Agile and Test-Driven Development

TDD Worked Example

Adapted from “Test-Driven Development By Example”, Kent Beck

Introduction

- Very simple example
- Implement a function to return the *n*th number in the Fibonacci sequence
  - 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, …
  - http://oeis.org/A000045

$$F_0 = 0,$$
$$F_1 = 1,$$
$$F_n = F_{n-1} + F_{n-2}$$

Set up

- Create two directories
  - `src`
  - `test`
- Add the `src` directory to PYTHONPATH
  - `export PYTHONPATH=`pwd`/src`

Step 1: Write a test

- Create a file: `test/test_fibonacci.py`

```
import unittest
from fibonacci import fibonacci

class TestFibonacci(unittest.TestCase):

    def test_fibonacci(self):
        self.assertEqual(0, fibonacci(0), "fibonacci(0) should equal 0")

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

Step 1: Run the test

```
Traceback (most recent call last):
  File "test/test_fibonacci.py", line 10, in <module>
    from fibonacci import fibonacci
ImportError: No module named fibonacci
```

FAIL

Step 1: Implement and re-test

- Create a file: src/fibonacci.py

```
def fibonacci(n):
    return 0
```

```
.
----------------------------------------------------------------------
Ran 1 test in 0.000s

OK
```

PASS

Step 2: Write a test

```
import unittest
from fibonacci import fibonacci

class TestFibonacci(unittest.TestCase):

    def test_fibonacci(self):
        self.assertEqual(0, fibonacci(0), "fibonacci(0) should equal 0")
        self.assertEqual(1, fibonacci(1), "fibonacci(1) should equal 1")

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

Step 2: Run the tests

```
F
======================================================================
FAIL: test_fibonacci (__main__.TestFibonacci)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "test/test_fibonacci.py", line 16, in test_fibonacci
    self.assertEqual(1, fibonacci(1), "fibonacci(1) should equal 1")
AssertionError: fibonacci(1) should equal 1

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

FAILED (failures=1)
```

FAIL

Step 2: Implement and re-test

```
def fibonacci(n):
    if n == 0: return 0
    return 1
```

```
.
----------------------------------------------------------------------
Ran 1 test in 0.000s

OK
```

PASS

Step 3: Write a test

```
import unittest
from fibonacci import fibonacci

class TestFibonacci(unittest.TestCase):

    def test_fibonacci(self):
        self.assertEqual(0, fibonacci(0), "fibonacci(0) should equal 0")
        self.assertEqual(1, fibonacci(1), "fibonacci(1) should equal 1")
        self.assertEqual(1, fibonacci(2), "fibonacci(2) should equal 1")

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

Step 3: Run the tests

!

```
.
----------------------------------------------------------------------
Ran 1 test in 0.000s

OK
```

PASS

Step 4: Write a test

```
import unittest
from fibonacci import fibonacci

class TestFibonacci(unittest.TestCase):

    def test_fibonacci(self):
        self.assertEqual(0, fibonacci(0), "fibonacci(0) should equal 0")
        self.assertEqual(1, fibonacci(1), "fibonacci(1) should equal 1")
        self.assertEqual(1, fibonacci(2), "fibonacci(2) should equal 1")
        self.assertEqual(2, fibonacci(3), "fibonacci(3) should equal 2")

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

Step 4: Run the tests

```
F
======================================================================
FAIL: test_fibonacci (__main__.TestFibonacci)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "test/test_fibonacci.py", line 18, in test_fibonacci
    self.assertEqual(2, fibonacci(3), "fibonacci(3) should equal 2")
AssertionError: fibonacci(3) should equal 2

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

FAILED (failures=1)
```

FAIL

Step 4: Implement and re-test

```
def fibonacci(n):
    if n == 0: return 0
    if n <= 2: return 1
    return 2
```

```
.
----------------------------------------------------------------------
Ran 1 test in 0.000s

OK
```

PASS

Step 5: Write a test

```
import unittest
from fibonacci import fibonacci

class TestFibonacci(unittest.TestCase):

    def test_fibonacci(self):
        self.assertEqual(0, fibonacci(0), "fibonacci(0) should equal 0")
        self.assertEqual(1, fibonacci(1), "fibonacci(1) should equal 1")
        self.assertEqual(1, fibonacci(2), "fibonacci(2) should equal 1")
        self.assertEqual(2, fibonacci(3), "fibonacci(3) should equal 2")
        self.assertEqual(3, fibonacci(4), "fibonacci(4) should equal 3")

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

Step 5: Run the tests

```
F
======================================================================
FAIL: test_fibonacci (__main__.TestFibonacci)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "test/test_fibonacci.py", line 19, in test_fibonacci
    self.assertEqual(3, fibonacci(4), "fibonacci(4) should equal 3")
AssertionError: fibonacci(4) should equal 3

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

FAILED (failures=1)
```

FAIL

Step 5: Implement and re-test

```
def fibonacci(n):
    if n == 0: return 0
    if n <= 2: return 1
    if n == 3: return 2
    return 3
```

```
.
----------------------------------------------------------------------
Ran 1 test in 0.000s

OK
```

PASS

Step 6: Refactor and test

```
def fibonacci(n):
    if n == 0: return 0
    if n <= 2: return 1
    if n == 3: return 2
    return 2 + 1
```

$$F_0 = 0,$$
$$F_1 = 1,$$
$$F_n = F_{n-1} + F_{n-2}$$

```
.
----------------------------------------------------------------------
Ran 1 test in 0.000s

OK
```

PASS

Step 7: Refactor and test

```
def fibonacci(n):
    if n == 0: return 0
    if n <= 2: return 1
    return fibonacci(n - 1) + fibonacci(n - 2)
```

$$F_0 = 0,$$
$$F_1 = 1,$$
$$F_n = F_{n-1} + F_{n-2}$$

```
.
----------------------------------------------------------------------
Ran 1 test in 0.000s

OK
```

PASS

Step 8: Refactor and test (and done)

```
def fibonacci(n):
    if n == 0: return 0
    if n == 1: return 1
    return fibonacci(n - 1) + fibonacci(n - 2)
```

$$F_0 = 0,$$
$$F_1 = 1,$$
$$F_n = F_{n-1} + F_{n-2}$$

```
.
----------------------------------------------------------------------
Ran 1 test in 0.000s

OK
```

PASS

Are we really done?

- What about fibonacci(-1) ?
  - Should we test for this input?
  - What should we return?
    - An exception?
- What about fibonacci(1000000000) ?
  - Will this take too long?

Things to note

- Fibonacci implementation completely derived from the tests
- Only implement what is needed to pass tests
  - If we only ever needed fibonacci(0), then

```
def fibonacci(n):
    return 0
```

    would have sufficed and we would have stopped
- No tests written during refactoring

Useful links

- Fibonacci on Wikipedia:
  - http://en.wikipedia.org/wiki/Fibonacci_number
- The step-by-step code from this lecture:
  - https://gitlab.cs.man.ac.uk/robert.haines/comp61542-fibonacci
- Python unittest module documentation:
  - http://docs.python.org/2/library/unittest.html