# Testing

## Testing: Introduction

Code without tests is like driving without seatbelts.

All code is subject to errors -- not the SyntaxErrors and TypeErrors encountered during development, but errors related to unexpected data anomalies or user input, or the unforeseen effects of functions run in untested combinations.

Many developers say they won't take a software package seriously unless it comes with tests. **Unit testing** is the front line of the effort to ensure code quality.

**testing: a brief rundown**
Unit testing is the most basic form and the one we will focus on in this unit. Other styles of testing:

| |
|---|
| **unit test**: testing individual units (function/method) |
| **integration test**: testing multiple units together |
| **regression test**: testing to see if changes have introduced errors |
| **end-to-end test**: testing the entire program |

## Unit Testing

"Unit" refers to a function. Unit testing calls individual functions and validates the output or result of each.

The most easily tested scripts are made up of small functions that can be called and validated in isolation. Therefore "pure functions" (functions that do not refer or change "external state" -- i.e., global variables) are best for testing.

**Testing for success, testing for failure**
A **unit test script** performs test by importing the script to be tested and calling its functions with varying arguments, including ones intended to cause an error. Basically, we are hammering the code as many ways as we can to make sure it succeeds properly *and fails properly*.

**Test-driven development**
As we develop our code, we can write tests simultaneously and run them periodically as we develop. This way we can know that further changes and additions are not interfering with anything we have done previously. Any time in the process we can run the testing program and it will run all tests.

In fact commonly accepted wisdom supports writing tests before writing code! The test is written with the function in mind: after seeing that the tests fail, we write a function to satisfy the tests. This called *test-driven development*.

## The *assert* statement

**assert** raises an **AssertionError** exception if the test returns **False**

```
assert 5 == 5        # no output

assert 5 == 10       # AssertionError raised
```

We can incorporate this facility in a simple testing program:

**program to be tested: "myprogram.py"**

```
import sys

def doubleit(x):
    var = x * 2
    return var

if __name__ == '__main__':
    input_val = sys.argv[1]
    doubled_val = doubleit(input_val)

    print "the value of {0} is {1}".format(input_val, doubled_val)
```

**testing program: "test_myprogram.py"**

```
import myprogram

def test_doubleit_value():
    assert myprogram.doubleit(10) == 20
```

If **doubleit()** didn't correctly return **20** with an argument of **10**, the **assert** would raise an **AssertionError**. So even with this basic approach (without a testing module like **pyttest** or **unittest**), we can do testing with **assert**.

## pytest Basics

All programs named ***something*_test.py** that have functions named **test_*something*()** will be noticed by **pytest** and run automatically when we run the pytest script **py.test**.

**running py.test from the command line**

```
$ py.test
=================================== test session starts ====================================
platform darwin -- Python 2.7.10 -- py-1.4.27 -- pytest-2.7.1
rootdir: /Users/dblaikie/testpytest, inifile:
collected 1 items

test_myprogram.py .

================================= 1 passed in 0.01 seconds =================================
```

**noticing failures**

```
def doubleit(x):
    var = x * 2
    return x      # oops, returned the original value rather than the doubled value
```

Having incorporated an error, run **py.test** again:

```
$ py.test
=================================== test session starts ====================================
platform darwin -- Python 2.7.10 -- py-1.4.27 -- pytest-2.7.1
rootdir: /Users/dblaikie/testpytest, inifile:
collected 1 items

test_myprogram.py F

========================================= FAILURES =========================================
___________________________________ test_doubleit_value ____________________________________

    def test_doubleit_value():
>       assert myprogram.doubleit(10) == 20
E       assert 10 == 20
E        +  where 10 = (10)
E        +    where  = myprogram.doubleit

test_myprogram.py:7: AssertionError
================================= 1 failed in 0.01 seconds =================================
```

## Testing the expected raising of an exception

Many of our tests will deliberately pass bad input and test to see that an appropriate exception is raised.

```
import sys

def doubleit(x):
    if not isinstance(x, (int, float)):           # make sure the arg is the right type
        raise TypeError, 'must be int or float'   # if not, raise a TypeError
    var = x * 2
    return var

if __name__ == '__main__':
    input_val = sys.argv[1]
    doubled_val = doubleit(input_val)

    print "the value of {0} is {1}".format(input_val, doubled_val)
```

Note that without type testing, the function could work, but incorrectly (for example if a string or list were passed instead of an integer).

To verify that this error condition is correctly raised, we can use **with pytest.raises(TypeError)**.

```
import myprogram
import pytest

def test_doubleit_value():
    assert myprogram.doubleit(10) == 20

def test_doubleit_type():
    with pytest.raises(TypeError):
        myprogram.doubleit('hello')
```

**with** is the same context manager we have used with **open()**:  it can also be used to detect when an exception occured inside the **with** block.

## Grouping tests into a class

We can organize related tests into a class, which can also include setup and teardown routines that are run automatically (discussed next).

```
""" test_myprogram.py -- test functions in a testing class """

import myprogram
import pytest

class TestDoubleit(object):

    def test_doubleit_value(self):
        assert myprogram.doubleit(10) == 20

    def test_doubleit_type(self):
        with pytest.raises(TypeError):
            myprogram.doubleit('hello')
```

So now the same rule applies for how py.test looks for tests -- if the class begins with the word **Test**, pytest will treat it as a testing class.

## Mock data: setup and teardown

Tests should not be run on "live" data; instead, it should be simulated, or "mocked" to provide the data the test needs.

```
""" myprogram.py -- makework functions for the purposes of demonstrating testing """

import sys

def doubleit(x):
    """ double a number argument, return doubled value """
    if not isinstance(x, (int, float)):
        raise TypeError, 'arg to doublit() must be int or float'
    var = x * 2
    return var

def doublelines(filename):
    """open a file of numbers, double each line, write each line to a new file"""
    with open(filename) as fh:
        newlist = []
        for line in fh:                   # file is assumed to have one number on each line
            floatval = float(line)
            doubleval = doubleit(floatval)
            newlist.append(str(doubleval))
    with open(filename, 'w') as fh:
        fh.write('\n'.join(newlist))

if __name__ == '__main__':
    input_val = sys.argv[1]
    doubled_val = doubleit(input_val)

    print "the value of {0} is {1}".format(input_val, doubled_val)
```

For this demo I've invented a rather arbitrary example to combine an external file with the **doubleit()** routine: **doublelines()** opens and reads a file, and for each line in the file, doubles the value, writing each value as a separate line to a new file (supplied to **doublelines()**).

```
""" test_myprogram.py --

import myprogram
import os
import pytest
import shutil

class TestDoubleit(object):

    numbers_file_template = 'testnums_template.txt'  # template for test file (stays the same)
    numbers_file_testor = 'testnums.txt'             # filename used for testing
                                                     # (changed during testing)

    def setup_class(self):
        shutil.copy(TestDoubleit.numbers_file_template, TestDoubleit.numbers_file_testor)

    def teardown_class(self):
        os.remove(TestDoubleit.numbers_file_testor)

    def test_doublelines(self):
        myprogram.doublelines(TestDoubleit.numbers_file_testor)
        old_vals = [ float(line) for line in open(TestDoubleit.numbers_file_template) ]
        new_vals = [ float(line) for line in open(TestDoubleit.numbers_file_testor) ]
        for old_val, new_val in zip(old_vals, new_vals):
            assert float(new_val) == float(old_val) * 2

    def test_doubleit_value(self):
        assert myprogram.doubleit(10) == 20

    def test_doubleit_type(self):
        with pytest.raises(TypeError):
            myprogram.doubleit('hello')
```

**setup_class** and **teardown_class** run automatically.  As you can see, they prepare a dummy file and when the testing is over, delete it.  In between, tests are run in order based on the function names.