Unit- and Regression Testing

as Part of Modern Software Development

Dr. Axel Kohlmeyer

Associate Dean for Scientific Computing
College of Science and Technology
Temple University, Philadelphia

http://sites.google.com/site/akohlmey/

a.kohlmeyer@temple.edu

ICTP SAIFR International Centre for Theoretical Physics South American Institute for Fundamental

Workshop on Advanced Techniques in Scientific Computing

Traditional Development Cycle

- Discuss and define features for next release
- Implement features individually or in teams
- Integrate features into main code branch
- When feature complete, declare feature freeze
- Start testing new and existing features
- Document new and changed features
- Do release, if all severe problems are resolved
- Do patchlevel releases with bugfixes (only)

Testing Stages

- Unit Testing (Developers):
  → test individual components of subsystems
- Integration Testing (Developers):
  → test if multiple subsystems work together
- System Testing (Developers):
  → test if all subsystems have been integrated
  → compare system against requirements
- Acceptance Testing (Users/Client):
  → test if the entire system works as expected

Why so Much Testing?

- Early testing limits complexity of bugs:
  → bugs are eliminated early in the development
  → saves time and money
- Testing confirms that added functionality is in compliance with the specified requirements
- Unit testing encourages modular programming
  → easier to add new functionality
- Tests demonstrate correct and **incorrect** usage
- Testing is easy and can be automated; debugging is complex and requires humans

Unit Testing

- Tests for the smallest usable units of a program
→ typically a function or a class
- Write tests for all documented/expected uses
→ use multiple argument values
- Write **additional** tests for unexpected uses
→ test for correct behavior on invalid usage
- Tests should execute fast
- The amount of code written in unit tests often exceeds the amount of tested code by far

Regression Testing

- A regression is an input deck that triggers a bug or unexpected behavior in an application
- This may not be a proper use of the application; it is often engineered to trigger the bug quickly
- Then the developer fixes the bug and records the corrected behavior of the application
- The regression is then added to a regression test library with its correct output for validation
- Regression testing is part of system testing

Extreme Programming (XP)

- Software development methodology to improve software quality and customer/user interaction
- Strategy:
  - write (unit) tests and documentation first
  - add implementation later
  - feature is complete when all tests are passed
- Rationale:
  while writing tests, behavior and interface of new features are reviewed and design flaws are detected before the implementation is started

Continuous Integration (CI)

- Designated (development) branches are continuously merged, compiled, and tested against all available (unit and regression) tests
- Developer that committed the change causing test failures is responsible to resolve them
- Typically done on dedicated servers → Jenkins
- Early discovery of integration bugs, side effects
- Code base always produces a working program
- Encourages simpler and more modular code

Code Review

- Developers mututally read and discuss the changes done by their peers
- Most effective when working in pairs, often one senior and one junior developer
- Discovered problems discussed in development team, if no fast agreement on resolution
- When integrating contributed code, typically approval from two core developers and successful integration test needed to have contribution accepted into code base

Testing in Python

- `unittest` module (part of standard library) works on explicitly written unit test classes derived from a base TestCase class: methods whose name start with test are test cases; various assertions are used to compare results
- `doctest` module (part of standard library) looks for pieces of text in a class's documentation that look like interactive python sessions and repeats them and verifies that they still work as expected → regression tests

Python Example: Particle Class

```
class particle(object):

  def __init__(self,x,m=1.0):
    if float(m) <= 0.0:
      raise ValueError('Mass must be > 0.0')
    self.m = float(m)
    self.x = float(x)
    self.v = 0.0

  def __repr__(self):
    return str(self.x)+ ":"+ str(self.m)+ "@"+ str(self.v)
```

Unit Test Example: Some Tests

```
import unittest

class ParticleTest(unittest.TestCase):
    def test_constructor1(self):
        p=particle(2.0)
        self.assertEqual(p.x,2.0)
        self.assertEqual(p.m,1.0)
        self.assertEqual(p.v,0.0)

    def test_constructor2(self):
        p=particle(0.1,0.2)
        self.assertEqual(p.x,0.1)
        self.assertEqual(p.m,0.2)
        self.assertEqual(p.v,0.0)
```

Unit Test Example: More Tests

```
class ParticleTest(unittest.TestCase):
 ...
 def test_output1(self):
  p=particle(2.0)
  p.v=-1.0
  self.assertEqual(str(p),'2.0:1.0@-1.0')

 def test_assert1(self):
  with self.assertRaises(ValueError):
   particle('x')

 def test_assert5(self):
  with self.assertRaises(TypeError):
   particle(complex(1.0,-1.0),10.0)
```

Unit Test Example: Running Tests

```
[~]$ python -m unittest -v particle
test_assert1 (particle.ParticleTest) ... ok
test_assert2 (particle.ParticleTest) ... ok
test_assert3 (particle.ParticleTest) ... ok
test_assert4 (particle.ParticleTest) ... ok
test_assert5 (particle.ParticleTest) ... ok
test_assert6 (particle.ParticleTest) ... ok
test_constructor1 (particle.ParticleTest) ... ok
test_constructor2 (particle.ParticleTest) ... ok
test_constructor3 (particle.ParticleTest) ... ok
test_constructor4 (particle.ParticleTest) ... ok
test_output1 (particle.ParticleTest) ... ok
-----------------------------------------------
Ran 11 tests in 0.001s
OK
```

Unit Test Example: Test Failure

```
[~]$ python -m unittest -v harmonic
test_compute1 (harmonic.HarmonicTest) ... ok
test_compute2 (harmonic.HarmonicTest) ... ok
test_compute3 (harmonic.HarmonicTest) ... FAIL
test_constructor1 (harmonic.HarmonicTest) ... ok
test_constructor2 (harmonic.HarmonicTest) ... ok
===============================================
FAIL: test_compute3 (harmonic.HarmonicTest)
-----------------------------------------------
Traceback (most recent call last):
  File "/home/akohlmey/Downloads/unit-and-
regtest/harmonic.py", line 69, in test_compute3
    self.assertEqual(e,50.0)
AssertionError: 500.0 != 50.0


FAILED (failures=1)
```

Regression Testing with doctest

```
    def update(self):
        """
>>> osc = [particle(x=-5.0), particle(x=5.0)]
>>> print(osc)
[-5.0:1.0@0.0, 5.0:1.0@0.0]
>>> pot = harmonic(10,5)
>>> v = integrator(pot,osc,0.005)
>>> v.update()
>>> print(osc)
[-4.999375:1.0@0.24996875, 4.999375:1.0@-0.24996875]
        """

 ...

if __name__ == "__main__":
    import doctest
    doctest.testmod()
```

Test Failure with doctest

```
[~]$ python integrator.py
******************************************************
File "integrator.py", line 35, in
__main__.integrator.update
Failed example:
 print(osc)
Expected:
 [-4.999375:1.0@0.24996875, 4.999375:1.0@-0.2499687]
Got:
 [-4.999375:1.0@0.24996875, 4.999375:1.0@-0.24996875]
******************************************************
1 items had failures:
   1 of  12 in __main__.integrator.update
***Test Failed*** 1 failures.
```

Unit- and Regression Testing

- as Part of Modern Software Development

Dr. Axel Kohlmeyer

Associate Dean for Scientific Computing
College of Science and Technology
Temple University, Philadelphia

http://sites.google.com/site/akohlmey/

a.kohlmeyer@temple.edu

ICTP SAIFR International Centre for Theoretical Physics South American Institute for Fundamental

Workshop on Advanced Techniques
in Scientific Computing