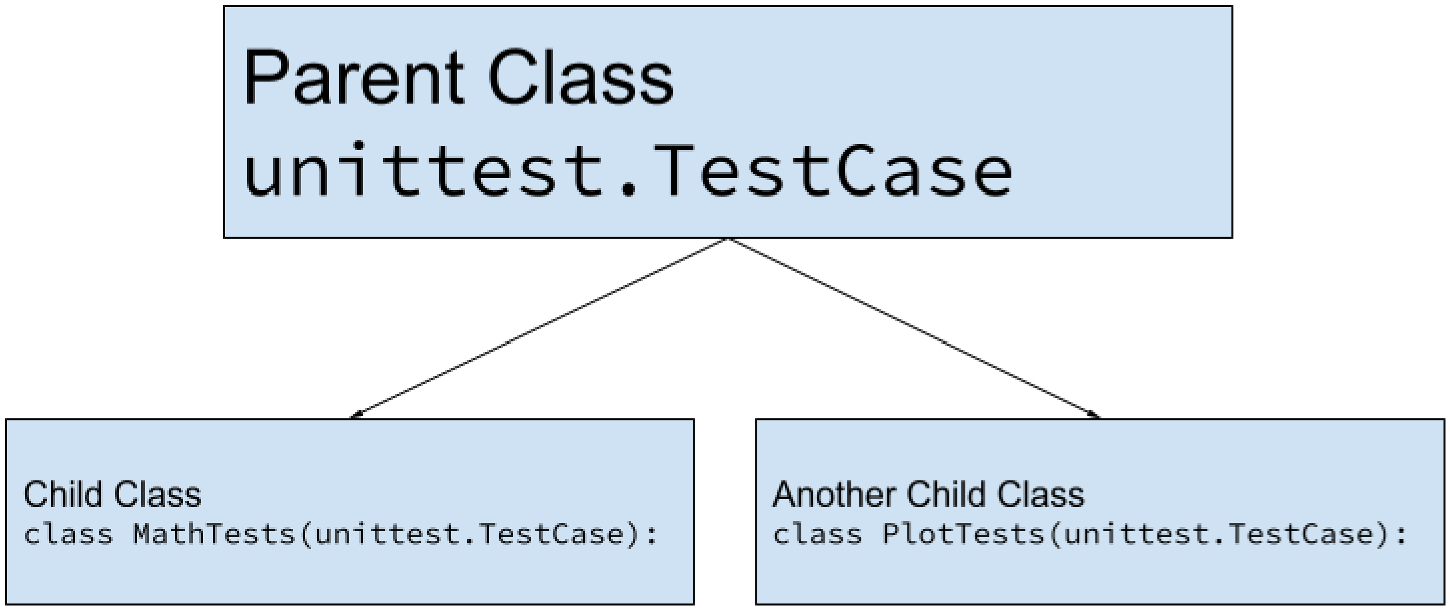
`test_libraries.py`

```python
import unittest
import math_lib

class TestCalc(unittest.TestCase):
    def test_add(self):
        self.assertEqual(calc.add(10,-5), 5)

    def test_sub(self):
        self.assertEqual(calc.sub(10,-5), 15)

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

Suppose we have made multiple modules which are used together (`math_lib.py` and `plots_lib.py`) and we want to test them all together.

We can do this cleanly by creating multiple children classes in the same `test_libraries.py` file.

Recall the structure of parent-child subclasses. The children inherit all the functionality from the parent, but the two (or more) independent children classes, do not know about each other.

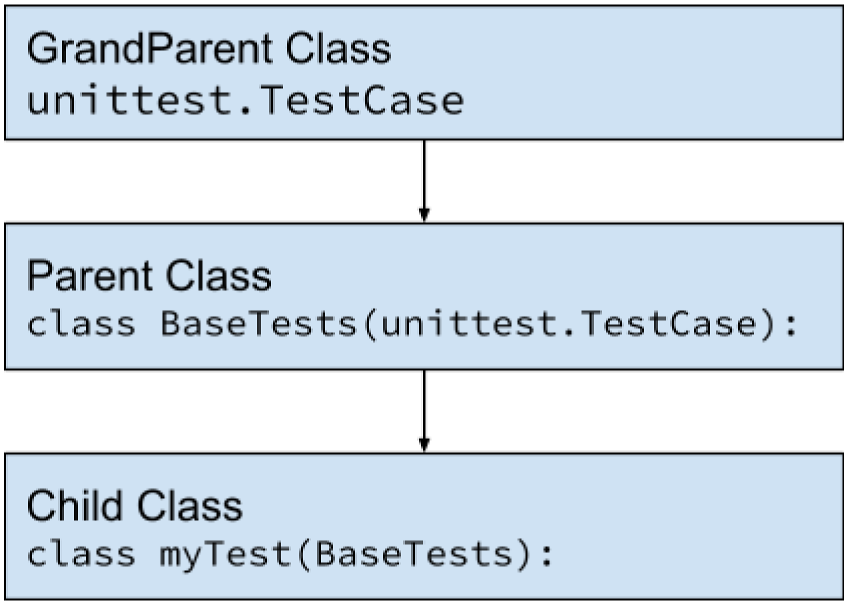

Parent Class
unittest.TestCase
Child Class
class MathTests(unittest.TestCase):
Another Child Class
class PlotTests(unittest.TestCase):

Luckily, the unittest constructor, builds and tests all the children classes so you can create two subclasses `TestCalc` and `TestPlot`, and run them with the single `unittest.main()` function:

test_libraries.py

```
import unittest
import os
import math_lib
import plot_lib


class TestCalc(unittest.TestCase):
    def test_add(self):
        self.assertEqual(calc.add(10,-5), 5)

    def test_sub(self):
        self.assertEqual(calc.sub(10,-5), 15)

class TestPlot(unittest.TestCase):
    def test_not_unique_file_name(self):
        test_file_name = ‘newPlot’
        file = open(test_file_name, 'w')
        file.write("1, 2, 3")
        file.close()
        self.assertEqual(hist.name_not_unique(test_file_name),
                         os.path.exists(test_file_name))


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

However, suppose you want to run *some* of the same tests in both TestCalc and TestPlot, you can easily do this *without* copy and pasting by building subsubclasses (grandchildren)



In order to make sure `unittest` properly calls all the children classes, we want to create an extra “empty” class wrapper `BaseTestCases` which contains our `BaseTests`, and build the grandchildren `TestCalc` and `TestPlot` on top of `BaseTest`. This is just to make it easier for the way the constructor in the `unittest` module is called.

```
import unittest
import os
import math_lib
import plot_lib

class BaseTestCases:
    class BaseTest(unittest.TestCase):
        def common_test(self):
            x = 4
            self.assertEqual(x, 4)

class TestCalc(BaseTestCases.BaseTest):
    def test_add(self):
        self.assertEqual(calc.add(10,-5), 5)

    def test_sub(self):
        self.assertEqual(calc.sub(10,-5), 15)

class TestPlot(BaseTestCases.BaseTest):
    def test_not_unique_file_name(self):
        test_file_name = ‘newPlot’
        file = open(test_file_name, 'w')
        file.write("1, 2, 3")
        file.close()
        self.assertEqual(hist.name_not_unique(test_file_name),
                         os.path.exists(test_file_name))


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

This will call the following sequence:

```
BaseTest.common_test
TestCalc.test_add
TestCast.test_sub
TestPlot.test_not_unique_file_name
```