Automated sassie Test

Aug. 05. 2016

Module Testing

Start with a simple module "center.py"

-> write a module test, think about design
-> pick a few other module types
-> monte carlo (home grown)
-> energy minimization (wrapped)
and on "paper" develop the template for all module tests (API)

As a module test is developed and passes review, then one can begin the re-factoring of the 1.0 module to the 2.0 API with an initial test already in place.

As each module is re-factored into 2.0 then we need to finalize the features that have been on the "to-do" list for that module via trac tickets. Tests will need to be modified.

Testing codes

- Bug handling during developments
- Type of testing

  - Unit testing: individual functions or methods

  - Integration testing: interactions between units

  - System testing: **module** and package level

Testing now and goal

- For sassie 1.0 branches

  - Unit testing: sasmol, some of simulate, utilities

  - Integrated testing: Some of modules that passed unit tests

  - System testing: None

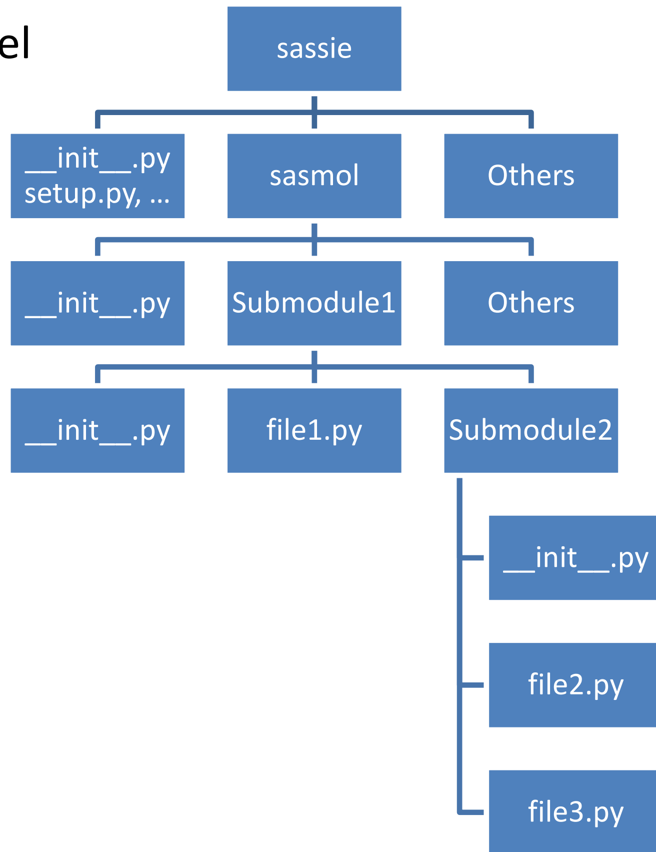
  - Input filter: Checking the input type and format correctness

- We need an automized module and package level testing.

- **What does module test do?**
- **Is there a general scheme to test in module level?**

  → Probably AI, but within sassie structure we may find a way.

Unit and integrated test

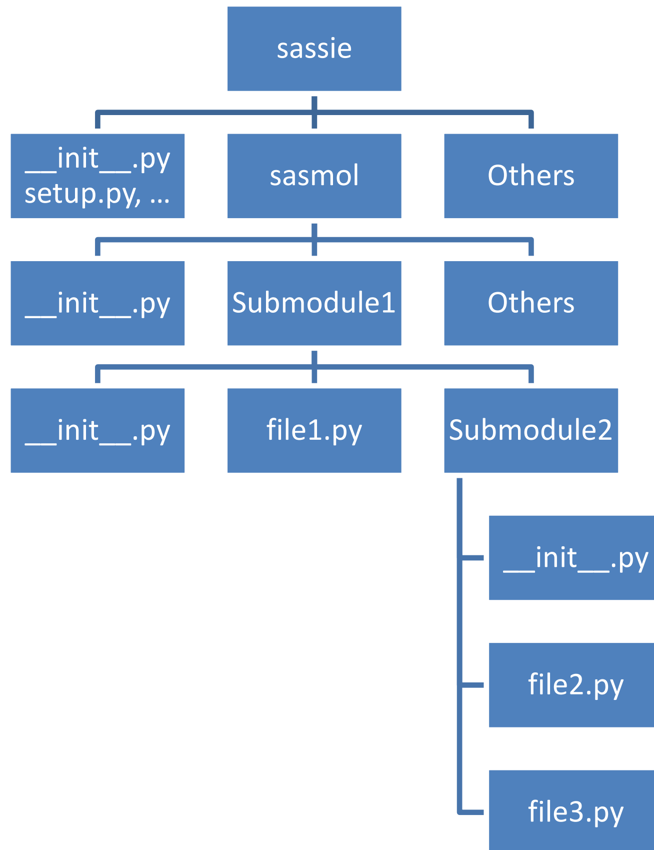


file2.py may define a module like,

```
class subclass1:
  def main():
     .....
  def method1():
     .....
  def method2():
     ......
```

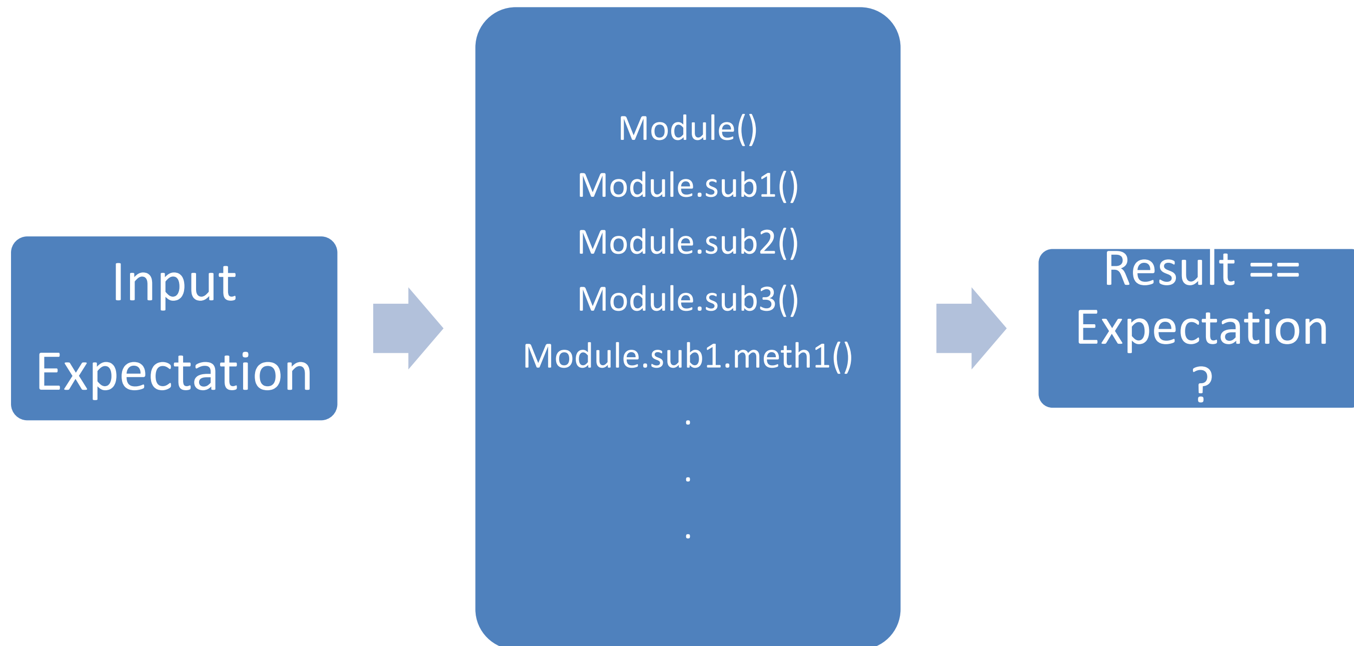
- Unit test: test method1, method2,… independently. In general, we need to consider an instance in methods.
- Integrated test: test the interaction of instances if the interaction exists.

Module test



- Within a hierarchical structure, we would like to test anything in submodule1 level.
- Then the question is what types of bugs are possible in module level?
- Can we generalize and categorize bug types in module level?
- Brainstorming…

Module testing model



- Lowest level check → **Unit test**
- If each call of functions is independent, module test means testing all possible units.
- If there exists dependence between functions, each interaction must be checked.
  → **Integrated test**
- All possible interactions within a module → **Module test**
- If a module is defined as main() in a file, main function is target of module test.
- If a module contains branch trees (if statement), work flow analysis is necessary.

Before writing a test code as a tester

- Analysis of usecase and workflow
  - usecase: statement of who, what, how?
  - workflow: how in detail
- Figure out the possible dependency (mocking if necessary)
- Coverage of unit tests as high as possible
- Idea of possible fails in the testing module even after passing unit tests
- What else?

Choice of Python test programs

- Unit testing tools: **unittest**, doctest, pytest, **nose**, testify, Trial, …
- Mock testing tools: Ludibrio, Python Mock,PyMock,mock,pMock…
- Fuzzing tools: Hypothesis, Pester,…
- Code Coverage tools: coverage, fgleaf…
- …
- We choose **unittest**, **nose**, **mocker**.

https://wiki.python.org/moin/PythonTestingToolsTaxonomy

Sassie reference:
Module template(sassie-trac)

```
import module_utilities
app = 'monte_carlo'
''' the specific name to be used is the name of the module '''
class vars():
    def __init__(self,parent = None):
    self.app = app
class simulation(): ''' the name of this class depends on the module '''
    def __init__(self,parent = None): pass
    def main(self,input_variables,txtOutput): ''' main method to manage module '''
    self.vars = vars()
    self.run_utils = module_utilities.run_utils(app,txtOutput) self.run_utils.setup_logging(self)
    self.log.debug('hello from main')
    self.unpack_variables(input_variables)
    self.run_utils.general_setup(self)
    self.initialization()
    self.monte_carlo()
    self.epilogue()
return
```

Requisites of testing

- Test request: decide the test object explicitly.
- Preparation of inputs and their expectations.
- Standardize testing directories

  core_testing/module/submodule

  core_testing/module/submodule/data

  core_testing/module/submodule/expectation
- file names

  test_unit_cases.py, test_intg_cases.py, test_modu_cases.py
- **All test will be executed by using "nosetests".**

Process of testing

1. Test request (define usecase)
2. Workflow analysis (find mock object if possible)
3. Preparation of inputs and their expectations
4. Write test code
5. Test and report results

Module Testing Template

```
import module_utilities ###see the definition of module_utilities in sassie module template###
from  unittest import TestCase ### import unittest module ###
from mocker import MockerTestCase   ### import mocker###


class  test_module(MockerTestCase):
     def variableSetup(self):
### setup test environment ###
     def setup(self):
        self.variableSetup()
     def teardown(self):
### define test case
     def test_unit_cases(self): ### include unit tests if not prepared yet###
        self.unittest.assertionMethods(result_from_unit,expectation_of_unit)
     def test_intg_cases(self): ### integrated test if not exist###
        self.unittest.assertionMethods(result_from_cases,expectation_of_cases)
     def test_modu_cases(self): ### module test if not exist###
        self.unittest.assertionMethods(result_from_cases,expectation_of_cases)
```

setup and teardown (Fixture)

- Define the independent environment set for each test.
- They are forced to run before and after each test instance when we perform nosetests.
- In setup, we may define mock objects.
- Mock will be used to “imitate” an object without calling real object.
- For example, we would like to test a module that needs an external objects (function, module, program) which is not possible to run now. Then we will mock this external objects in setup and pass imaginary values to the test code.
- They make “nose” recognize a testing routine.

Mocking a module and methods

```
fake_module = self.mocker.mock() # mock a module
fake_module(module_variable) # initialize
self.mocker.result(fake_module_result) # give imaginary results
self.mocker.count(0,Num) # Maximum number of instances to call this mock object

fake_module.method(method_variables) # imitate a method in mocked module
self.mocker.result(fake_method_result)
self.mocker.count(0,Num)
```

test_modu_center.py

```
import os, sys, multiprocessing, filecmp, sassie.core_testing.util.FileCmp
import sassie.interface.input_filter as input_filter
import import center as center
from unittest import main, TestCase
from mocker import Mocker, MockerTestCase
sys.path.append('./')   #  Only for linux

class test_module_center(MockerTestCase):

  def variablesetup(self):
    module ='center'
    runname = 'run_0'
    data_path = './data/'
    self.result_path = './'+runname+'/'+module+'/'
    self.expected_path = './expectation/'+module+'/'
    svariables={}
    svariables['runname']          = (runname,'string')
    svariables['infile']           = (data_path+'input.dcd','string')
    svariables['refpdb']           = (data_path+'input.pdb','string')
    svariables['ofile']            = ('output.dcd','string')
    svariables['path']             = ('./','string')

    self.input_error,self.variables=input_filter.type_check_and_convert(svariables)
    self.txtQueue=multiprocessing.JoinableQueue()

  def setup(self):
    self.variablesetup()

  def teardown(self):
    pass

  def test_modu_input_filter(self):
    self.assertAlmostEqual(len(self.input_error),0)

  def test_modu_output1(self):
    center.center(self.variables,self.txtQueue)
    self.assertTrue(filecmp.cmp(self.expected_path+'output.dcd', self.result_path+'output.dcd'))

  def test_modu_output2(self):
    self.assertTrue(filecmp.cmp(self.expected_path+'output.dcd.minmax',self.result_path+'output.dcd.minmax'))
```

Things to do

- Decide organization of test directories
- Build up an inventory table to manage a unit, intg, module test codes (start minimize, monte_carlo)
- Replace current module test files
- Think of feedback routine with users
- Explore different test cases that need another test template
- File comparison of binary files from different machines

Python Module Structure (1.0)

module_str.txt

Line	
1	sassie.analyze
2	sassie.analyze.apbs
3	sassie.analyze.chi_helper
4	sassie.analyze.chi_square_filter
5	sassie.analyze.cube
6	sassie.analyze.density_driver
7	sassie.analyze.density_plot
8	sassie.analyze.filter_driver
9	sassie.analyze.gsct_anal
10	sassie.analyze.gsct_gen_weight
11	sassie.analyze.renorm
12	sassie.analyze.write_apbs_input
13	sassie.build
14	sassie.build.fix_pdb
15	sassie.build.header_reader

254	sassie.simulate.rigid_body.two_body_grid.foverlap
255	sassie.simulate.rigid_body.two_body_grid.poverlap
256	sassie.simulate.rigid_body.two_body_grid.two_body_grid
257	sassie.simulate.torsion_angle_md
258	sassie.simulate.torsion_angle_md.torsion_angle_md
259	sassie.simulate.torsion_angle_md.write_tamd_input
260	sassie.tools
261	sassie.tools.align
262	sassie.tools.contrast_calculator
263	sassie.tools.coor_tools
264	sassie.tools.data_interpolation
265	sassie.tools.extract_utilities
266	sassie.tools.merge_utilities
267	sassie.tools.old_merge_utilities
268	sassie.upgrade
269	sassie.util
270	sassie.util.basis_to_python
271	sassie.util.folder_management