

# Test Driven Development in Python

---

---

Kevin Dahlhausen  
kevin.dahlhausen@keybank.com

---

# My (pythonic) Background

- learned of python in 96 ← Vim Editor
  - Fast-Light Toolkit python wrappers
  - PyGallery – one of the early online photo gallery generators
  - wxGlade plugin
  
  - despise HDD – *Hype-Driven Design*
-

---

# Could there be something behind the hype?

[...] I had been a programmer for nearly thirty years before I was introduced to TDD. I did not think anyone could teach me a low level programming practice that would make a difference. Thirty years is a lot of experience after all. But when I started to use TDD, *I was dumbfounded at the effectiveness of the technique.*

-- Robert Martin

---

---

# Test Driven Development

- ❑ design methodology -> artifacts for testing
  - ❑ short cycle repeated many times:
    - write a test
    - watch it fail
    - make it compile
    - make it pass
    - refactor the code
    - refactor the test (and elaborate)
    - rinse and repeat
-

---

# What does this do for you?

- your api is determined by using it
  - written minimal amount of *application* code
    - total application + tests is probably more
  - objects: simpler, stand-alone, minimal dependencies
  - tends to result in extensible architectures
  - instant feedback
  - safety net – same feeling as w/scc
-

---

# Python Makes TDD Easier

- typing: dynamic & duck
  - dynamic object changes
  - reflection
    - flexible testing infrastructures
    - reduced developer overhead
  - rapid edit-compile-run cycles
  - ease of on-the-fly class definition
  - it's python
-

---

# Tools for test-driven Python

- there are many. we will cover:
    - unittest
    - pymock
    - nose
    - nosy
    - py.test
  
  - Grig Gheorghiu's *Python Testing Tools Taxonomy*:
    - <http://pycheesecake.org/wiki/PythonTestingToolsTaxonomy>
    - tool, category, author, **claim to fame**
-

---

# unittest

- *'batteries included'* since 2.1
    - *PyUnit* before that
  - python version of *JUnit*
    - **TestCase**
      - `assert_`, `equality`, `almost equality`, raises exception
      - `assertEqual( a, b, msg)`
      - `setUp / tearDown` methods
      - doc strings
    - **TestSuite**
      - `addTest`, `run`
      - auto-discovery tools vs. maintenance by hand of test-suites
-

---

# unittest/TDD example

- Game needs a 'Tank'
  - Pygame -> Sprite -> image, rect attr
  - start with test:
    - testTankImage:
      - create a tank object and assert that tank has image and image is not None
-

---

# unittest/TDD: first step

## TestTank.1.py:

```
import unittest
```

```
class TestTank(unittest.TestCase):
```

```
    def testTankHasImage(self):
```

```
        tank = Tank()
```

```
        assert tank.image != None, "tank image is None"
```

```
if __name__ == "__main__":
```

```
    unittest.main()
```

---

---

# unittest/TDD: first step results

```
c:/cygwin/bin/sh.exe -c "python TestTank.1.py"
```

```
E
```

```
=====
```

```
ERROR: testTankHasImage (__main__.TestTank)
```

```
-----
```

```
Traceback (most recent call last):
```

```
  File "TestTank.1.py", line 6, in testTankHasImage
```

```
    tank = Tank()
```

```
NameError: global name 'Tank' is not defined
```

```
-----
```

```
Ran 1 test in 0.000s
```

---

---

# unittest/TDD: second step

## TestTank.2.py

```
import unittest
from Tank import Tank

class TestTank(unittest.TestCase):

    def testTankHasImage(self):
        tank = Tank()
        assert tank.image != None, "tank image is None"

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

---

---

# unittest/TDD: second step results

~/: python TestTank.1.py

Traceback (most recent call last):

File "TestTank.1.py", line 2, in ?

from Tank import Tank

ImportError: No module named Tank

---

---

# unittest/TDD: third step

add missing module *'doing the simplest thing that could possibly work'*:

**Tank.py:**

```
pass
```

yes, kind of silly at this point, but important for TDD to stick to this model of coding

---

---

# unittest/TDD: third step results

~: python TestTank.2.py

Traceback (most recent call last):

File "TestTank.2.py", line 2, in ?

from Tank import Tank

ImportError: cannot import name Tank

---

---

# unittest/TDD: fourth step

## **Tank.py:**

```
class Tank:  
    pass
```

## **Results:**

```
c:/cygwin/bin/sh.exe -c "python TestTank.2.py"
```

```
E
```

```
=====
```

```
ERROR: testTankHasImage (__main__.TestTank)
```

```
-----
```

```
Traceback (most recent call last):
```

```
  File "TestTank.2.py", line 8, in testTankHasImage  
    assert tank.image != None, "tank image is None"  
AttributeError: Tank instance has no attribute 'image'
```

```
-----
```

```
Ran 1 test in 0.000s
```

---

---

# unittest/TDD: fifth step

## **Tank.py:**

```
class Tank:  
    def __init__(self):  
        self.image = pygame.image.load("tank.png")
```

## **Results:**

```
c:/cygwin/bin/sh.exe -c "python TestTank.2.py"
```

```
.
```

```
-----
```

```
Ran 1 test in 0.000s
```

```
OK
```

---

---

# unittest/TDD: sixth step

sprites in pygame must also have a rect attribute. so let's add a test for this:

```
def testTankHasRectangle(self):  
    tank = Tank()  
    assert tank.rect != None,  
           "tank rectangle is None"
```

test fails -> Tank instance has no attribute 'rect'

---

---

# unittest/TDD: sixth step

- doing the simplest thing:

```
import pygame
class Tank:
    def __init__(self):
        self.image = pygame.image.load("tank.png")
        self.rect = None
```

- now the assertion fails:

- 'tank rectangle is None'



---

# unittest/TDD: seventh step

so make the test pass:

```
import pygame
class Tank:
    def __init__(self):
        self.image = pygame.image.load("tank.png")
        self.rect = self.image.get_rect()
```

## Results:

```
~: python TestTAnk.3.py
```

```
..
```

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

```
OK
```

---

# unittest/TDD: eighth step

- **all tests pass so:**
  - **refactor code**
  - **ensure tests still pass**
  - **refactor tests:**

```
import unittest
from Tank import Tank

class TestTank(unittest.TestCase):

    def testTankHasImage(self):
->         tank = Tank()
           assert tank.image != None, "tank image is None"

    def testTankHasRectangle(self):
->         tank = Tank()
           assert tank.rect != None, "tank rectangle is None"

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

---

# unittest/TDD: ninth step

factor out duplicated code and place in *setUp* function:

```
import unittest
import Tank

class TestTank(unittest.TestCase):

    def setUp(self):
        self.tank = Tank.Tank()

    def testTankHasImage(self):
        self.assertNotEqual(self.tank.image, None,
                             "tank does not have an image")

    def testTankHasRect(self):
        self.assertNotEqual(self.tank.rect, None,
                             "tank does not have a rectangle")

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

---

---

# unittest/TDD: tenth step

verify that all tests still pass:

```
c:/cygwin/bin/sh.exe -c "python TestTank.py"
```

```
..
```

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

```
OK
```

---

---

# PyMock

- by Jeff Younker
  - mock-objects:
    - most objects collaborate with others
    - mock object is a “*test-oriented*’ replacement (object) for a collaborator” – java’s EasyMock website
    - verifies method calls/parameters (vs. stub)
  - simulates objects (function calls, properties, functions, and generators) that interact with your object under test - and verify the calls
  - record – playback – verify
  - (demo 2)
-

# PyMock - sample

## TestTankDraw.py:

```
import unittest
import pymock
import pygame
from Tank import Tank

class TestTankDraw(unittest.TestCase):

    def testDrawTank(self):

        tankGroup = pygame.sprite.Group()
        tank = Tank()

        controller = pymock.Controller()

1         surface = controller.mock()
2         surface.blit( tank.image, tank.rect)

3         controller.replay()

        tankGroup.add(tank)
        tankGroup.draw(surface)

4         controller.verify()

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

# PyMock

- can temporarily override parts of existing objects!!

```
import pymock
import os

class TestOverride(pymock.PyMockTestCase):

    def testOverride(self):

1         self.override(os, 'listdir') # object, method name
2         self.expectAndReturn(os.listdir(), ['file-one', 'file-two'])

3         self.replay()

         result = os.listdir()

         self.assertEqual( result, ['file-one', 'file-two'],
                           "wrong files returned")
4         self.verify()

if __name__=="__main__":
    import unittest
    unittest.main()
```

---

# Nose

- by Jason Pellerin
  - discovers and runs unit tests
    - fast – tests while gathering
  - anything that matches regex is a test
    - no required superclass
    - even stand-alone functions
      - `@with_setup( setup, teardown )` decorator
  - plugins:
    - tagging and running by tags
    - doctest
-

---

# Nose – simple functions

## TestNoseSimpleFunction.py:

```
import Tank

def testDriveLeft():
    tank = Tank.Tank(100,100)
    tank.driveLeft()
    assert tank.velocity == (-1,0),
           "tank velocity wrong when driving left"
```

---

---

# Nose – functions with setup/teardown

**simple functions can have setup/teardown:**

```
from nose import with_setup
import Tank

tank = None

def setupFunction():
    print "setup"
    global tank
    tank = Tank.Tank(100, 100)

def teardownFunction():
    print "tear down"

@with_setup(setupFunction)
def testDriveUp():
    tank.driveUp()
    assert tank.velocity == (0,-1), "tank velocity wrong when driving up"

@with_setup(setupFunction, teardownFunction)
def testDriveDown():
    tank.driveDown()
    assert tank.velocity == (0,1), "tank velocity wrong when driving down"
```

---

---

# Nose – functions with setup/teardown

results:

```
~/setupTests: nosetests -v -s TestNoseSetup.py
```

```
TestNoseSetup.testDriveUp ... setup
```

```
ok
```

```
TestNoseSetup.testDriveDown ... setup
```

```
tear down
```

```
ok
```

```
-----
```

```
Ran 2 tests in 0.234s
```

---

---

# Nose

- supports test generators
    - test function yields a function and parameters to apply to that function
    - will automatically call the function for each set returned by the generator
-

---

# Nose: generator demo

## TestNoseGenerator.py:

```
def test_tankXPositioning():
    for x in [ 0, 160, 161]:
        yield checkTankPosition, x

def checkTankPosition(x):
    tank = Tank.Tank( x, 100)
    assert tank.rect.left==x
```

## Results:

```
c:/cygwin/bin/sh.exe -c "nosetests -v"
TestNoseGenerator.test_tankXPositioning:(0,) ... ok
TestNoseGenerator.test_tankXPositioning:(160,) ... ok
TestNoseGenerator.test_tankXPositioning:(161,) ... ok
```

```
-----
Ran 3 tests in 0.219s
```

---

---

# Nose: code coverage

- uses Ned Batchelder's *Coverage.py*
    - just download and place on path
  - features:
    - optionally run coverage for tests
    - can check source under tree not imported by tests
    - specify particular package
  - **run:** `nosetests --with-coverage`
  - **demo**
-

# Nose: code coverage report

```
21 ~/proj/superTank2: nosetests --with-coverage
```

```
.....
```

Name	Stmts	Exec	Cover	Missing
Border	10	10	100%	
...				
JoystickDriver	76	48	63%	25, 37, 52-53, 60, 77-110
KeyboardDriver	28	28	100%	
MockPyGame	18	18	100%	
MockSurface	10	10	100%	
Shot	57	52	91%	16, 69, 72, 78, 81
SuperTank	120	105	87%	35-38, 47-53, 94-97
Tank	93	92	98%	23
TankExplosion	21	20	95%	19
Tree	8	8	100%	
-----				
TOTAL	2985	1260	42%	
-----				

```
Ran 94 tests in 0.891s
```

---

# Nose: test profiling

- uses 'hotshot' profiler
- very configurable – requires some studying
- best practice:
  - save to file with `--profile-stats-file`
  - post process with python scripts for reporting
- example:

```
nosetests --with-profile --profile-statsfile=stats.dat
```

---

# Nose: profiling report

- with reporting script:

```
from hotshot import stats
s = stats.load("stats.dat")
s.sort_stats("time").print_stats()
```

26409 function calls (26158 primitive calls) in 1.997 CPU seconds

Ordered by: internal time

```
ncalls  tottime  percall  cumtime  percall  filename:lineno(function)
   1   0.422   0.422   0.651   0.651  c:\python24\lib\site-
packages\numeric\numeric.py:85(?)
   1   0.238   0.238   0.896   0.896  c:\python24\lib\site-packages\pygame\__init__.py:25(?)
   1   0.159   0.159   0.166   0.166  c:\python24\lib\pickle.py:25(?)
  56   0.131   0.002   0.135   0.002
c:\kpd\proj\codemashpresentation\supertank2\src\tank.py:11(__init__)
   1   0.095   0.095   1.051   1.051
c:\kpd\proj\codemashpresentation\supertank2\src\testborder.py:1(?)
```

...

---

# Nose: profiling – limited report

with reporting script:

```
from hotshot import stats
s = stats.load("stats.dat")
s.sort_stats("time").print_stats('src.test', .1)
```

26409 function calls (26158 primitive calls) in 1.997 CPU seconds

Ordered by: internal time

List reduced from 484 to 164 due to restriction <'src.test'>

List reduced from 164 to 16 due to restriction <0.10000000000000001>

```
ncalls tottime percall cumtime percall filename:lineno(function)
  1  0.095  0.095  1.051  1.051
c:\kpd\proj\codemashpresentation\supertank2\src\testborder.py:1(?)
  3  0.011  0.004  0.018  0.006
c:\kpd\proj\codemashpresentation\supertank2\src\testtankimagecoloring.py:6(verifyTankC
oloring)
  1  0.007  0.007  0.018  0.018
c:\kpd\proj\codemashpresentation\supertank2\src\testdrawtankexplosion.py:1(?)
```

...

---

---

# Nose – debugging support

- drop into python debugger
  - on error
    - `noستests --pdb`
  - on failure
    - `noستests -pdb-failures`

---

# Nosy

- started as blog post by Jeff Winkler
  - automatic continuous testing
    - similar to 'Cruise Control'
  - runs nose when changes detected
    - run in a window while you work
    - save file -> view unit test output in nosy window
  - scans directory for changes in python source code
    - mod in comments adds directory recursion
  - live demo
-

---

# py.test

- by Holger Krekel (and others)
  - nose is very similar
  - no api model -> test\_funcs, TestXXX class
  - collects tests
  - module, class, and member-function setup/teardown
  - test generators
  - test ordering
  - test for deprecation warnings
    - `py.test.deprecated_call(func, *args, **kwargs)`
  - test selection by keyword `-k <prefix>`
-

# Case Study - Super Tank in Pygame

## ■ history:

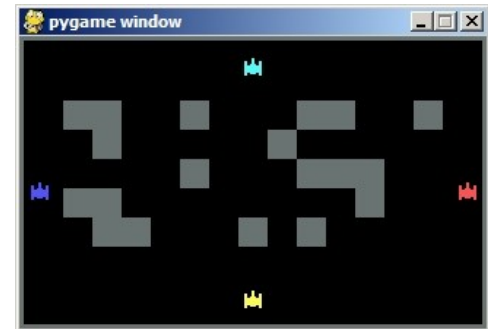
- late 70's coin-op
- 1982 TI-99/4a ext. basic
- 1992 'C' w/Mode 13 VGA
- 2007 pygame!



## ■ goal: rigorous TDD game development

## ■ current status:

- tanks, shots, borders, key/joystick input, movement, collision detection, explosions
- 92 tests run in 0.6s
- 773 lines in 28 files
  - 471 lines game code in 10 files (37%)
  - 813 lines test code in 18 files (63%)



---

# Case-Study: Super Tank Game

first few cycles went like this:

2. test pygame initialization (testInit)
  3. test game runs until 'escape' pressed
    - ❑ mocked out framework's `get_event` function
  4. test drawing tank on the screen
    - ❑ may be too big a chunk of testing at once
    - ❑ drove need for 'Tank' class
    - ❑ drove need for Tank to be pygame sprite
    - ❑ tested draw by mocking out pygame drawing surface and checking for image blit
-

---

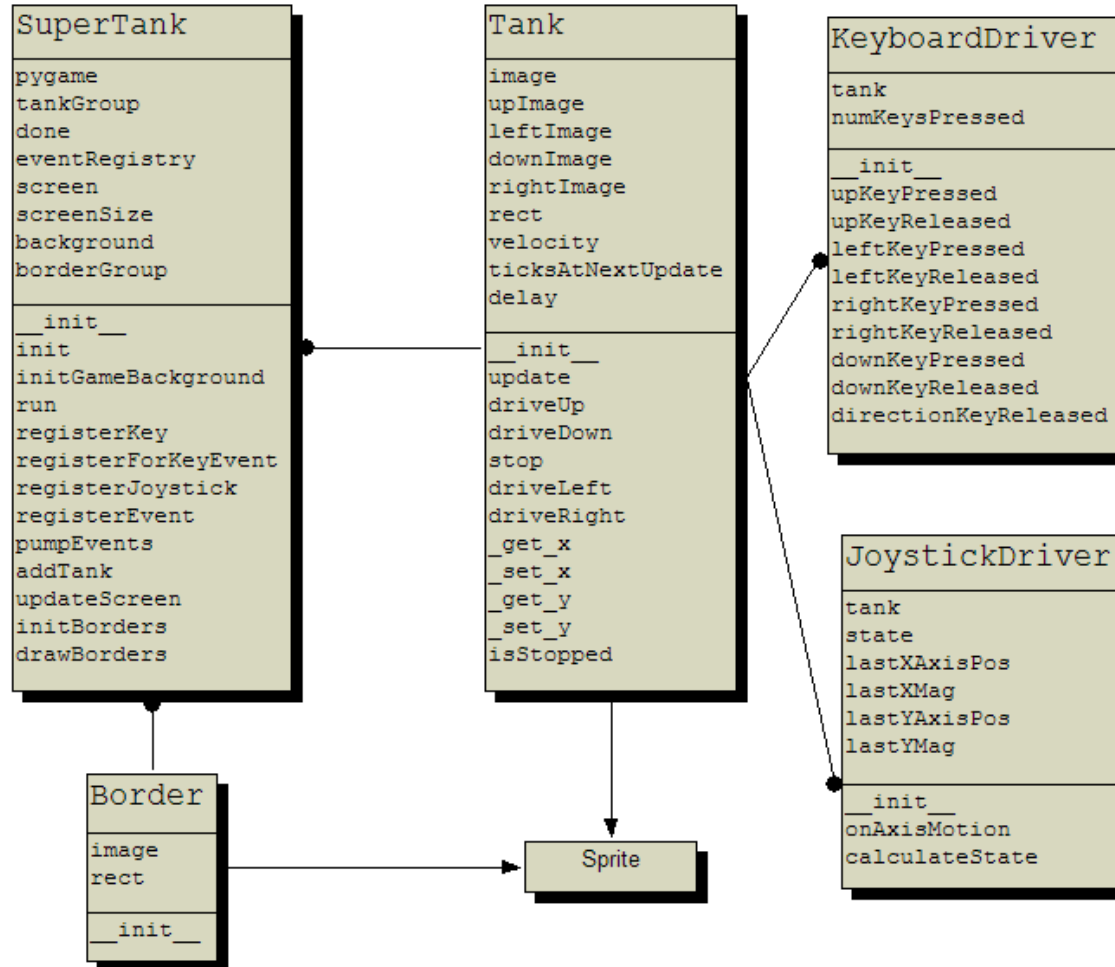
# Case-Study: Super Tank Game

1. ensure tank drawn from game loop
  - ❑ game loop is hard to unit test so this forced 'updateScreen' function
    - ❑ now could test updateScreen blits tank
    - ❑ also test game loop calls updateScreen

tests progressed through tank colors, input event registration and callbacks, tank movement, tank direction changes (and respective tank images), border blocks on screen, tank collision with blocks, joystick driver, interior walls, shots, explosions

---

# (Case Study) TDD Generated Class



---

# (Case Study) Lessons Learned

- was able to TDD everything except:
    - background paint when sprite moves
    - image conversion to current screen format
    - tank movement delay / FPS timing
  - progress starts off slow then accelerates
  - the resulting app is very extensible
    - adding joystick tank driver
  - working with a safety net is *\*really\** nice
  - unit tests (can) run fast
-

---

# (Case Study) Lessons Learned

- mock objects are almost indispensable when dealing with interacting libraries
  - PyMock is amazing
  - ratio of source to test code stayed fairly constant at about 1:2 ( 1:1.9 -> 1:1.7)
  - nosetest / nosy combination is quite handy
  - thinking about how to test for a behavior drives nicer implementations
  - tests first – no shortcuts
-

---

# Resources

- *Python Testing Tools Taxonomy:*
    - <http://pycheesecake.org/wiki/PythonTestingToolsTaxonomy>
  - *unittest module*
    - <http://docs.python.org/lib/module-unittest.html>
  - *Pymock:*
    - <http://theblobshop.com/pymock/>
  - *Nose:*
    - <http://code.google.com/p/python-nose/>
    - code coverage plugin:  
<http://www.nedbatchelder.com/code/modules/coverage.html>
  - *Nosy:*
    - <http://jeffwinkler.net/2006/04/27/keeping-your-nose-green/>
  - *py.test*
    - <http://codespeak.net/py/current/doc/test.html>
-