Python Intermediate Programming

Unit 5: Software Development

CHAPTER 13: PYTHON UNIT TEST FRAMEWORK

DR. ERIC CHOU IEEE SENIOR MEMBER

Objectives

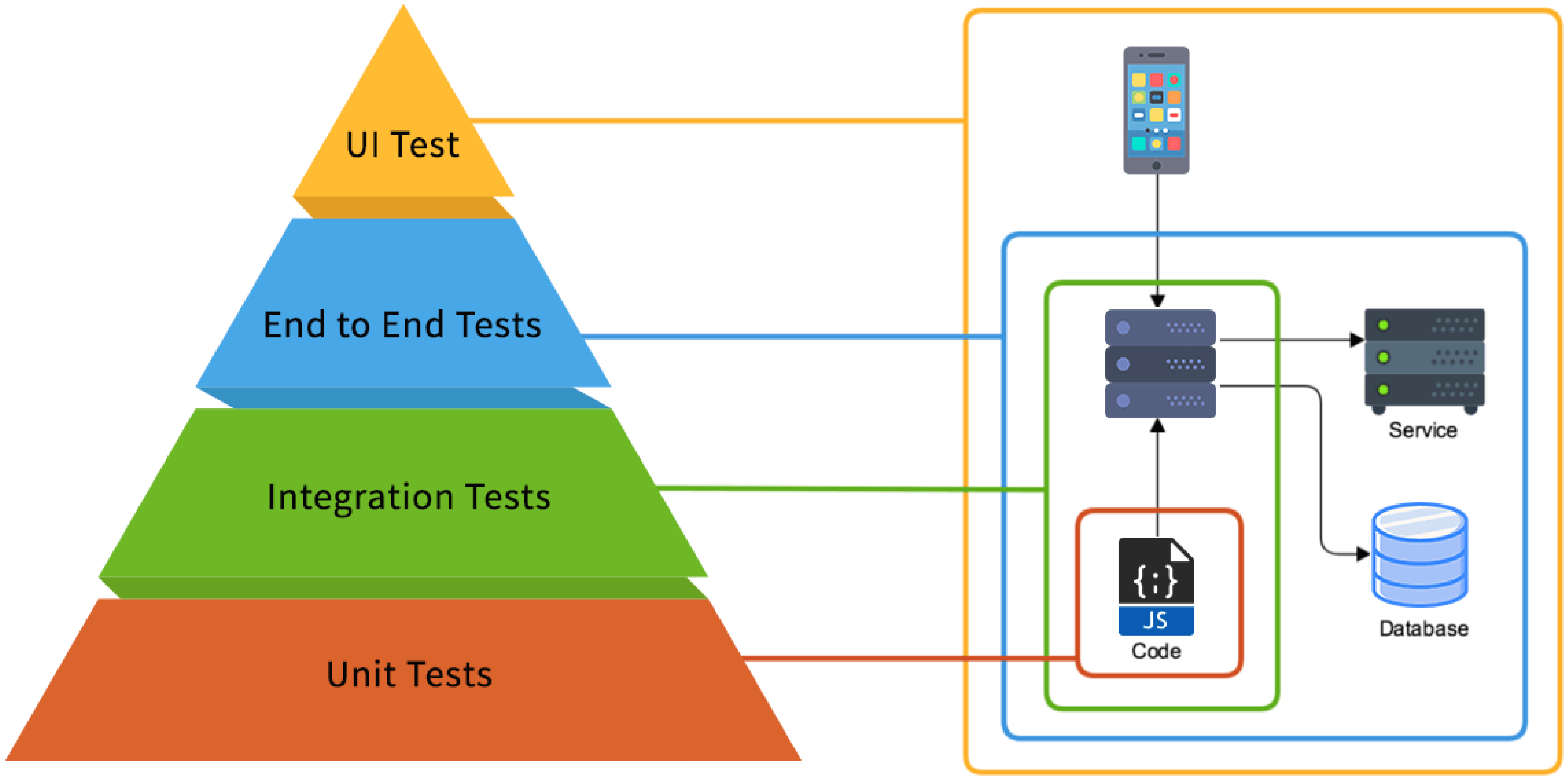
- In this lecture we will discuss testing in general, and then discuss how to perform unit testing in Python. The standard Python library supplies a module named **`unittest`**; it defines a class named **`TestCase`** from which we can create subclasses to perform unit testing.
- My driver code for testing your programs is a quick and dirty way to do unit tests. The actual **`unittest`** class is more elegant, powerful, and comprehensive, but is a more heavyweight and requires more work than the batch self-checks use when testing simple code.

Testing

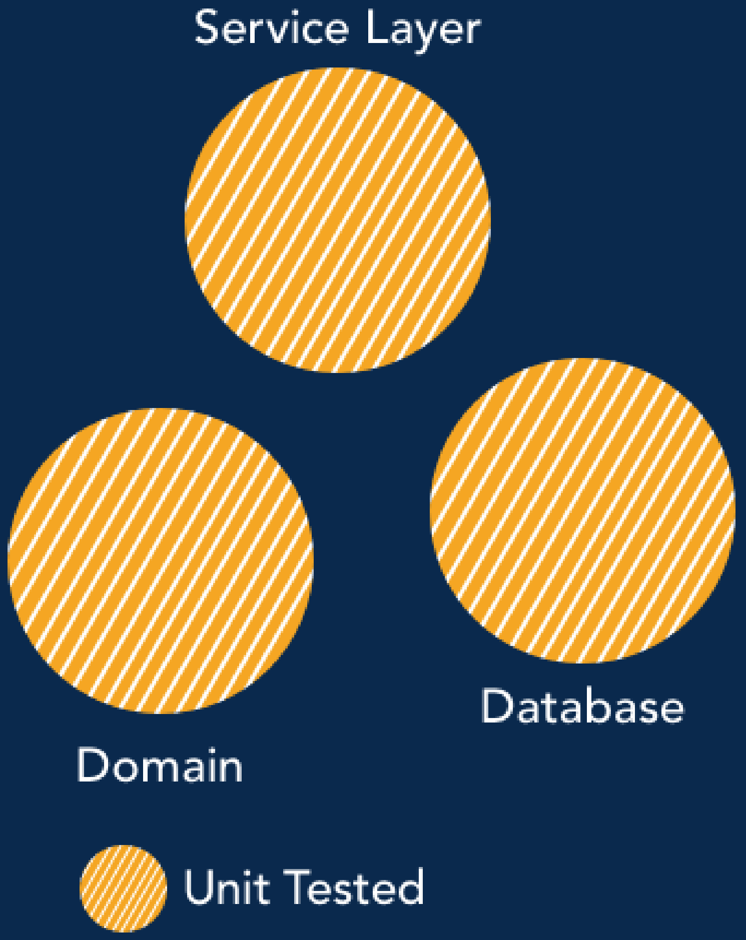
LECTURE 1

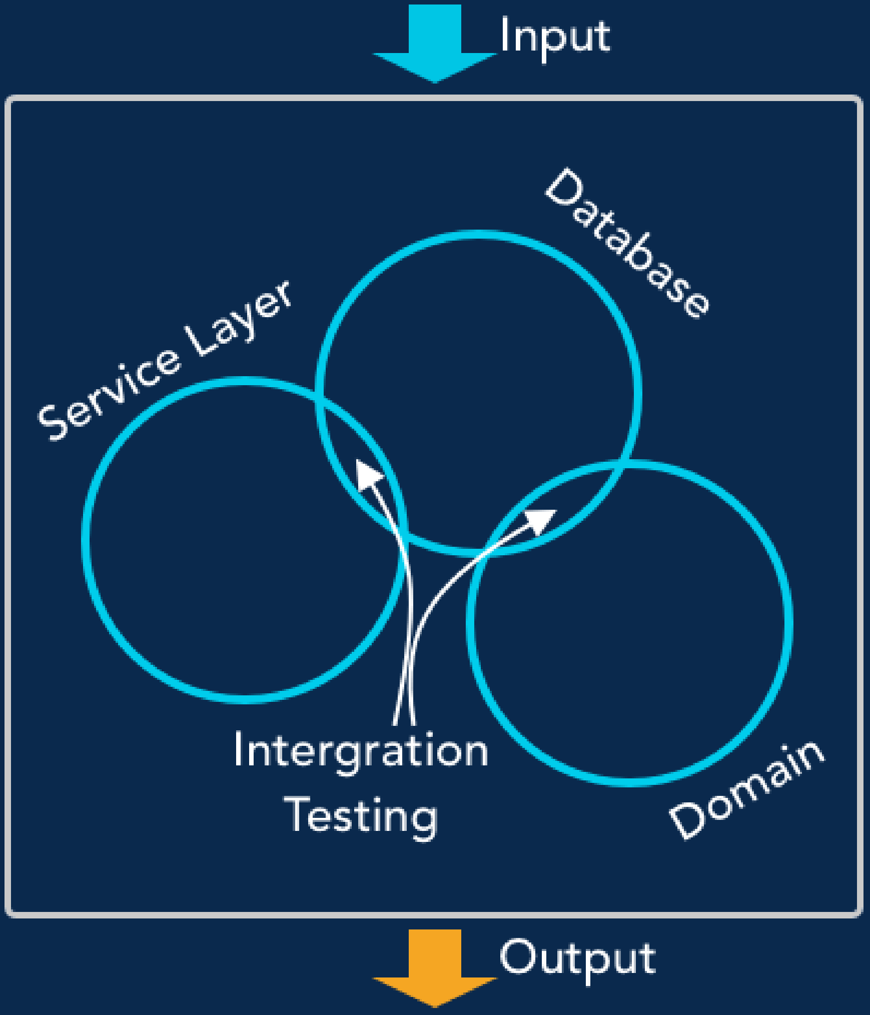
Testing

- There are many different ways of categorizing tests, and many names for subtly different styles of testing. Broadly speaking, the three categories of tests are as follow:
  - **Unit tests**
  - **Functional tests** (black box tests, acceptance tests, integration tests)
  - **Regression tests**


UI Test
End to End Tests
Integration Tests
Unit Tests
JS
Code
Service
Database

Unit Testing vs Functional Testing



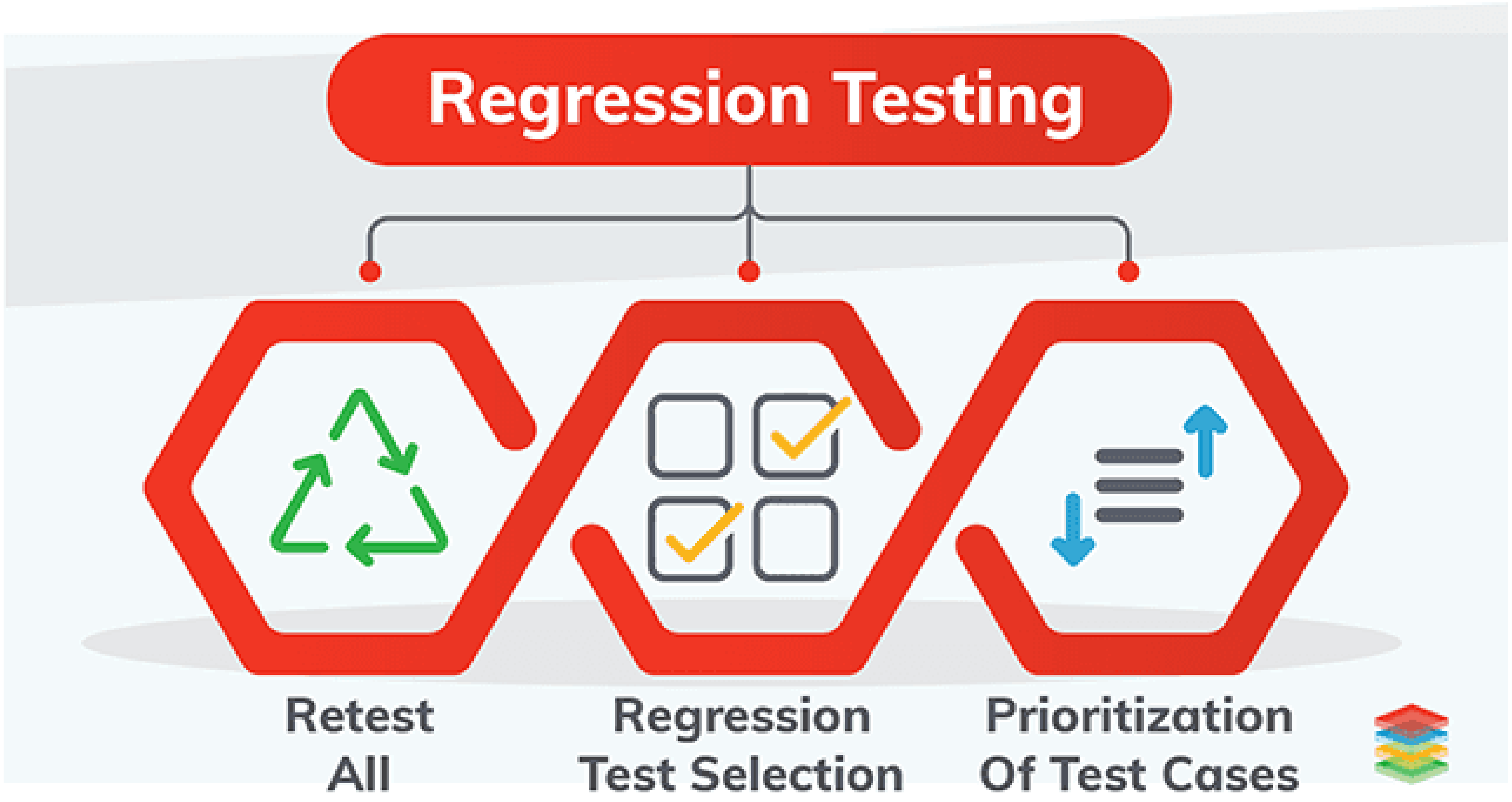


Testing

- **Unit tests** are for testing components of your code, usually individual classes or functions. The elements under test should be testable in isolation from other parts, which means eliminating dependencies. It isn't always obvious how to do this, but there are ways of handling these dependencies within your tests. Dependencies that particularly need to be managed include cases where your tests need to access external resources like databases or the filesystem. As well as looking at the basics of setting up a test framework we'll also be looking at some of the techniques you can use to control dependencies (mock objects).

Testing

- **Functional tests** are higher-level tests that drive your application from the outside. This can be done with automation tools, by your test framework, or by providing hooks within your application. Functional tests mimic user actions and test that specific input produces the right output. As well as testing the individual units of code that the tests exercise, they also check that all the parts are wired together correctly - something that unit testing alone doesn’t achieve. For some ideas about functional testing techniques with Python see my article Functional Testing of GUI Applications.


Regression Testing
Retest All
Regression Test Selection
Prioritization Of Test Cases

Testing

- **Regression testing** checks that bugs you've fixed don't recur. Regression tests are basically unit tests, but your motivation for writing them is different. Once you've identified and fixed a bug, the regression test guarantees that it doesn't come back.
- The easiest way to start with testing in Python is to use the standard library module unittest.

The unittest class

LECTURE 2

unittest module

- unittest has its origins in the Java testing framework JUnit, which itself was a port of the Smalltalk testing framework SUnit created by Kent Beck. As it is part of the xUnit family unittest is sometimes known as pyUnit.
- unittest is an object oriented framework based around test fixtures. In unittest the test fixture is the TestCase class, and its basic usage is very simple:

```
import unittest
class MyTest(unittest.TestCase):
    def testMethod(self):
        self.assertEqual(1 + 2, 3, "1 + 2 not equal to 3")
if __name__ == '__main__':
    unittest.main()
```

`unittest` module

- You create a new test fixture by subclassing TestCase and defining test methods whose names start with test. The test methods perform actions with your production classes and call assert methods to verify the expected behavior and results.

unittest module

- In large test frameworks it is common to subclass TestCase and provide methods useful for testing your specific project. Your test modules will then subclass your custom test fixture rather than directly inheriting from unittest.TestCase.
- The block at the end of the example above calls unittest.main() when the test module is executed directly from the command line:

```
python unittest1.py
```

- This executes all the tests in the test module reporting failures and errors. Test passes are shown as a '.', failures with an 'F' and errors with an 'E'.

unittest1.py

```
import unittest
class MyTest(unittest.TestCase):
     def testMethod(self):
          self.assertEqual(1 + 2, 3, "1 + 2 not equal to 3")
if __name__ == '__main__':
     unittest.main()
```

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

OK
```

Design for Test

LECTURE 3

The unittest class – Design for Test

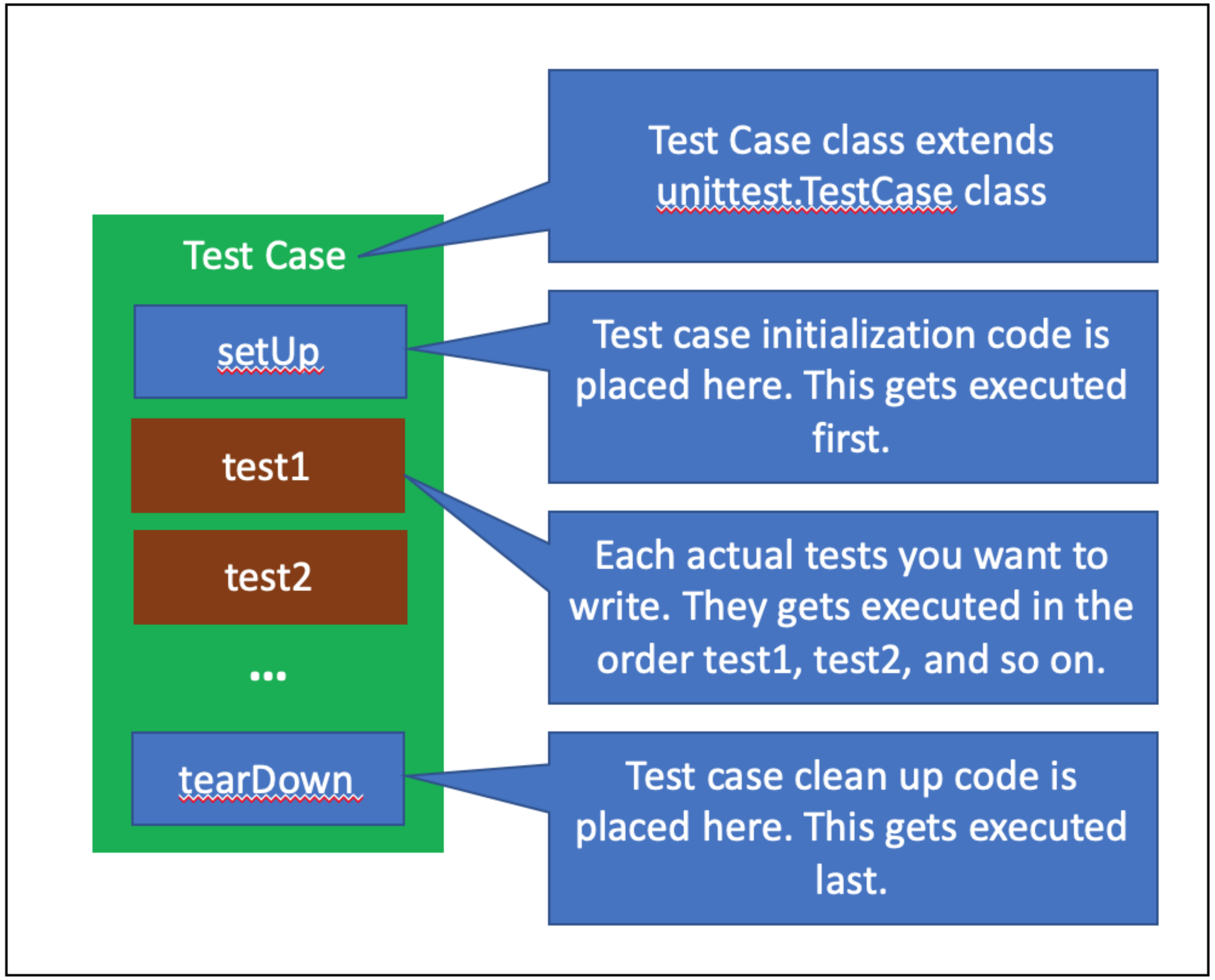
- To test software, we must write both the tests and the software. Typically, a programmer should understand the problem first, and then write the tests based on this understanding of the problem, and then write the code.
- Of course, the programmer can also write the code first, but it is better if the programmer can continually check the code he/she is writing against the suite of tests he/she has written: he/she then knows how much progress is being made towards passing all the tests.

The unittest class

- For a first simple example we will discuss testing a sort function. The function won't care what it is sorting, so we will test it on list of integers. There are two specifications that sorting functions must pass:
    1. Permutation: the sorted list has the same values as the original list
    2. Ordered   : the values in the list appear in non-decreasing order
- Why are both these specifications necessary? A function that put 0s in all positions in a list is ordered but not a permuation (so isn't sorting the list).
- A function that shuffles the values in the list (swaps them randomly) is a permutation but only rarely would it be ordered (so isn't sorting the list).

The unittest class

- While this is a bit of overkill, here is a complete class that tests the standard **`list.sort`** function. This is module sorting1.py in the download for this lecture


Test Case
setUp
test1
test2
...
tearDown
Test Case class extends unittest.TestCase class
Test case initialization code is placed here. This gets executed first.
Each actual tests you want to write. They gets executed in the order test1, test2, and so on.
Test case clean up code is placed here. This gets executed last.

sorting1.py

```
import unittest
class Sorting(unittest.TestCase):
    def setUp(self):
        self.original = [4, 1, 2, 5, 3]
        self.sorted   = list(self.original)
        list.sort(self.sorted)
    def test_order(self):
        self.assertTrue(self._is_ordered(), 'List is not in order')
    def test_permutation(self):
        self.assertCountEqual(self.original,self.sorted,
                              'List is not a permutation of the original')
    def _is_ordered(self):
        for i in range(len(self.sorted)-1):
            if self.sorted[i] > self.sorted[i+1]:
                return False
        return True
if __name__ == '__main__':
     unittest.main()
```

```
..
----------------------------------------------------------------------
Ran 2 tests in 0.015s

OK
```

The unittest class

- Here is an overview of what is happening in this module: First, we import the **`unittest`** module. Then we define the Sorting class, which is a class derived form **`unittest.TestCase`** (a class in **`unittest`**). Sorting inherits many methods, some of which (the assertXXX methods) we will discuss in more detail below.
- The standard form of a typically **`unittest`** is a **`setUp`** method (we can omit this method, but if it appears it must appear with exactly this name: it overrides a **`setUp`** method that is defined in **`TestCase`** that does nothing) followed by a series of methods whose names start with test (**`test_order`**, **`test_permutation`**).

The unittest class

- There are other special methods we can override, but don't need to for this simple example. This class also defines a helper function: **`_is_ordered`**, not starting with the word test.
- To run the test that is this class, we will right click this file (in the text editor) and select the "Run as" and then the "Python unit-test" option (instead of "Python Run" which we have always chosen before).

The unittest class

- What Python does in this case is call `unittest.main()` automatically. This function finds all the methods in the class whose names start with test and calls those methods, but first, before calling each method, it calls `setUp`. The Performance class operated similarly: it ran setup code untimed, and then it timed the real code (the specified number of times). Test can be "destructive", because `setUp` **is called before each test**.
- So, for this class it calls `setUp` and then runs `test_ordered` and then runs setup again and calls `test_permutation`. It calls the methods (and reports their results) in alphabetical order (it constructs a list of function to run and then runs them in sorted order).
- The `setUp` method creates two instance names: `self.original` which is a specific 5-list that is not ordered and `self.sorted` which is that same list; then setup calls `list.sort` on `self.sorted` to sort it.

The unittest class

- So, Python calls **`setUp`** and then the **`test_order`** method, which calls assertTrue (a method inherited by Sorting, defined in **`unittest.TestCase`**) evaluating whether the helper method **`self._is_ordred`** returns True: if so this test passes; if not the test fails. We will see how failed tests are handled soon. Then Python calls **`setUp`** again, and then the **`test_permutation`** method, which calls **`assertCountEqual`** (a method inherited by **`Sorting`**, defined in **`unittest.TestCase`**) evaluating whether its first argument has the same values, appearing the same number of times (what a permutation means) as its second argument. At this point the results of the test appear in the PU: **`PyUnit`** tab near the Console tab (typically at the bottom) or Eclipse.

The unittest class

- The console also shows some less complete testing information.
- Because the `list.sort` method is correct, both of these assertions are True.
- There are two different ways a test can fail
  1. The code raises an unexpected exception when it shouldn't (see the red x)
  2. The code fails a test (and assertion in a testing method: see the blue x)
- Note that if any test raises an unexpected exception, Python marks the test as failing and moves on to the next test (it doesn't terminate testing; the batch self-checks operate similarly).

The unittest class

Look at the picture in the unittest.pdf accompanying this lecture. The heading Sorting1 shows the result of running the test described above. Here is a key to this picture.

All the information is displayed in a Pu PyTest tab. To the right of this tab are the following icons

Show : toggle it to show all tests/only failed tests

Rerun : rerun the entire test

Error rerun: rerun only the failed tests (more focus, less time)

Stop run : stop running the current test

Ignore

History : examine recent test runs (restores appearance at end of that test)

The unittest class

- The next line indicates that it has finished all tests: 2 tests out of 2; for long tests, it will show the testing progress: 1/n, 2/n, ... n/n. Next it shows unexpected exceptions (red x) 0 and failed assertions (blue x) 0. The green line is a progress bar, showing all testing is done: it is green because all tests succeeded (it turns red if any failed).
- The next line shows the total testing time (so fast here it records 0.00). For long tests, this line will show which test it is currently performing; when testing is finished it shows the total time.
- Interesting sidenote. You can use this little timer to perform performance tests on the the sort function. You can also import cProile and profile the testing.

The unittest class

- Finally, there is a list of all the tests (sortable by any column): each line is numbered, says whether that line's test was OK or failed, names the test run, and indicates its file. Using advanced functions in unittest, it is possible to run tests in other files. Not a topic we will cover. Eclipse uses the space to the right of this information to describe failed tests (see below).
- So that is unittest in a nutshell. If you replace line 8 by self.sorted = [1, 0] and rerun the test, both the test_order and test_permutation method will fail (see it as the Sorting1 Failed picture in the .pdf). Or you can just comment-out this line and only the test_ordered method will fail.
- So be careful. If you specify the wrong answer in an assertion, the assertion fails not because the code is incorrect, but because your test is inccorrect.

The unittest class

- Notice the 2 to the right of the blue x (failed tests) and the red progress bar. In the list I have highlighted the second failed test (test_permutation) on the right it shows the line whose assertion failed (including the error message). It also tries to show the REASON for the failure (based on the assertCountEquals) by showing all the values where the counts differed (not for 0 and 1, but for 2, 3, and 4).
- Here is a table of the most useful assertions and what they test. A last string argument can be added to each, which will be printed if there is a failure).
- Note that for assertTrue/assertFalse the REASON will just say what the boolean was; but for assertEquals, if the values aren't equal, the REASON will show the both of the unequal values: generally a failed assert will try to show all relevant information/values in the error message. These are the main tools you have to check for correctness.

Assertion	Test
assertTrue(x)	bool(x) is True
assertFalse(x)	bool(x) is False
assertEqual(a, b)	a == b
assertNotEqual(a, b)	a != b
assertCountEqual(a, b)	a and b have the same elements and the same number of each, regardless of their order
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

Assertion	Test
assertIsInstance(a, b)	isinstance(a, b)
assertNotIsInstance(a, b)	not isinstance(a, b)
assertMultiLineEqual(a, b)	strings
assertSequenceEqual(a, b)	sequences, and are equal
assertListEqual(a, b)	lists, and are equal
assertTupleEqual(a, b)	tuples, and and equal
assertSetEqual(a, b)	sets/frozensets, and are equal
assertDictEqual(a, b)	dicts, and are equal
assertRegex(s, re)	regex.search(s)
assertNotRegex(s, re)	not regex.search(s)

Assertion	Test
assertAlmostEqual(a, b)	round(a-b, 7) == 0 (the same to the 7th decimal)
assertNotAlmostEqual(a, b)	round(a-b, 7) != 0
assertGreater(a, b)	a > b
assertGreaterEqual(a, b)	a >= b
assertLess(a, b)	a < b
assertLessEqual(a, b)	a <= b

The unittest class

- There is one assertion that deals with requiring an exception be raised. Calling

  ```
  assertRaises(exception,f,*args,**kargs)
  ```

  calls f(*args,**kargs) and fails if it doesn't raise the required exception.
- For example, if f('a',b) should raise the AssertionError exception, we would check it by assertRaise(AssertionError,f,'a',b).
- Also related is

  ```
  assertRaisesRegex(exception,re,f,*args,**kargs)
  ```

  which does the same thing, but also checks the exception message against the regular expression re, and also fails if there is no match. In addition, the following assertions just work on regular expressions.

The unittest class

- Before going on to a bigger example, any print functions executed in a test method appear to the right of the test when that method is selected in the **`PyUnit`** tab (with either the heading **`==ERRORS==`** or **`==CAPTURED OUTPUT==`** (if there are no errors). It is very userful to put such debugging-print statements in failing tests, to help us further understand the nature of the failure.

Loaders, runners and all that stuff

LECTURE 4

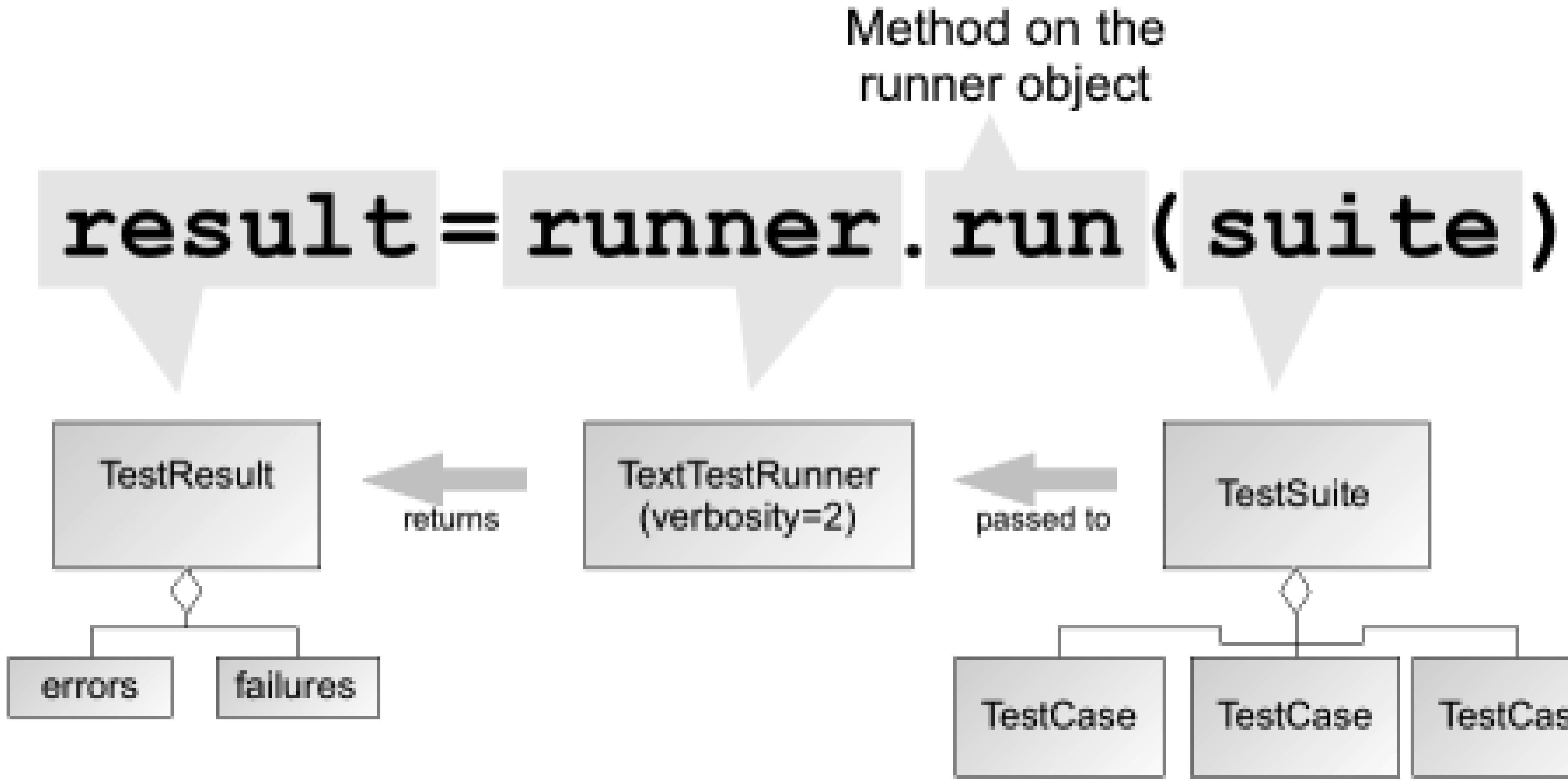
Loaders, runners and all that stuff

- There are a whole bunch of other classes in unittest; test runners, test loaders, test suites, test results and all that stuff. My book IronPython in Action has a more detailed description of how they wire together.
- Code to execute tests in multiple test modules might look like this:

```
import unittest
import test_something
import test_something2
import test_something3

loader = unittest.TestLoader()

suite = loader.loadTestsFromModule(test_something)
suite.addTests(loader.loadTestsFromModule(test_something2))
suite.addTests(loader.loadTestsFromModule(test_something3))
runner = unittest.TextTestRunner(verbosity=2)
result = runner.run(suite)
```


Method on the runner object
result = runner . run (suite)
TestResult
returns
TextTestRunner (verbosity=2)
passed to
TestSuite
errors
failures
TestCase
TestCase
TestCase

How to run test cases?

Loaders, runners and all that stuff

- The code above imports all the test modules separately **`(test_something, test_something2...)`** and turns them into a test suite before executing them with a runner. The image below shows the interactions between the classes.
- All of these classes can be subclassed to customize their behavior. Fortunately, there is also a simpler way of collecting and running all the tests in a project.

Enhanced Sorting Example (sorting2.py)

LECTURE 5

Enhanced Sorting Example

- In the enhanced version, I wrote three other "sorting" methods that fail in "interesting" ways.
- Notice the global name sorter, which is used in the class, and is bound to the sorting function we want to test. The **`test_large_scale`** method test 100 random lists, each of **`size_to_sort`**. The **`test_order/test_permutation`** now include print statements: look at the resulting output compartmentalized for each test (whether it passes or not).

sorting2.py

```
import random
import unittest
# ordered (all 0) but not a permuation
def sort_not_permutation(alist):
    for i in range(len(alist)):
        alist[i] = 0
# permutation, but not likely ordered (last 3 values not checked)
def sort_not_ordered(alist):
    for base in range(len(alist)-3):  # last 3 values unchecked
        for check in range(base+1,len(alist)):
            if alist[base] > alist[check]:
                alist[base], alist[check] = alist[check],alist[base]
    return None  # list is mutated
# raises exception
def sort_exception_sometimes(alist):
    for base in range(len(alist)): #-n for error
        for check in range(base+1,len(alist)):
            assert random.random() > .000001
            if alist[base] > alist[check]:
                alist[base], alist[check] = alist[check],alist[base]
    return None  # list is mutated

sorter       = list.sort #or any other sort above
size_to_sort = 100
```

sorting2.py

```
class Test_Sorting(unittest.TestCase):
    def setUp(self):
        self.original = [4, 1, 2, 5, 3]
        self.sorted   = list(self.original)
        sorter(self.sorted)
    def test_order(self):
        #print('Checking for order:',self.alist)
        self.assertTrue(self._is_ordered())
    def test_permutation(self):
        #print('Checking for permutation:',self.alist,'vs',self.sorted)
        self.assertCountEqual(self.original,self.sorted,
                              'List is not a permutation of the original')
    def test_large_scale(self):
        self.original  = [i for i in range(size_to_sort)]
        random.shuffle(self.original)
        for i in range(100):
            random.shuffle(self.original)
            self.sorted = list(self.original)
            sorter(self.sorted)
            self.test_order()
            self.test_permutation()
    def _is_ordered(self):
        for i in range(len(self.sorted)-1):
            if self.sorted[i] > self.sorted[i+1]:
                return False
        return True
```

sorting2.py

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

```
...
----------------------------------------------------------------------
Ran 3 tests in 0.009s

OK
```

Larger example for priority queue

LECTURE 6

PriorityQueue

- The **`courselib`** includes a class named **`PriorityQueue`**. You can read the documentation for this class. To summarize here, we can put values in a priority queue when it is constructed or by using the add function. The remove method removes the highest/largest value, so values come out from highest to lowest. The supporting methods are clear (which removes all values), peek (which returns the current highest value but doesn't remove it), **`is_empty`** (which is a boolean: True if there are no values in the priority queue, False if there is at least one), and size (the number of values in the priority queue).

PriorityQueue

- The `pq` module is a test for each of these methods in the priority queue. The logic is a bit complex (remember bigger values come out first), but this gives a more reasonable idea about how classes are tested (compared to just one function for sorting).
- The `unnittest` module had many more interesting and advanced functions: there are many more sophisticated things we can do when testing classes. This lecture is just an introduction to the topic, which is documented thoroughly in Section 26.3 of the Python online library documentation.

pq.py+priorityqueue.py+goody.py

```
from priorityqueue import PriorityQueue
import unittest, random

# Methods Tested: add, remove, clear, peek, is_empty, size
standard_size = 100
class Test_PQ(unittest.TestCase):
    def setUp(self):
        alist = [i for i in range(standard_size)]
        random.shuffle(alist)
        # puts in numbers 0 to standard_size-1
        # should come out from biggest to smallest
        self.pq  = PriorityQueue(alist)
    def test_add(self):
        # Throw out the setUp in this case; start empty
        self.pq = PriorityQueue()
        alist = [i for i in range(standard_size)]
        random.shuffle(alist)
        for i in range(len(alist)):
            self.pq.add(alist[i])
            self.assertEqual(self.pq.size(),i+1)
        # adding should be the same as setUp;
        self.test_remove()
    def test_remove(self):
        # values should come out from standard_size-1 down to 0
        for i in reversed(range(self.pq.size())):
            self.assertEqual(self.pq.remove(),i)
        # with nothing left, remove should raise an exception
        self.assertRaises(AssertionError,PriorityQueue.remove,self.pq)
```

pq.py+priorityqueue.py+goody.py

```
    def test_clear(self):
        self.pq.clear()
        # size is 0 in a cleared priority queue
        self.assertEqual(self.pq.size(), 0)
    def test_peek(self):
        # values should come out from standard_size-1 down to 0
        for i in reversed(range(self.pq.size())):
            self.assertEqual(self.pq.peek(),i)
            self.pq.remove()
        # with nothing left, remove should raise an exception
        self.assertRaises(AssertionError,PriorityQueue.peek,self.pq)
    def test_is_empty(self):
        # not empty until the last value is removed
        for _i in reversed(range(self.pq.size())):
            self.assertFalse(self.pq.is_empty())
            self.pq.remove()
        # should be be empty now; the last value has been removed
        self.assertEqual(self.pq.size(), 0)
    def test_size(self):
        size = self.pq.size()
        # self.pq.size() should get 1 smaller with each remove
        for _i in reversed(range(self.pq.size())):
            self.assertEqual(self.pq.size(),size)
            self.pq.remove()
            size -= 1

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

```
......
----------------------------------------------------------------------
Ran 6 tests in 0.007s

OK
```

Automatic test discovery

LECTURE 7

Automatic test discovery

- An easier way of running all the tests in a project is to use automatic test discovery. This is a feature that has been in alternative Python testing frameworks, such as nose and py.test, for a long time. Test discovery has finally been added to unittest in what will become Python 2.7 and Python 3.2. The test discovery has been backported as a separate module that can be used with Python 2.4 or more recent, including IronPython.
- The discover module: automatic test discovery for unittest
- When you run discover.py from the command line it searches from the current directory, recursing into Python packages that it finds, running all the test modules that it is able to import. The basic way of running test discovery is from the command line with the current directory at the top level of the project:

```
python discover.py
```

Installation of discovery.py

Project description

This is the test discovery mechanism and load_tests protocol for unittest backported from Python 2.7 to work with Python 2.4 or more recent (including Python 3).

Installation

```
pip install discover
```

Usage

- Or if the discover module is on your default module path (either in a directory pointed to by the PYTHONPATH environment variable or a directory added to the path by site.py) then you can execute it with:

```
python -m discover
```

- discover identifies test modules as importable files (inside a Python package) matching the pattern 'test*.py'. You can configure the pattern, and options like the directory discover starts its search in, with command line parameters:

```
> python -m discover -h
```

Usage

```
Usage: discover.py [options]
Options:
  -h, --help            show this help message and exit
  -v, --verbose         Verbose output
  -s START, --start-directory=START
                        Directory to start discovery ('.' default)
  -p PATTERN, --pattern=PATTERN
                        Pattern to match tests ('test*.py' default)
  -t TOP, --top-level-directory=TOP
                        Top level directory of project (defaults to start directory)
```

Testing Complex Systems

- If you want to build a more complex test framework, perhaps with a custom test runner that pushes results to a database, you can still use discovery by importing and using the **`DiscoveringTestLoader`** which is a subclass of the standard unittest **`TestLoader`**.