COMPILE-TIME UNIT TESTING IN PYTHON

JUSTIN SHAW
THE AEROSPACE CORPORATION

Abstract. Ideally unit tests will be run each time a unit is compiled. Without an explicit compilation step it becomes unclear when to execute the test code. Implicit test execution that occurs as part of the implicit compilation is the answer.

1. Introduction

Unit testing as described in *Extreme Programming Explained* by Kent Beck [1] makes sense for any piece of code that could possibly used more than once. This includes small projects. Unit testing is just a formalization of the ad-hoc testing methods that *every* programmer performs to one degree or another. This task should be made as simple as possible.

Many sophisticated testing frameworks have been developed, `PyUnit`[5], `OmPyUnit` [3], and `doctest` [4] for example. All of these frameworks serve two major functions. First they each specify the way in which the test code is written and second they provide a mechanism to execute the test code. Test execution usually requires an explicit testing step. For compiled languages it is generally not a problem to execute the tests as part of the make. This is the ideal time to execute unit tests.

As a scripting language, Python does not require an explicit compilation step. So when should unit tests be kicked off? A simple solution is to execute the test code whenever the module is run as a script. This solution is not ideal as unit tests are never garentee to be executed. Another solution is to execute the tests on `import`. This has the draw back of over-testing. I propose making a change to Python to support the automatic execution of test code during compilation. Compile-time test execution would provide many benefits.

1. Test execution would be easy to implement. It would be so simple even the smallest task could include unit testing.
2. It would be totally automated. This way the programmer needn't be concerned whether the testing for each module used was current. The tests would automatically be executed always and only when the tested code changed. (They could be run manually at the will of the programmer.)
3. It would be compatible with existing test frameworks.

First I will introduce a simple test framework modeled after Bruce Eckel's UnitTest.java [2]. The framework is kept simple to highlight how compile-time unit testing would be incorporated.

1.1. **A simple test framework.**

The goal of this framework is to be as simple to implement and execute as possible. The framework is a single function that scans the input module for functions with no arguments whose name ends in '`__test__`' and executes them.

```
##< myunittester.py def unittest(modulename):
"""Executes each function in module whose name ends in '__test__'
and takes no arguments"""
exec("import %s as module" % modulename)
for name in dir(module):
  searchString = '__test__'
  if name[-len(searchString):] == searchString:
    if type(module.__dict__[name]) == type(unittest):
      if module.__dict__[name].func_code.co_argcount == 0:
        module.__dict__[name]()
##>
```

2. The Solution

One solution that does not require a change to Python would be call unittest('mymodule') in the outer most block of each testable module. This would insure that the code would run each time the module was read in by the interpreter. This is certainly test overkill as testing continues long after the code becomes stable. We need only execute the unit test each time the code is compiled. That being the case why not have the compiler call the unit tests. This could be done very simply. We have the compiler look for a function called '__test__' in each module. When module compilation was complete a call to '__test__' would be made. The programmer would be free to adopt any test framework she desires, the test would be executed only when there was a change in the source-code, and it would be easy enough to include unit testing in the smallest of jobs. This solution may raise exception because special names are generally frowned upon, but exemptions have already been made for operator overloading and class initialization.

Here is the solution is action.

```
##< testable.py
def add(x, y):
  return x + y

def _add__test__():
  '''test is private so that it does not get tested with modules
  importing * from this one.'''
  assert add(2, 2) == 4, `add Problem`

def __test__():
  '''Called at compile-time.  Sole function is to execute
  test code'''
  import myunittester
  myunittester.unittest('testable')
##>
```

3. Possible Objections

3.1. **Test code in a separate file.** Steve Purcell [5] points out many advantages to writing test code in a file separate from the code being tested. In order to retest whenever the production code is changed, each module will have to know how to test itself. This knowledge is stored in the '__test__' function of the module. The

sole purpose of the '__test__' function is to *call* the test code. The test code does not need to be in the same file.

3.2. **Special names are ugly.** I have to admit I feel the same way and didn't like the looks of Python because of all of the double underscores. I found that they are not necessarily so bad (they are easier to type than to say). Their ease of use and automatic integration they provide with the rest of Python more than make up for their ungainly appearance. I think the same will be true for '__test__'.

4. Conclusions

No second thoughts should be given whether or not to implement a test strategy. It simply always makes sense. All hurdles that can be removed, should be. Test execution at compile-time brings testing down to its most essential element: the writing of unit tests.

References

1. Kent Beck, *Extreme programming explained*, Addison-Wesley Pub Co, 1999.
2. Bruce Eckel, *Thinking in patterns*, unpublished, www.bruceeckel.com, 2000.
3. Bob Martin, `OmPyUnit.py`, `martin@objectmentor.com`
4. Tim Peters, `doctest.py`, `tim_one@email.msn.com`
5. Steve Purcell, `PyUnit.py`, `steve@webcentric.com`

E-mail address: `Thomas.J.Shaw@aero.org`