Unit testing and debugging

- Objectives
  - Discover the benefits of unit testing
  - Learn how to define and run unit tests in Python
  - Take advantage of your unit test running in your IDE
  - Structure your project for unit testing
  - Test for common error conditions

Guidelines for testing

- Tests should answer the question
  - Is the program still working after the change I just made?
- This has several direct consequences
  - Each test must run fast (milliseconds, not seconds)
  - Tests should be easy to run (no complex config)
  - Tests should run everywhere
- PyCharm includes capabilities for continuous testing
  - each keystroke reruns all tests

Guidelines for testing [Feather's guidelines]

- A test is not a unit test if:
  - It talks to the database
  - It communicates across the network
  - It touches the file system
- Consider these guidelines an ideal, not hard rules

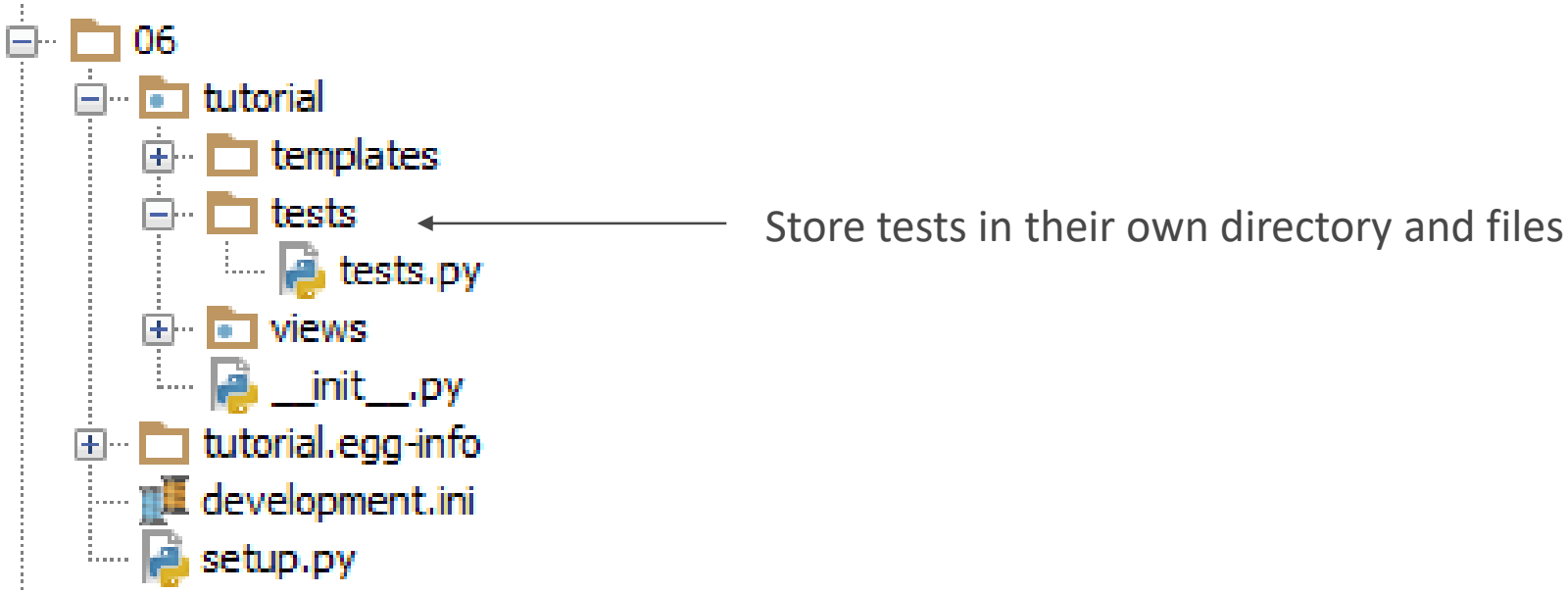
http://www.artima.com/weblogs/viewpost.jsp?thread=126923

Choosing a unit testing framework

- Python has many unit testing frameworks
  - Complete list here
- **`unittest`** is the built-in framework
  - based on Erich Gamma's JUnit and Kent Beck's Smalltalk testing framework
  - recommended framework for tests distributed with packages

Structuring projects for testing

- Separating app code and test code allows each to evolve without affecting the other
  - unit test files should match the pattern: **`test*.py`**
  - can be located in any subfolder(s)



Defining tests [classes]

- Tests are grouped into **test cases** (via classes)
  - each case is focused on testing a common set of code or features

Test cases are classes that derive from **`unittest.TestCase`**

```
import unittest

class StackTests( unittest.TestCase ):

    # shared test data

    # test 1...

    # test 2...
```

Defining tests [methods]

Each test is a single method in a **TestCase** class

```
class StackTests( unittest.TestCase ):

  def test_can_pop_items_off_stack(self):
      # arrange
      stack = Stack()
      stack.push(2)
      stack.push(3)

      # act
      res = ( stack.pop(), stack.pop() )

      #assert
      self.assertEquals(res, (3, 2) )
      self.assertEquals(stack.count, 0)
```

Test names must start with **test_**

Validation is done with one or more of the assert* methods

Tests typically follow the three A's of unit testing: **Arrange, Act, Assert**

Assertions

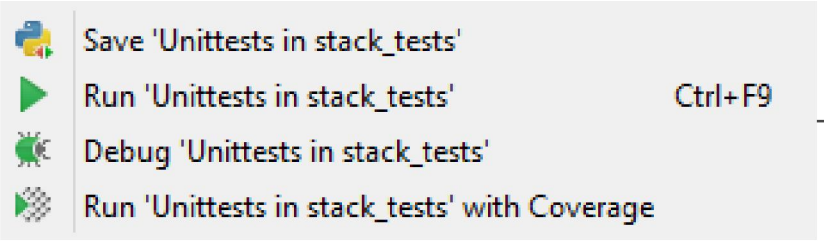
- Correctness tests are done via a set of assertion methods
  - e.g. `self.assertEqual( 42, self.count )`

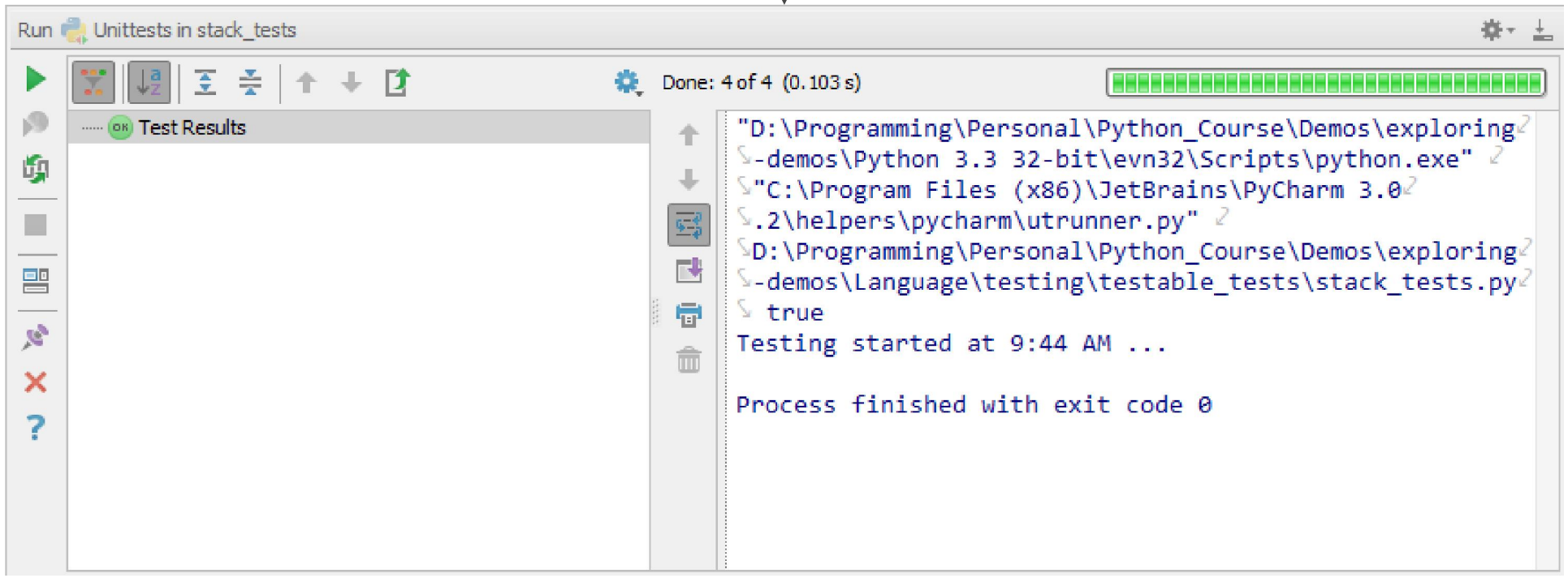
Method	Checks that
`assertEqual(a, b)`	`a == b`
`assertNotEqual(a, b)`	`a != b`
`assertTrue(x)`	`bool(x) is True`
`assertFalse(x)`	`bool(x) is False`
`assertIs(a, b)`	`a is b`
`assertIsNot(a, b)`	`a is not b`
`assertIsNone(x)`	`x is None`
`assertIsNotNone(x)`	`x is not None`
`assertIn(a, b)`	`a in b`
`assertNotIn(a, b)`	`a not in b`
`assertIsInstance(a, b)`	`isinstance(a, b)`
`assertNotIsInstance(a, b)`	`not isinstance(a, b)`

Note: a = expected value, b = actual value

Executing tests [PyCharm]

- PyCharm has excellent built-in support for testing





Summary

- Unit testing allows you to evolve your software with confidence
- Define unit tests by deriving from unittest.TestCase
- PyCharm has built-in testing and code coverage support
- Separate your test code from app code