Automated lazy testing for Python

Motivation

"Good software engineers build large test suites as they develop their code (or before) in order to ensure that they deliver bug free code. But writing tests is a time consuming task and it is always frustrating to spend significant time in development that is not visible to your user. The solution: automatically generate the tests. This is easier said than done, and Tristan Allwood has done a PhD in generating tests for Haskell programs and has also worked on automatic test suite generation for Java. This project comes from him and he would provide technical supervision for it."

This has also been my personal experience during my Industrial Placement, where I often found myself spending just as much time coming up with the unit test itself as writing the code for a particular new feature, and a great deal of this work was trivial enough to warrant automating. Hence, this project certainly would offer added value in this respect for such a use case.

As previous work done in this area has already dealt with programming languages like Haskell and Java, it would be more comprehensive to cover other types of languages. In this case, one might wish to explore a dynamic, structured language like Python, with relatively clean syntax, and a bunch of other useful features like support for introspection. Moreover, I have prior experience coding in this language from projects during my internship, so I am familiar with working in this language.

Objective

The primary goal of this project is initially set out to be an implementation of a fully automated regression test suite generator in Python, of which the ideas and optimisations herein can be further extended to other languages in future like Ruby, Javascript, or Dart.

More specifically, this means, in the context of Object-oriented programming (OOP), that given a class definition, the aim is to automatically generate a complete suite of unit test cases that will specify the behaviour of the classes under test, and could be suitable for use in incremental regression testing, as software versions evolve over time. Furthermore, this solution should also be able to report changes in behaviour across these different software versions.

Example

Given a basic standard complete implementation of `class LinkedList`, with the prototype as follows (only a sample of method signatures are mentioned below):

```
    def add(self, isAdd):
    def size(self):
    ...
```

The project aims to then create the following test suite to validate the correctness of the behaviour for this class:

```
Test #1:
   l = LinkedList()
   assert (l.size == 0)

Test #2:
   l = LinkedList()
   l.add(true)
   assert (l.size == 1)

Test #3:
   l = LinkedList()
   l.add(true)
   i = l.iterator
   assert (i.hasNext)
```

These generated test cases within the suite should generally be user friendly (maybe even close to business logic specifications as a project extension).

A list of unit testing frameworks for Python include PyUnit?, XPyUnit, TestOOB, Doctest, Nose, py.test, TwistedTrial? (according to wiki @ http://en.wikipedia.org/wiki/List_of_unit_testing_frameworks#Python). Currently, it is my intention to use `unitttest`, due to prior familiarity with it, but it still stands to evaluate the suitability of the exhaustive list of unit testing frameworks available in the wild (possibly from sources like PyPI).

Key deliverables

The end product of this project would be a Python module, created to answer (hopefully) all of the following questions posed:

1. test class constructors & methods by passing in the necessary primitive values/objects as required parameters. Of course, these entities will first be declared and instantiated in a strategic manner, to be suitable for testing, eg. not to test over the entire domain of integers, for fear of execution path explosion & meaningless results from these tests
2. Instantiating object values - possibly determine class/type of arguments and instantiate accordingly. Ideally, this should follow a lazy instantiation technique, as in Haskell and Java, where dummy objects are first passed in, and test data for return values are only invoked on demand from method calls. This remains to be explored if it is viable for a dynamic language like Python.
3. Establishing contents of assertions in unit tests, as well as values captured therein
4. Experimental evaluation - benchmarking solution against popular Python libraries to analyse and report on its performance and success of project
5. Bug detection (and diagnosis) across different versions of existing Python libraries
6. Keeping track of code coverage of classes achieved with this implementation

Outline of proposed solution 'Pyrulan'

- The main target deployment platform is Python 2.7.x, on Mac OS X 10.7 Lion 64-bit architecture, although it should be trivially portable to Python 2.x on any flavour of Linux, especially Ubuntu.
- Unit test suite generation will ideally provide high code coverage (in the entire sense of function, statement, decision/condition coverage)
- As this tool will mainly cover systematic black-box testing, target programs are unobfuscated Python bytecode (*.pyc), inspected at runtime
- It is originally intended to narrow focus to object oriented programming in Python, ie. generating unit test suites for classes in Python modules/packages
- Should be released as a library with a proper API to generate these (regression) unit test suites for Python code (and others later)
- Possible applications in running these automatically generated unit test suites, to search for changes in program behaviour, crash testing, vulnerabilities & exploits...

- Maybe when this tool can be bootstrapped until it is powerful enough to generate unit tests for itself?
- The main underpinnings to direct this project should approximately be described here, though slight modifications as the project comes along may be admitted

- It is envisaged to initially fulfill the core elements as specified in the previous sections above, then further work on project extensions in future.
- Possible extensions:
    a. Test data generation - basically a function minimisation problem according to some adequacy criterion, hence genetic search, simulated annealing, tabu search, or RANDOOP(? suggested by Tristan);

       or even fuzz testing, may be appropriate to efficiently crawl the search space (to cope with #1), and discover "good" input test datasets

b. Taking advantage of features in a dynamic language (as opposed to Irulan developed against Haskell) to improve unit test suite generation

    `eval`, monkey patching, functional programming constructs, closure, continuations, reflection/introspection (possibly with the help of tools like pyREtic as well), macros

d. Formal specification for language model to generate unit test suite (possibly to XML/JSON), so that techniques developed here to automatically generate unit tests can be reused by translating down to other languages via additional plugins
e. Appropriate visualisation of unit test suite

- This project does not intend to cover any 'white-box testing', although it would benefit greatly from it.
- ie. static program code analysis, generating unit tests from code specifications `doctest`, symbolic execution & constraint solving, boundary-value analysis (valid/invalid partition space of test data values), model-based checking `PyModel` etc.

- A quick-and-dirty hackery to demonstrate lazy instantiation (see #2) in Python can be found here @ https://github.com/evandrix/Pyrulan (`git clone` for source)
    - define:lazy instantiation - for now, it just means that the actual implementations will be swapped into the 'tests' on demand, instead of dummy implementations, like in the Java case.
- Development work will be performed in isolation, using tools like virtualenv/pip, for confident delivery upon final release of this tool.
- In order to maintain project schedule, incremental development should be adopted, by first beginning with a subset of Python (like tinyPy) and trivial Python programs to generate unit tests automatically against, then enriching these results in future iterations of this project.