Testing

Moorthy

Testing

Created by Moorthy / @mskmoorthy and modified by Wesley Turner / @wd-turner

Reading Material

- unittest - Unit testing framework
- doctest - Test interactive Python examples

Testing

- Testing of Systems
- Testing of Code

Purpose

- Software Quality Control/Software Quality Assurance
- Validation - building the right product
- Verification - does the code/system implement the specification
- Possibly avoid debugging

Debugging vs Testing

- Debugging is a cyclic activity - code execution and correction
- Aim of Debugging is to locate errors
- Testing is to demonstrate "correctness"

Testing of Code

- Black Box - Treat the system as a black box and generate test cases based on input/output specification

- Gray Box - Limited knowledge of the system and generate test cases based on it. Ensure that code correctly cleans up after itself
- White Box - Code details are known. Tests are based on the code structure.

Types of Tests

- Smoke Tests - most common kind of test - non extensive - most crucial functions of a program work
- Functional Tests - Done by QA to test functionality according to a test plan based on requirements and design specs.
- Unit Tests - Done by developers to test specific code.
- Regressions Tests - Change in code does not affect other parts

Why Write Unit Tests

- Increase Developer Confidence
- Avoid regression - If a unit test is run frequently enough, one knows when new code breaks old code.
- If you write tests first, you know when you are done.
- Encourage maximal modularity and minimal interface

When to Write Unit Tests

- Always write tests.
- Before you check the code into repository, you know your code works.
- Before and after refactoring - redesign/reimplementation does not break
- Before Debugging, to ease the process and help you know when you have finished debugging.

Terminology

An error is made by an Engineer/Algorithm Designer/Implementor

A fault is manifestation of that error in the code

A failure is incorrect output behavior is caused by executing a fault

Testing attempts to discover failures

- Debugging associates failures with faults and then corrects the fault.

If a system passes all of its tests, is the system free of faults?

NO!

Why No ...

- Faults may be hiding in portions of code that rarely get executed.
- Sometimes faults mask one another

- Creating a test or test suite that covers all code paths and all functional units is essentially impossible.

However, Having all tests pass increases our confidence that our system has high quality

Looking for Faults

- The input/output space of any software system is vast
- Tests are a way of sampling the behaviors of a software system looking for failures
- Partition the space into equivalent behaviors and sample each partition

Example

- GCD Program takes two numbers and computes the GCD

```
def gcd(a,b):
    while(1 > 0):
        if (a == 0):
            return b
        b = b % a
        if (b == 0):
            return a
        a = a % b
```

Test Cases for the GCD Program

- Assumptions in data a and b are nonnegative integers. Try edge cases
- Assumptions about Data are implemented in the program.
- a = 9, b =3 gcd = 3
- a = 2, b = 6 gcd = 2
- a = 3 , b= 11 gcd =1 prime numbers
- a = 0, b = 10 gcd=? edge case
- a = -9, b =18 gcd=? data constraint

Test Coverage

Covers all the paths

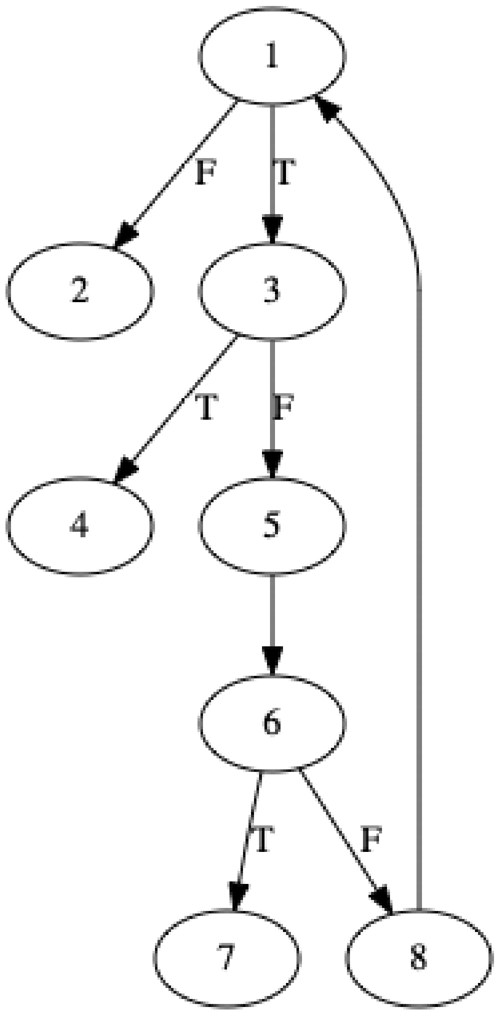
Covers all the loops and conditionals

Statement Coverage - All statements have been executed.

Branch Coverage - all edges in the control graph have been executed at least once .

Condition Coverage -all combination of conditions have been covered.

Control Flow Graph



g3.pdf

In Class Exercise

- Write the Input/Output specification for a Binary Search
- Write the functional spec
- What tests would we have?

Language Support for Testing

- ctest - Covered previously
- PyUnit, unittest, doctest
- JUnit

Testing in Python

unnecessary_math

- copy unnecessary_math.py

unittest

- copy test_um_unittest.py

doctest

- copy test_unnecessary_math.txt

```
Unnecessary_math.py
'''
Module showing how doctests can be included with source code
Each '>>>' line is run as if in a python shell, and counts as a test.
The next line, if not '>>>' is the expected output of the previous line.
If anything doesn't match exactly (including trailing spaces), the test fails.
'''

def multiply(a, b):
    """
    >>> multiply(4, 3)
    12
    >>> multiply('a', 3)
    'aaa'
    """
    return a * b
```

```
 test_um_unittest.py
import unittest
from unnecessary_math import multiply

class TestUM(unittest.TestCase):

    def setUp(self):
        pass

    def test_numbers_3_4(self):
        self.assertEqual( multiply(3,4), 12)

    def test_strings_a_3(self):
        self.assertEqual( multiply('a',3), 'aaa')

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

Run Test

```
$ python test_um_unittest.py
..
----------------------------------------------------------------------
Ran 2 tests in 0.000s

OK
```

Run Test

```
$ python -m doctest -v unnecessary_math.py
Trying:
    multiply(4, 3)
Expecting:
    12
ok
Trying:
    multiply('a', 3)
Expecting:
    'aaa'
ok
1 items had no tests:
    unnecessary_math
1 items passed all tests:
   2 tests in unnecessary_math.multiply
2 tests in 2 items.
2 passed and 0 failed.
Test passed.
```

```
test_unecessary_math.txt
This is a doctest based regression suite for unnecessary_math.py
Each '>>' line is run as if in a python shell, and counts as a test.
The next line, if not '>>' is the expected output of the previous line.
If anything doesn't match exactly (including trailing spaces), the test fails.
```

```
>>> from unnecessary_math import multiply
>>> multiply(3, 4)
12
>>> multiply('a', 3)
'aaa'
```

Run Test

```
$ python -m doctest -v test_unecessary_math.txt

Trying:
    from unnecessary_math import multiply
Expecting nothing
ok
Trying:
    multiply(3, 4)
Expecting:
    12
ok
Trying:
    multiply('a', 3)
Expecting:
    'aaa'
ok
1 items passed all tests:
   3 tests in test_unecessary_math.txt
3 tests in 1 items.
3 passed and 0 failed.
Test passed.
```

The End

by Moorthy