# Automated Testing with Python

`assertEqual(code.state(), 'happy')`

Martin Pitt
<martin.pitt@canonical.com>

CANONICAL™

# Why automated tests?

- avoid regressions
- easy code changes/refactoring
- simplify integration
- design tool
- documentation

# Unit test from Apport

```
$ python problem_report.py -v
basic creation and operation. ... ok
writing and re-decoding a big random file. ... ok
handling of CompressedValue values. ... ok
reading a report with binary data. ... ok
write_mime() with key filters. ... ok
[...]
----------------------------------------------------------------------
Ran 25 tests in 4.889s
```

# Unit tests

- smallest possible testable portion of the code
- noninteractive
- independent from each other
- no assumptions about environment or privileges
- small and fast

# Integration test from pkgbinarymangler

```
$ test/run  -v
Installed-Size gets updated ... ok
language packs are not stripped ... ok
OEM PPA for main package ... ok
OEM PPA for main package for blacklisted project ... FAIL
[...]

======================================================================
FAIL: OEM PPA for main package for blacklisted project
----------------------------------------------------------------------
Traceback (most recent call last):
  File "test/run", line 379, in test_ppa_oem_main_blacklisted
    self.assert_('locale/fr/LC_MESSAGES/vanilla.mo' in out)
AssertionError

----------------------------------------------------------------------
Ran 12 tests in 91.783s
FAILED (failures=1)
```

# Integration tests

- end-to-end testing of larger scenarios
- more expensive
- reasonable assumptions about environment
- should still be mostly noninteractive

# Python unittest structure

```
1  import counter
2  import unittest
3
4  class CounterTest(unittest.TestCase):
5      def setUp(self):
6          self.c = counter.Counter()
7          self.workdir = tempfile.mkdtemp()
8
9      def tearDown(self):
10         shutil.rmtree(self.workdir)
11
12     def test_foo(self):
13         #...
14
15 # main
16 unittest.main()
```

# Python unittest test cases

```
1  def test_init(self):
2      '''Counter is initialized with 0'''
3      self.assertEqual(self.c.count(), 0)
4
5  def test_inc(self):
6      '''Counter.inc() adds +1'''
7      self.c.inc()
8      self.assertEqual(self.c.count(), 1)
9
10 def test_str(self):
11     '''Counter string formatting'''
12     self.c.set(23)
13     self.assertEqual('x%sy' % self.c, 'x23y')
```

# Errors and corner cases

```
1  def test_nonint(self):
2      '''Rejects non-integers'''
3      self.assertRaises(TypeError, self.c.set, 1.5)
4
5  def test_huge(self):
6      '''Supports arbitrarily large numbers'''
7      self.c.set(sys.maxint)
8      self.c.inc()
9      self.assert_(self.c.count() > sys.maxint)
10     self.assertRaises(TypeError, self.c.set, 1.5)
11
12 def test_nonneg(self):
13     '''Can't count below zero'''
14     self.c.inc()
15     self.c.dec() # should be 0 now
16     self.assertRaises(ValueError, self.c.dec)
17     # defined value after error
18     self.assertEqual(self.c.count(), 0)
```

# Test-friendly code

- avoid hardcoded external addresses
- use proper abstractions (dbapi2, complete URLs, logic vs. GUI)
- break complex tasks into separate methods

# Techniques

- JFDI!
- locality for unit tests: mock objects
- locality for integration tests:

| | |
|---|---|
| files: | mini-chroots with `mkdtemp()` |
| databases: | sqlite :memory: |
| network: | `SimpleHttpServer` on `localhost` |
| packages: | `$APT_CONFIG` |
| devices: | loop, `$SYSFS_PATH` |

- UI testing: event synthesis, xvfb

# doctest

```
1  def factorial(n):
2      '''Return the factorial of n, an exact integer >= 0.
3
4      If the result is small enough to fit in an int,
5      return an int. Else return a long.
6
7      >>> factorial(0)
8      1
9      >>> factorial(5)
10     120
11
12     Negative values are not allowed:
13
14     >>> factorial(-1)
15     Traceback (most recent call last):
16     ...
17     ValueError: n must be >= 0
18     '''
19
20     n = 0 # ...
```

# Need more power?

- Mock objects: `python-mock`
- Comprehensive UI testing: `mago`
- kvm

# References

- http://en.wikipedia.org/wiki/Unit_testing
- http://docs.python.org/library/unittest.html
- test suites of Apport, Jockey, pkgbinarymangler, apt