Getting Started Testing

Ned Batchelder
@nedbat

http://bit.ly/pytest0

Goals

- Show you a way to test
- Remove mystery

Why test?

- Know if your code works
- Save time
- Better code
- Remove fear
- "Debugging is hard, testing is easy"

I AM BAD!

AND I SHOULD FEEL BAD

bit.ly/pytest0

@nedbat

Yeah, it's hard

- A lot of work
- People (you) won't want to
- But: it pays off

Chaos!

Roadmap

- Growing tests
- unittest
- Mocks

First principles

Growing tests

Stock portfolio class

```python
# portfolio1.py

class Portfolio(object):
    """A simple stock portfolio"""
    def __init__(self):
        # stocks is a list of lists:
        #   [[name, shares, price], ...]
        self.stocks = []

    def buy(self, name, shares, price):
        """Buy `name`: `shares` shares at `price`."""
        self.stocks.append([name, shares, price])

    def cost(self):
        """What was the total cost of this portfolio?"""
        amt = 0.0
        for name, shares, price in self.stocks:
            amt += shares * price
        return amt
```

First test: interactive

```
>>> p = Portfolio()
>>> p.cost()
0.0

>>> p.buy("IBM", 100, 176.48)
>>> p.cost()
17648.0

>>> p.buy("HPQ", 100, 36.15)
>>> p.cost()
21263.0
```

- ✓ Good: testing the code
- ✘ Bad: not repeatable
- ✘ Bad: labor intensive
- ✘ Bad: is it right?

Second test: standalone

```
# porttest1.py
from portfolio1 import Portfolio

p = Portfolio()
print "Empty portfolio cost: %s" % p.cost()
p.buy("IBM", 100, 176.48)
print "With 100 IBM @ 176.48: %s" % p.cost()
p.buy("HPQ", 100, 36.15)
print "With 100 HPQ @ 36.15: %s" % p.cost()
```

```
$ python porttest1.py
Empty portfolio cost: 0.0
With 100 IBM @ 176.48: 17648.0
With 100 HPQ @ 36.15: 21263.0
```

- ✓ Good: testing the code
- ✓ Better: repeatable
- ✓ Better: low effort
- ✘ Bad: is it right?

Third test: expected results

```
4 p = Portfolio()
5 print "Empty portfolio cost: %s, should be 0.0" % p.cost()
6 p.buy("IBM", 100, 176.48)
7 print "With 100 IBM @ 176.48: %s, should be 17648.0" % p.cost()
8 p.buy("HPQ", 100, 36.15)
9 print "With 100 HPQ @ 36.15: %s, should be 21263.0" % p.cost()
```

```
1 $ python porttest2.py
2 Empty portfolio cost: 0.0, should be 0.0
3 With 100 IBM @ 176.48: 17648.0, should be 17648.0
4 With 100 HPQ @ 36.15: 21263.0, should be 21263.0
```

✓ Good: repeatable with low effort

✓ Better: explicit expected results

✘ Bad: have to check the results yourself

Fourth test: check results automatically

```
4  p = Portfolio()
5  print "Empty portfolio cost: %s, should be 0.0" % p.cost()
6  assert p.cost() == 0.0
7  p.buy("IBM", 100, 176.48)
8  print "With 100 IBM @ 176.48: %s, should be 17648.0" % p.cost()
9  assert p.cost() == 17648.0
10 p.buy("HPQ", 100, 36.15)
11 print "With 100 HPQ @ 36.15: %s, should be 21263.0" % p.cost()
12 assert p.cost() == 21263.0
```

```
1  $ python porttest3.py
2  Empty portfolio cost: 0.0, should be 0.0
3  With 100 IBM @ 176.48: 17648.0, should be 17648.0
4  With 100 HPQ @ 36.15: 21263.0, should be 21263.0
```

- ✓ Good: repeatable with low effort
- ✓ Good: explicit expected results
- ✓ Good: results checked automatically

Fourth test: what failure looks like

```
$ python porttest3_broken.py
Empty portfolio cost: 0.0, should be 0.0
With 100 IBM @ 176.48: 17648.0, should be 17600.0
Traceback (most recent call last):
  File "porttest3_broken.py", line 9, in <module>
    assert p.cost() == 17600.0
AssertionError
```

- ✓ Good: repeatable with low effort
- ✓ Good: expected results checked automatically
- ✓ OK: visible failure visible, but cluttered output
- ✘ Bad: failure stops tests

Getting complicated!

- Tests will grow
- Real programs
- Real engineering
- Handle common issues in standard ways

Good tests

- Automated
- Fast
- Reliable
- Informative
- Focused

unittest

Writing tests

unittest

- Python standard library
- Infrastructure for well-structured tests
- Patterned on xUnit

A simple unit test

```python
# test_port1.py

import unittest
from portfolio1 import Portfolio

class PortfolioTest(unittest.TestCase):
    def test_buy_one_stock(self):
        p = Portfolio()
        p.buy("IBM", 100, 176.48)
        assert p.cost() == 17648.0
```

```
$ python -m unittest test_port1
.
----------------------------------------------------------------
Ran 1 test in 0.000s

OK
```

Under the covers

```
# unittest runs the tests as if I had written:
testcase = PortfolioTest()
try:
    testcase.test_buy_one_stock()
except AssertionError:
    [record failure]
else:
    [record success]
```

Add more tests

```
6  class PortfolioTest(unittest.TestCase):
7      def test_empty(self):
8          p = Portfolio()
9          assert p.cost() == 0.0
10
11     def test_buy_one_stock(self):
12         p = Portfolio()
13         p.buy("IBM", 100, 176.48)
14         assert p.cost() == 17648.0
15
16     def test_buy_two_stocks(self):
17         p = Portfolio()
18         p.buy("IBM", 100, 176.48)
19         p.buy("HPQ", 100, 36.15)
20         assert p.cost() == 21263.0
```

```
1  $ python -m unittest test_port2
2  ...
3  ----------------------------------------------------------------
4  Ran 3 tests in 0.000s
5
6  OK
```

- A dot for every passed test

Under the covers

```
# unittest runs the tests as if I had written:
testcase = PortfolioTest()
try:
    testcase.test_empty()
except AssertionError:
    [record failure]
else:
    [record success]

testcase = PortfolioTest()
try:
    testcase.test_buy_one_stock()
except AssertionError:
    [record failure]
else:
    [record success]

testcase = PortfolioTest()
try:
    testcase.test_buy_two_stocks()
except AssertionError:
    [record failure]
else:
    [record success]
```

Test isolation

- Every test gets a new test object
- Tests can't affect each other
- Failure doesn't stop next tests

What failure looks like

```
$ python -m unittest test_port2_broken
F..
======================================================================
FAIL: test_buy_one_stock (test_port2_broken.PortfolioTest)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "test_port2_broken.py", line 14, in test_buy_one_stock
    assert p.cost() == 17648.0
AssertionError

----------------------------------------------------------------------
Ran 3 tests in 0.000s

FAILED (failures=1)
```

✓ Better: failed test didn't stop others

✘ Bad: what value was returned?

unittest assert helpers

self.assertEqual(x, y) instead of assert x == y

```
11 def test_buy_one_stock(self):
12     p = Portfolio()
13     p.buy("IBM", 100, 176.48)
14     self.assertEqual(p.cost(), 17648.0)
```

```
1  $ python -m unittest test_port3_broken
2  F..
3  ======================================================================
4  FAIL: test_buy_one_stock (test_port3_broken.PortfolioTest)
5  ----------------------------------------------------------------------
6  Traceback (most recent call last):
7    File "test_port3_broken.py", line 14, in test_buy_one_stock
8      self.assertEqual(p.cost(), 17648.0)
9  AssertionError: 17600.0 != 17648.0
10
11 ----------------------------------------------------------------------
12 Ran 3 tests in 0.000s
13
14 FAILED (failures=1)
```

Lots of assert helpers

```
assertEqual(first, second)
assertNotEqual(first, second)
assertTrue(expr)
assertFalse(expr)
assertIn(first, second)
assertNotIn(first, second)
assertIs(first, second)
assertIsNot(first, second)
assertAlmostEqual(first, second)
assertNotAlmostEqual(first, second)
assertGreater(first, second)
assertLess(first, second)
assertRegexpMatches(text, regexp)
assertRaises(exc_class, func, ...)
assertSequenceEqual(seq1, seq2)
assertItemsEqual(seq1, seq2)

.. etc ..
```

Pro tip: your own base class

```
6  class PortfolioTestCase(unittest.TestCase):
7      """Base class for all Portfolio tests."""
8
9      def assertCostEqual(self, p, cost):
10         """Assert that `p`'s cost is equal to `cost`."""
11         self.assertEqual(p.cost(), cost)
12
13
14 class PortfolioTest(PortfolioTestCase):
15     def test_empty(self):
16         p = Portfolio()
17         self.assertCostEqual(p, 0.0)
18
19     def test_buy_one_stock(self):
20         p = Portfolio()
21         p.buy("IBM", 100, 176.48)
22         self.assertCostEqual(p, 17648.0)
23
24     def test_buy_two_stocks(self):
25         p = Portfolio()
26         p.buy("IBM", 100, 176.48)
27         p.buy("HPQ", 100, 36.15)
28         self.assertCostEqual(p, 21263.0)
```

Third possible outcome: E

Test raises an exception

```
$ python -m unittest test_port3_broken2
E..
======================================================================
ERROR: test_buy_one_stock (test_port3_broken2.PortfolioTest)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "test_port3_broken2.py", line 13, in test_buy_one_stock
    p.buyXX("IBM", 100, 176.48)
AttributeError: 'Portfolio' object has no attribute 'buyXX'

----------------------------------------------------------------------
Ran 3 tests in 0.000s

FAILED (errors=1)
```

Can't call the function

```
22 def test_bad_input(self):
23     p = Portfolio()
24     p.buy("IBM")
```

```
1  $ python -m unittest test_port4_broken
2  E...
3  ======================================================================
4  ERROR: test_bad_input (test_port4_broken.PortfolioTest)
5  ----------------------------------------------------------------------
6  Traceback (most recent call last):
7    File "test_port4_broken.py", line 24, in test_bad_input
8      p.buy("IBM")
9  TypeError: buy() takes exactly 4 arguments (2 given)
10
11 ----------------------------------------------------------------------
12 Ran 4 tests in 0.001s
13
14 FAILED (errors=1)
```

assertRaises

```
22 def test_bad_input(self):
23     p = Portfolio()
24     with self.assertRaises(TypeError):
25         p.buy("IBM")
```

```
1 $ python -m unittest test_port4
2 ....
3 ----------------------------------------------------------------------
4 Ran 4 tests in 0.000s
5
6 OK
```

Portfolio: .sell()

```
21 def sell(self, name, shares):
22     """Sell some shares."""
23     for holding in self.stocks:
24         if holding[0] == name:
25             if holding[1] < shares:
26                 raise ValueError("Not enough shares")
27             holding[1] -= shares
28             break
29     else:
30         raise ValueError("You don't own that stock")
```

Testing sell()

```
34 class PortfolioSellTest(PortfolioTestCase):
35     def test_sell(self):
36         p = Portfolio()
37         p.buy("MSFT", 100, 27.0)
38         p.buy("DELL", 100, 17.0)
39         p.buy("ORCL", 100, 34.0)
40         p.sell("MSFT", 50)
41         self.assertCostEqual(p, 6450)
42 
43     def test_not_enough(self):
44         p = Portfolio()             # Didn't I just do this?
45         p.buy("MSFT", 100, 27.0)
46         p.buy("DELL", 100, 17.0)
47         p.buy("ORCL", 100, 34.0)
48         with self.assertRaises(ValueError):
49             p.sell("MSFT", 200)
50 
51     def test_dont_own_it(self):
52         p = Portfolio()             # What, again!?!?
53         p.buy("MSFT", 100, 27.0)
54         p.buy("DELL", 100, 17.0)
55         p.buy("ORCL", 100, 34.0)
56         with self.assertRaises(ValueError):
57             p.sell("IBM", 1)
```

Setting up a test

```
34 class PortfolioSellTest(PortfolioTestCase):
35     # Invoked before each test method
36     def setUp(self):
37         self.p = Portfolio()
38         self.p.buy("MSFT", 100, 27.0)
39         self.p.buy("DELL", 100, 17.0)
40         self.p.buy("ORCL", 100, 34.0)
41 
42     def test_sell(self):
43         self.p.sell("MSFT", 50)
44         self.assertCostEqual(self.p, 6450)
45 
46     def test_not_enough(self):
47         with self.assertRaises(ValueError):
48             self.p.sell("MSFT", 200)
49 
50     def test_dont_own_it(self):
51         with self.assertRaises(ValueError):
52             self.p.sell("IBM", 1)
```

Under the covers

```
testcase = PortfolioTest()
try:
    testcase.setUp()
except:
    [record error]
else:
    try:
        testcase.test_method()
    except AssertionError:
        [record failure]
    except:
        [record error]
    else:
        [record success]
    finally:
        try:
            testcase.tearDown()
        except:
            [record error]
```

setUp and tearDown: isolation!

- Establish context
- Common pre- or post- work
- Isolation, even with failures
- Also: "fixtures"

Tests are real code!

- Helper functions, classes, etc.
- Can become significant
- Might need tests!

Mocks

Focusing tests

Testing small amounts of code

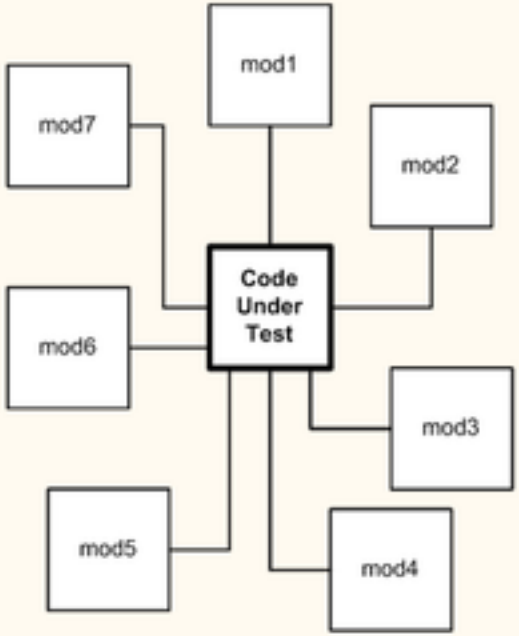
- Systems are built in layers
- Components depend on each other
- How to test just one component?

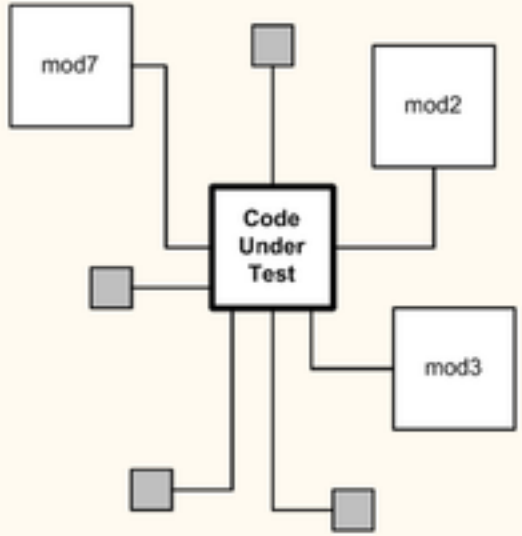
Dependencies are bad

- More suspect code in each test
- Slow components
- Unpredictable components

Test Doubles

- Replace a component's dependencies
- Focus on one component





Portfolio: Real-time data!

```
51  def current_prices(self):
52      """Return a dict mapping names to current prices."""
53      url = "http://finance.yahoo.com/d/quotes.csv?f=sl1&s="
54      url += ",".join(sorted(s[0] for s in self.stocks))
55      data = urllib.urlopen(url)
56      return { sym: float(last) for sym, last in csv.reader(data) }
57
58  def value(self):
59      """Return the current value of the portfolio."""
60      prices = self.current_prices()
61      total = 0.0
62      for name, shares, _ in self.stocks:
63          total += shares * prices[name]
64      return total
```

```
>>> p = Portfolio()
>>> p.buy("IBM", 100, 150.0)
>>> p.buy("HPQ", 100, 30.0)

>>> p.current_prices()
{'HPQ': 32.45, 'IBM': 195.19}

>>> p.value()
22764.0
```

But how to test it?

- Live data: unpredictable
- Slow?
- Unavailable?
- Question should be:
  - "Assuming yahoo.com is working,
  - does my code work?"

Fake implementation of current_prices

```
45 # Replace Portfolio.current_prices with a stub implementation.
46 # This avoids the web, but also skips all our current_prices
47 # code.
48 class PortfolioValueTest(unittest.TestCase):
49     def fake_current_prices(self):
50         return {'IBM': 140.0, 'HPQ': 32.0}
51
52     def setUp(self):
53         self.p = Portfolio()
54         self.p.buy("IBM", 100, 120.0)
55         self.p.buy("HPQ", 100, 30.0)
56         self.p.current_prices = self.fake_current_prices
57
58     def test_value(self):
59         self.assertEqual(self.p.value(), 17200)
```

✓ Good: test results are predictable

But some code isn't tested!

```
$ coverage report -m
Name          Stmts   Miss  Cover   Missing
---------------------------------------------
portfolio3       31      4    87%   53-56
test_port7       43      0   100%
---------------------------------------------
TOTAL            74      4    95%
```

```
def current_prices(self):
    """Return a dict mapping names to current prices."""
    url = "http://finance.yahoo.com/d/quotes.csv?f=sl1&s="
    url += ",".join(sorted(s[0] for s in self.stocks))
    data = urllib.urlopen(url)
    return { sym: float(last) for sym, last in csv.reader(data) }

def value(self):
    """Return the current value of the portfolio."""
    prices = self.current_prices()
    total = 0.0
```

Fake urllib.urlopen instead

```
48 # A simple fake for urllib that implements only one method,
49 # and is only good for one request.  You can make this much
50 # more complex for your own needs.
51 class FakeUrllib(object):
52     def urlopen(self, url):
53         return StringIO('"IBM",140\n"HPQ",32\n')
54
55 class PortfolioValueTest(unittest.TestCase):
56     def setUp(self):
57         # Save the real urllib, and install our fake.
58         self.old_urllib = portfolio3.urllib
59         portfolio3.urllib = FakeUrllib()
60
61         self.p = Portfolio()
62         self.p.buy("IBM", 100, 120.0)
63         self.p.buy("HPQ", 100, 30.0)
64
65     def test_value(self):
66         self.assertEqual(self.p.value(), 17200)
67
68     def tearDown(self):
69         # Restore the real urllib.
70         portfolio3.urllib = self.old_urllib
```

All of our code is executed

```
$ coverage report -m
Name          Stmts   Miss  Cover   Missing
---------------------------------------------
portfolio3       31      0   100%
test_port8       49      0   100%
---------------------------------------------
TOTAL            80      0   100%
```

- ✓ Stdlib is stubbed
- ✓ All our code is run
- ✓ No web access during tests

Even better: mock objects

- Automatic chameleons
- Act like any object
- Record what happened to them
- You can make assertions afterward

```
>>> from mock import Mock

>>> func = Mock()
>>> func.return_value = "Hello!"

>>> func(17, "something")
'Hello!'

>>> func.call_args
call(17, 'something')
```

Mocking with no setup

```
50 class PortfolioValueTest(unittest.TestCase):
51     def setUp(self):
52         self.p = Portfolio()
53         self.p.buy("IBM", 100, 120.0)
54         self.p.buy("HPQ", 100, 30.0)
55 
56     def test_value(self):
57         # Create a mock urllib.urlopen.
58         with mock.patch('urllib.urlopen') as urlopen:
59 
60             # When called, it will return this value:
61             fake_yahoo = StringIO('"IBM",140\n"HPQ",32\n')
62             urlopen.return_value = fake_yahoo
63 
64             # Run the test!
65             self.assertEqual(self.p.value(), 17200)
66 
67             # We can ask the mock what its arguments were.
68             urlopen.assert_called_with(
69                 "http://finance.yahoo.com/d/quotes.csv"
70                 "?f=sl1&s=HPQ,IBM"
71                 )
```

Test doubles

- Powerful: isolates code
- Focuses tests
- Removes speed bumps and randomness
- BUT: fragile tests!
- Also: "dependency injection"

Also

Too many things I couldn't fit!

Topics

TDD: tests before code!?

BDD: describe external behavior

integration tests: bigger chunks

load testing: how much traffic is OK?

others, I'm sure....

Summing up

Testing is...

- Complicated
- Important

Questions?

http://bit.ly/pytest0 @nedbat