Software Engineering with Intermediate Python

Friday: automated testing; unit tests

Outline

- Introduction to testing automation
- Unit tests part I: concepts and frameworks
- Practical 1+2
- Unit tests part II: mocks and stubs
- Practical 3

Test automation: a quick introduction

Test automation

Google™

- *Test automation* is the use of additional software (written code and external frameworks) to programmatically verify the behavior of a codebase, by executing it with known inputs and in a known environment.

- An *automated test* is a fragment of code that verifies one particular aspect of the behavior of the codebase.

Simple example

```
def fib(n):
  if n <= 1:
    return n
  else:
    ret
```

```
>>> test_fib()
True
```

```
def tes
  if (fib(0) == 0 and fib(1) == 1 and
      fib(2) == 1 and fib(20) == 6765):
    return True
  else:
    return False
```

If later we rewrite fib()...

```
def fib(n):
  """Iterative version of fib(). Should be faster."""
  a, b = 0, 1
  for i in range(n):
    a, b = b, a+b
  return a
```

```
>>> test_fib()
True
```

Benefits of automated tests

Google™

- Finding bugs early
- Better design of programs
- Easing maintenance and refactoring
- Verifying your fix for a bug is valid (with TDD)
- Reducing the need for manual testing

Desired traits in automated test cases

Google™

- Deterministic
- Comprehensive (to a certain level)
  - Regarding aspects
  - Regarding behavior
- Independent of other tests
- Test *one* property or aspect of the program behavior

Types of automated tests

Google™

- Unit tests
- Integration tests
- End-to-end tests

Introducing unit testing

Unit tests...

Google™

- Test one component in isolation
  - No calls to external resources (databases, remote servers)
- One test case: one code path through one method
  - Many test cases to test a method/class/module
- Must be fast (< 10ms because you'll have many of them)

Think...

Google™

- What tests/checks would you write for a "dict" object?
  - store, retrieval
  - contains ("key in dict")
  - length
  - deletion of keys
  - constructor
- Did anybody think of the edge cases?
  - passing both sequence & kwargs to __init__
  - dict[x] = 1; dict[x] = 2; dict[x]?

Approaches & helpers

- Approaches
  - Black box
  - White box
- Helpers
  - Coverage

Initial tips for writing testable code

Google™

- Prefer pure functions when possible (by separating pure and non-pure code)
- Keep your methods simple, and focused at one discrete task
- Avoid global state
- Don't do work in the constructor

Anatomy of a Python unit test (1)

```
class DictionaryTest(unittest.TestCase):

  def testStoreRetrieve(self):
    d = dict()
    d['x'] = 'y'
    d['y'] = 'z'
    self.assertEqual('y', d['x'])  # (expected,
actual)
    self.assertEqual('z', d['y'])

  def testContains(self):
    d = dict()
    d['x'] = 'y'
    self.assertTrue('x' in d)
    self.assertFalse('z' in d)
```

Anatomy of a Python unit test (2)

```
class CalculatorTest(unittest.TestCase):

  def setUp(self):
    self.calc = utils.Calculator()

  def testDivision(self):
    self.assertAlmostEqual(3.2,
                           self.calc.Divide(32, 10))

  def testDivisionByZero(self):
    self.assertRaises(ZeroDivisionError,
                      self.calc.Divide, 32, 0)
```

Anatomy of a Python unit test (3)

```
import unittest

class DictionaryTest(unittest.TestCase):
  ...

class CalculatorTest(unittest.TestCase):
  ...

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

Running Python tests (1)

```
% python example_test.py
....
----------------------------------------------------
Ran 4 tests in 0.000s

OK

% python sreu_example_test.py -v
testDivision (__main__.CalculatorTest) ... ok
testDivisionByZero (__main__.CalculatorTest) ... ok
testContains (__main__.DictionaryTest) ... ok
testStoreRetrieve (__main__.DictionaryTest) ... ok

----------------------------------------------------
Ran 4 tests in 0.000s

OK
```

Running Python tests (2)

```
% python example_test.py DictionaryTest
..
----------------------------------------------------------
Ran 2 tests in 0.000s

OK

% python sreu_example_test.py -v CalculatorTest.testDivision
testDivision (__main__.CalculatorTest) ... ok

----------------------------------------------------
Ran 1 test in 0.000s

OK
```

Python unit tests in Google

```
from google3.testing.pybase import googletest


class DictionaryTest(googletest.TestCase):
  ...


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

googletest.TestCase has a very rich set of assertion methods

Running Python tests in Google

```
### BUILD rule
py_test(name = "my_module_test",
        size = "small",
        srcs = ["my_module_test.py"],
        deps = [":my_module",
                "//testing/pybase"])

% blaze test :my_module_test

% blaze test :my_module_test MyModuleTest.testFoo
```

Running tests explained

Google

- *main()* scans the current file for classes derived from *TestCase*
- *main()* scans each class for methods prefixed by *test*
- then, for each method in each test class:
  - instantiates the class
  - calls the *setUp()* method, if one exists
  - runs the test method
  - calls the *tearDown()* method, if one exists

setUp()/*tearDown()* are called <u>once per test method</u>

Test Driven Development: the rules

Google™

1. You can't write production code unless there is a broken test.

2. When there is a broken test, change your code to make it pass.

3. When your tests are passing, refactoring is allowed.

Benefits of TDD

Google™

- The final state to achieve is carefully thought out in advance, and the tests serve as a concrete measure of progress.
- When writing libraries, the tests are your first users, and make you think about an API that makes sense.
- The resulting code tends to be better factored, loosely coupled, more readable and maintenable.
- It guarantees there *will* be tests at the end of the implementation.

Mocks and stubs for unit tests

The need for substitutive objects

Google™

- Unit tests can't access remote resources like databases and servers. Reasons (all of them):
  - speed
  - determinism of tests
  - load on production servers
- In unit tests, substitutive objects (*stubs* or *mocks*) are used to avoid such external communications
- Used also for other sources of variability, like *time()*

Conceptual example (1)

```
def UpdateFileWithBackup(path, new_contents):
  backup_path = path + '.bak-%d' % time.time()
  shutil.copy2(path, backup_path)
  with open(path, 'w') as f:
    f.write(new_contents)

def testUpdateFileWithBackup(self):
  open('/tmp/test.txt', 'w').write('old contents')
  UpdateFileWithBackup('/tmp/test.txt', 'new contents')
  self.assertEqual('new contents',
                   open('/tmp/test.txt').read())
  backup_file = '/tmp/test.txt.bak-%d' % time.time()
  self.assertEqual('old contents', open(backup_file).
read())
```

Conceptual example (2)

```
def TimeStub():
  return 1234567890

def testUpdateFileWithBackup(self):
  open('/tmp/test.txt', 'w').write('old contents')
  old_time_time = time.time
  time.time = TimeStub
  UpdateFileWithBackup('/tmp/test.txt', 'new contents')
  self.assertEqual('new contents',
                   open('/tmp/test.txt').read())
  backup_file = '/tmp/test.txt.bak-1234567890'
  self.assertEqual('old contents', open(backup_file).
read())
  time.time = old_time_time
```

StubOutForTesting

Google

```
class MyTest(googletest.TestCase):
  def setUp(self):
    self.stubs = googletest.StubOutForTesting()

  def tearDown(self):
    self.stubs.UnsetAll()

def testUpdateFileWithBackup(self):
  open('/tmp/test.txt', 'w').write('old contents')
  self.stubs.Set(time, 'time', TimeStub)
  UpdateFileWithBackup('/tmp/test.txt', 'new contents')
  self.assertEqual('new contents',
                   open('/tmp/test.txt').read())
  backup_file = '/tmp/test.txt.bak-1234567890'
  self.assertEqual('old contents', open(backup_file).
read())
```

Dependency injection

If you can't access that, get http://www/~dato/no_crawl/sreu_c03b.zip.

```
def UpdateFileWithBackup(path, new_contents,
                         time_func=None):
  if not time_func:
    time_func = time.time()
  backup_path = path + '.bak-%d' % time_func()
  ...

def testUpdateFileWithBackup(self):
  open('/tmp/test.txt', 'w').write('old contents')
  UpdateFileWithBackup('/tmp/test.txt', 'new contents',
                       time_func=TimeStub)
  self.assertEqual('new contents',
                   open('/tmp/test.txt').read())
  backup_file = '/tmp/test.txt.bak-1234567890'
  self.assertEqual('old contents', open(backup_file).
read())
```

Dependency injection at the module level

```
time_func = time.time()

def UpdateFileWithBackup(path, new_contents):
  backup_path = path + '.bak-%d' % time_func()
  ...

def testUpdateFileWithBackup(self):
  open('/tmp/test.txt', 'w').write('old contents')
  self.stubs.Set(util, 'time_func', TimeStub)
  util.UpdateFileWithBackup('/tmp/test.txt', 'new
contents')
  self.assertEqual('new contents',
                   open('/tmp/test.txt').read())
  backup_file = '/tmp/test.txt.bak-1234567890'
  self.assertEqual('old contents', open(backup_file).
read())
```

More complex example

```
def GetUrlContents(url, max_retries=5):
  attempt = 1
  while True:
    try:
      return urllib.urlopen(url).read()
    except IOError:
      if attempt == max_retries:
        raise
      else:
        time.sleep(attempt * 5)
        attempt += 1
```

Mocks; pymox

```
import mox

def testGetUrlContentsDoesRetry(self):
  mox = mox.Mox()
  mox.StubOutWithMock(urllib, 'urlopen')
  mox.StubOutWithMock(time, 'sleep')
  fp = StringIO.StringIO('url contents')
  urllib.urlopen(self.url).AndRaise(IOError())
  urllib.urlopen(self.url).AndRaise(IOError())
  urllib.urlopen(self.url).AndReturn(fp)
  time.sleep(5)
  time.sleep(10)
  mox.ReplayAll()
  self.assertEqual('url contents', GetUrlContents(self.
url))
  mox.VerifyAll()
```

go/MocksAtGoogle

Google

- Google-specific mocks to avoid abusing PyMox
  - filesystems (*fake_filesystem*)
  - BigTable (*//bigtable/python:pywrapmocktestutil*)
  - MapReduces (*//mapreduce/public:pywraptesthelper*)

- (Many others for C++ and Java)