PYTHON NOSE+ MOCK UNIT TESTS

Cody Lane

WHAT IS A UNIT TEST?

"Unit testing is a software development process in which the smallest testable parts of an application, called units, are individually and independently scrutinized for proper operation."

- http://searchsoftwarequality.techtarget.com/definition/unit-testing

WHAT IS PYTHON NOSE

- It is a unit test framework that extends the Python unit test framework making testing easier.
- It is easy to setup and get started with.
- It integrates with other Python unit testing tools like (doctest, unittest).

WHAT IS A MOCK?

- It allows you to replace parts of your system under test with mock objects and make assertions about how they have been used.

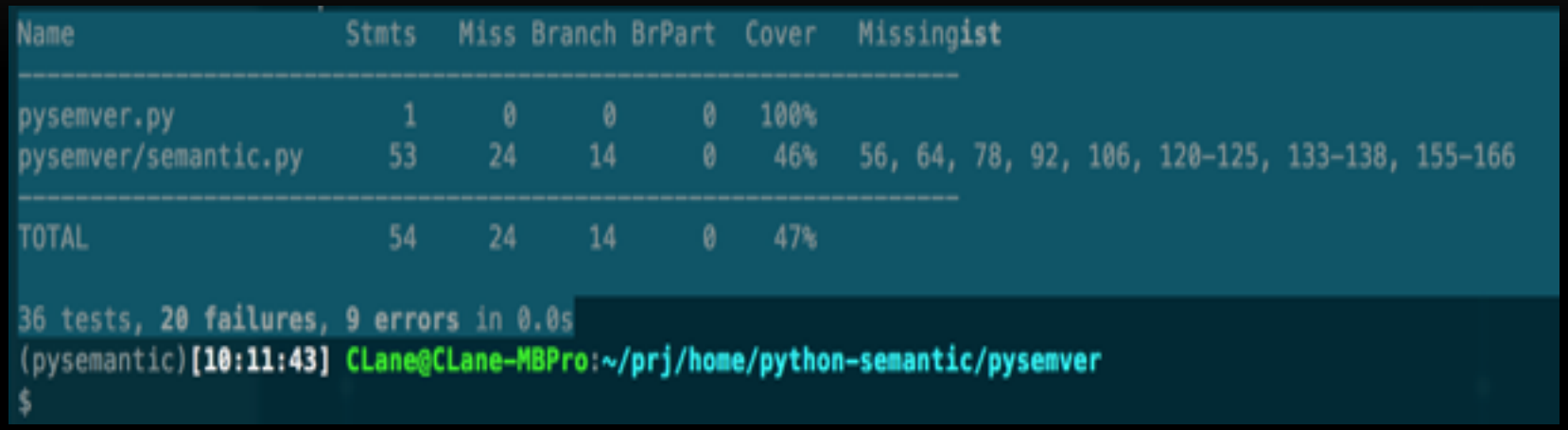
MOCK EXAMPLE

```
>>> from unittest.mock import MagicMock
>>> thing = ProductionClass()
>>> thing.method = MagicMock(return_value=3)
>>> thing.method(3, 4, 5, key='value')
3
>>> thing.method.assert_called_with(3, 4, 5, key='value')
```

WHAT IS CODE COVERAGE?

- It's an automated tool that compares a source i.e. (the code being tested) to actual tests and reports the results.

CODE COVERAGE EXAMPLE



```
Name                    Stmts   Miss Branch BrPart  Cover   Missingist
-------------------------------------------------------------------------
pysemver.py                 1      0      0      0   100%
pysemver/semantic.py       53     24     14      0    46%   56, 64, 78, 92, 106, 120-125, 133-138, 155-166
-------------------------------------------------------------------------
TOTAL                      54     24     14      0    47%

36 tests, 20 failures, 9 errors in 0.0s
(pysemantic)[10:11:43] CLane@CLane-MBPro:~/prj/home/python-semantic/pysemver
$
```

INTRODUCTION TO OUR EXAMPLE PROJECT

- We will be looking at this github.com repo found here: https://github.com/codylane/python-semantic
- The code in this project contains a small python module that compares a semantic versions like '2.3.4' < '3.0.4' and returns the result, which in this example '2.3.4' is less than version '3.0.4' so the result should be True.
- This module supports all basic equality comparisons: [<, <=, >. >=, ==, !=] for comparing semantic versions.
- For more information on Semantic check out this website: http://semver.org/
- In the next few slides, we will begin by adding new test coverage using nose and mocks (where needed) to add automated tests for our python module.

MY SETUP.CFG

```
~   1 # python-semantic/pysemver/setup.cfg
    2
    3 # nosettest setup
    4 [nosetests]
    5 verbose=3
    6 with-progressive=1
    7 logging-clear-handlers=1
    8 with-coverage=1
    9 cover-erase=1
   10 cover-inclusive=1
   11 cover-package=pysemver
   12
   13 [coverage:run]
   14 branch = True
   15 omit =
   16   */tests/*
   17   */encodings/*
   18
   19 [coveage:report]
   20 omit =
   21   */tests/*
   22   */encodings/*
   23 show_missing = True
```

PROJECT LAYOUT

```
python-semantic
├── LICENSE
├── README.md
├── pysemver
│   ├── __init__.py
│   ├── semantic.py
│   ├── setup.cfg
│   └── tests
│       ├── __init__.py
│       ├── integration
│       └── unit
│           ├── TestInvalidVersion.py
│           ├── TestVersion.py
│           └── __init__.py
├── requirements.txt
└── test.py


4 directories, 11 files
```

TESTS LAYOUT

```
cd python-semantic/pysemver/tests
touch __init__.py

mkdir -p integration
touch integration/__init__.py

mkdir -p unit
touch unit/__init__.py
```

OUR FIRST TESTS

```python
import nose.tools as nt

from pysemver import semantic

class TestInvalidVersin():
    def setup(self):
        pass

    def teardown(self):
        pass

    def test_InvalidVersion_is_Exception(self):
        '''
        Tests semantic.InvalidVersion is a base class for Exception
        '''
        nt.assert_is_instance(semantic.InvalidVersion(), Exception)

    def test_InvalidVersion_with_custom_exception_message(self):
        '''
        Tests when we raise InvalidVersion exception, that we get a custom
        error message.
        '''
        msg = semantic.InvalidVersion('my custom message')

        nt.eq_(msg.message, 'my custom message')
```

RUN NOSETESTS

```
~/prj/home/python-semantic/pysemver
$ nosetests tests/unit/TestInvalidVersion.py
Name                    Stmts   Miss Branch BrPart  Cover
Missingtion_message
----------------------------------------------------------------
pysemver.py                 1      0      0      0   100%
pysemver/semantic.py       53     36     14      0    25%   11, 18, 25,
32, 41-50, 56, 64, 78, 92, 106, 120-125, 133-138, 155-166
----------------------------------------------------------------
TOTAL                      54     36     14      0    26%

OK!  2 tests, 0 failures, 0 errors in 0.0s
```

NOSETESTS WITH AUTOMATIC DISCOVERY

- As long as we following the unittest method of defining our tests with a prefix of ‘test’ or ‘Test’ our tests will automatically be discovered when we run ‘nosetests’ command.

```
~/prj/home/python-semantic/pysemver
$ nosetests
Name                    Stmts   Miss Branch BrPart  Cover   Missingsion_1_a_b_c
-------------------------------------------------------------------
pysemver.py                 1      0      0      0   100%
pysemver/semantic.py       53     24     14      0    46%   56, 64, 78, 92,
106, 120-125, 133-138, 155-166
-------------------------------------------------------------------
TOTAL                      54     24     14      0    47%

OK!  7 tests, 0 failures, 0 errors in 0.0s
```

HOW TO USE MOCK IN YOUR UNIT TEST

- In order to use mock in your unit test, you MUST patch the method under test in the module it is specified. See this link (where to patch) for more information.
- For example, if you wanted to make an assertion that the method 're.compile' was called in 'pysemver.semantic.Version.to_maj_min_patch' then you must patch 'pysemver.semantic.re.compile' not 're.compile'.

```
34    def to_maj_min_patch(self, version):
35        '''
36        Takes a string like 4.2.3 and converts it to
37        int(major), int(minor), int(version)
38
39        If minor is not provided '0' will be returned for this value.
40
41        If patch is not provided '0' will be returned for this value.
42
43        @returns (major, minor, version) integer tuple
44        @raises InvalidVersion if the version string is not parsable
45        '''
46        mmp_finder = re.compile('(\d+)\.?(\d+)?\.?(\d+)?')
47        matcher = mmp_finder.search(version)
48        if matcher is None:
49            raise InvalidVersion('Invalid version %s must be numeric' %(version))
50
51        major, minor, patch = matcher.groups()
52        if minor is None: minor = '0'
53        if patch is None: patch = '0'
54
55        return int(major), int(minor), int(patch)
```

THE MOCKED TEST

```
110     def test_to_maj_min_patch_invokes_re_module_methods(self, mocked_method):
111         '''
112         to_maj_min_patch uses re to return a (major, minor, patch) tuple.
113         * We want to make sure that re.compile is called with arguments
114         * We want to make sure that the the compiled regex method .search
115           is called with a version string.
116         * We want to make sure that matcher.groups() is called and returns
117           a (int, int, int)
118         '''
119         # re.compile('(\d+)\.?(\d+)?\.?(\d+)?').search('1.2.3').groups()
120         mocked_method.return_value.search.return_value.groups.return_value = (1, 2, 3)
121 
122         # the Version.__init__ method invokes to_maj_min_patch.
123         inst = semantic.Version('1.2.3')
124 
125         # assertion for re.compile('(\d+)\.?(\d+)?\.?(\d+)?')
126         mocked_method.assert_called_once_with('(\d+)\.?(\d+)?\.?(\d+)?')
127 
128         # assertion for mmp_find.search('1.2.3')
129         mocked_method.return_value.search.assert_called_once_with('1.2.3')
130 
131         # assertion for matcher.groups()
132         mocked_method.return_value.search.return_value.groups.assert_called_once_with()
133 
134         nt.eq_(inst.to_maj_min_patch('1.2.3'), (1, 2, 3))
```

WHY IS 'RETURN_VALUE' IS USED IN OUR MOCK?

- If we use ***mocked_method('some argument')*** then the internal reference counter for our mocked function ***re.compile*** increasing the call counter by 1. While using ***mocked_method('some argument')*** in our tests maybe ok in some situations, it wouldn't work for us because we are using ***'mocked_method.assert_called_once_with(….)*** defined later in our test. Our assertion ***'mocked_method.assert_called_once_with(…)*** would fail.
- We use ***mocked_method.return_value*** which simulates ***mocked_method()*** in our test but, it doesn't increment the call counter.
- As an added bonus you can nest ***return_value*** as many times as you want.
- Just make sure that the lowest returns something that mimics the desired behavior. Otherwise, you will most likely get a confusing trace back. This is because the value returned by the mocked method is actually a ***MagicMock*** object.

FAILING ASSERTION USING MOCK WITHOUT A RETURN VALUE ON MATCHER.GROUPS()

```
======================================================================
ERROR: to_maj_min_patch uses re to return a (major, minor, patch) tuple.
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/Users/CLane/.virtualenvs/pysemantic/lib/python2.7/site-packages/nose/case.py", line 197, in runTest
    self.test(*self.arg)
  File "/Users/CLane/.virtualenvs/pysemantic/lib/python2.7/site-packages/mock/mock.py", line 1305, in patched
    return func(*args, **keywargs)
  File "/Users/CLane/prj/home/python-semantic/pysemver/tests/unit/TestVersion.py", line 123, in
test_to_maj_min_patch_invokes_re_module_methods
    inst = semantic.Version('1.2.3')
  File "/Users/CLane/prj/home/python-semantic/pysemver/semantic.py", line 11, in __init__
    self._major, self._minor, self._patch = self.to_maj_min_patch(version_str)
  File "/Users/CLane/prj/home/python-semantic/pysemver/semantic.py", line 51, in to_maj_min_patch
    major, minor, patch = matcher.groups()
ValueError: need more than 0 values to unpack


----------------------------------------------------------------------
Ran 11 tests in 0.007s
```

FOUND A BUG!

```
 166     @parameterized.expand([
  167         '',
  168         'a.b.c',
+ 169         '4.5.6.7',
  170     ])
  171     @nt.raises(semantic.InvalidVersion)
S>172     def test_to_maj_min_patch_returns_raises_InvalidVersion(self,
invalid_version):
~ 173         # yes, this is a valid version, but because our constructor
calls
~ 174         # to_maj_min_patch, we require a instance first.
+ 175         inst = semantic.Version('4.5.6')
+ 176
+ 177         # This is where we expect our invalid version to be raised
+ 178         inst.to_maj_min_patch(invalid_version)
```

FOUND A BUG!

- Confirmation that the bug exists!

```
~/prj/home/python-semantic/pysemver
$ nosetests


FAIL: pysemver.tests.unit.TestVersion:TestVersion.test_to_maj_min_patch_returns_raises_InvalidVersion_2_4_5_6_7
  vim +197 /Users/CLane/.virtualenvs/pysemantic/lib/python2.7/site-packages/nose/case.py  # runTest
    self.test(*self.arg)
  vim +365 /Users/CLane/.virtualenvs/pysemantic/lib/python2.7/site-packages/nose_parameterized/parameterized.py  #
standalone_func
    return func(*(a + p.args), **p.kwargs)
  vim +67  /Users/CLane/.virtualenvs/pysemantic/lib/python2.7/site-packages/nose/tools/nontrivial.py  # newfunc
    raise AssertionError(message)
AssertionError: test_to_maj_min_patch_returns_raises_InvalidVersion() did not raise InvalidVersion
Name                  Stmts   Miss Branch BrPart  Cover   Missingreturns_tuple_2_1_2_0
-----------------------------------------------------------------
pysemver.py               1      0      0      0   100%
pysemver/semantic.py      53     24     14      0    46%   61, 69, 83, 97, 111, 125-130, 138-143, 160-171
-----------------------------------------------------------------
TOTAL                     54     24     14      0    47%


17 tests, 1 failure, 0 errors in 0.0s
```

FOUND A BUG!

- Let's fix our bug in a new branch called ***bugfix_to_maj_min_patch_raise_InvalidVersion***

```
~/prj/home/python-semantic/pysemver
$ git branch -l
  Master
```

- Create a new branch

```
~/prj/home/python-semantic/pysemver
$ git checkout -b bugfix_to_maj_min_patch_raise_InvalidVersion
Switched to a new branch 'bugfix_to_maj_min_patch_raise_InvalidVersion'
```

FOUND A BUG!

- Confirm we are in our new branch bug_to_maj_min_version

```
~/prj/home/python-semantic/pysemver
$ git branch -l
* bugfix_to_maj_min_patch_raise_InvalidVersion
  master
```

- Yup, we are in our branch as denoted by the *

FIX THE BUG WITH TDD!

- We already have failing test, so we just need a spot in ***semantic.py*** where we want to insert raising a new ***InvalidVersion*** exception. With TDD, you begin by writing the smallest test that causes the failing test to pass.
- Let's look at our existing code.

FIX THE BUG WITH TDD!

```
34      def to_maj_min_patch(self, version):
35          '''
36          Takes a string like 4.2.3 and converts it to
37          int(major), int(minor), int(version)
38
39          If minor is not provided '0' will be returned for this value.
40
41          If patch is not provided '0' will be returned for this value.
42
43          @returns (major, minor, version) integer tuple
44          @raises InvalidVersion if the version string is not parsable
45          '''
+ 46          raise InvalidVersion('version is invalid')
47          mmp_finder = re.compile('(\d+)\.?(\d+)?\.?(\d+)?')
48          matcher = mmp_finder.search(version)
49          if matcher is None:
50              raise InvalidVersion('Invalid version %s must be numeric' %(version))
51
52          major, minor, patch = matcher.groups()
53          if minor is None: minor = '0'
54          if patch is None: patch = '0'
```

FIX THE BUG WITH TDD! YIKES!

```
~/prj/home/python-semantic/pysemver/tests/unit
$ nosetests
Ran 17 tests in 0.006s
FAILED (errors=9, failures=2)
```

- Our existing tests now show 9 failures, this is because we are raising ***InvalidVersion*** every time this method is called. We will fix this, but with TDD, we start small. It is good that our tests are keeping us honest and are catching those failures. We now know that our small change has caused other dependent calls to fail.
- This is a very simple example but in the real world, having solid tests helps you find these types of regressions.

FIX THE BUG WITH TDD! LET'S ADD TO OUR EXISTING TEST.

- We are going to insert a new custom message when the ***InvalidVersion*** exception is raised so we first start by writing another failing test.

```
166     @parameterized.expand([
167         '',
168         'a.b.c',
169         '4.5.6.7',
170     ])
171     @nt.raises(semantic.InvalidVersion)
S>172   def test_to_maj_min_patch_returns_raises_InvalidVersion(self, invalid_version):
173         # yes, this is a valid version, but because our constructor calls
174         # to_maj_min_patch, we require a instance first.
175         inst = semantic.Version('4.5.6')
176
177         # This is where we expect our invalid version to be raised
~ 178         try:
+ 179             inst.to_maj_min_patch(invalid_version)
+ 180         except semantic.InvalidVersion as e:
+ 181             expected_err_msgs = [
+ 182                 'Invalid version {0} must be numeric'.format(invalid_version),
S>183                 'Invalid version {0} cannot contain more than 2 dots'.format(invalid_version)
+ 184             ]
+ 185             nt.assert_in(str(e), expected_err_msgs)
+ 186             raise
```

FIX THE BUG WITH TDD! LET'S ADD TO OUR EXISTING TEST.

- Next, we run nosetests on the command line to see if this causes additional errors.

```
Ran 17 tests in 0.006s
FAILED (errors=9, failures=2)
```

- Yay! It does not, so lets add the logic to raise a new ***InvalidVersion*** exception in ***semantic.py***.

```
$ git diff semantic.py
diff --git a/pysemver/semantic.py b/pysemver/semantic.py
index e587b0e..315f18b 100755
--- a/pysemver/semantic.py
+++ b/pysemver/semantic.py
@@ -43,6 +43,8 @@ class Version(object):
         @returns (major, minor, version) integer tuple
         @raises InvalidVersion if the version string is not parsable
         '''
+        if version.count('.') > 2:
+            raise InvalidVersion('version is invalid')
         mmp_finder = re.compile('(\d+)\.?(\d+)?\.?(\d+)?')
         matcher = mmp_finder.search(version)
         if matcher is None:
```

RUN NOSETESTS TO SEE IF WE REGRESSED

```
~/prj/home/python-semantic/pysemver
$ nosetests


FAIL:
pysemver.tests.unit.TestVersion:TestVersion.test_to_maj_min_patch_returns_raises_InvalidVersi
on_2_4_5_6_7
  vim +185 tests/unit/TestVersion.py  # test_to_maj_min_patch_returns_raises_InvalidVersion
    nt.assert_in(str(e), expected_err_msgs)
AssertionError: 'version is invalid' not found in ['Invalid version 4.5.6.7 must be numeric',
'Invalid version 4.5.6.7 cannot contain more than 2 dots']
Name                  Stmts   Miss Branch BrPart  Cover   Missingreturns_tuple_2_1_2_0
-----------------------------------------------------------------
pysemver.py               1      0      0      0   100%
pysemver/semantic.py     55     24     16      0    49%   63, 71, 85, 99, 113, 127-132,
140-145, 162-173
-----------------------------------------------------------------
TOTAL                    56     24     16      0    50%


17 tests, 1 failure, 0 errors in 0.0s
```

RUN NOSETESTS TO SEE IF WE REGRESSED

- Nice, only one failure left
- `AssertionError: 'version is invalid' not found in ['Invalid version 4.5.6.7 must be numeric', 'Invalid version 4.5.6.7 cannot contain more than 2 dots']`
- This is easy to fix, but first, let's check in our code in our branch.

~/prj/home/python-semantic/pysemver

```
$ git add semantic.py
$ git commit –m ‘Updates to semantic to_maj_min_patch to handle version that has more than 2 dots and raise InvalidVersion exception’
$ git push origin bugfix_to_maj_min_patch_raise_InvalidVersion
```

MOAR TDD TO MAKE THE FINAL FAILING TEST PASS

```
~/prj/home/python-semantic/pysemver
$ git diff semantic.py
diff --git a/pysemver/semantic.py b/pysemver/semantic.py
index 315f18b..6a60fce 100755
--- a/pysemver/semantic.py
+++ b/pysemver/semantic.py
@@ -44,7 +44,9 @@ class Version(object):
         @raises InvalidVersion if the version string is not parsable
         '''
         if version.count('.') > 2:
-            raise InvalidVersion('version is invalid')
+            raise InvalidVersion(
+                'Invalid version {0} cannot contain more than 2
dots'.format(version)
+            )
         mmp_finder = re.compile('(\d+)\.?(\d+)?\.?(\d+)?')
         matcher = mmp_finder.search(version)
         if matcher is None:
```

RUN NOSETESTS

```
~/prj/home/python-semantic/pysemver
$ nosetests
Name                    Stmts   Miss Branch BrPart  Cover
Missingreturns_tuple_2_1_2_0
----------------------------------------------------------
--
pysemver.py                 1      0      0      0   100%
pysemver/semantic.py       55     24     16      0    49%   65,
73, 87, 101, 115, 129-134, 142-147, 164-175
----------------------------------------------------------
--
TOTAL                      56     24     16      0    50%

OK!  17 tests, 0 failures, 0 errors in 0.0s
```

CHECK IN DA CODEZ

- Yay! All our defined tests pass now, lets check in our branch and merge to master.

```
$ git add semantic.py
$ git commit –m 'adding custom InvalidVersion exception to to_maj_min_patch when version has more than 2 dots in version’
$ git push origin bugfix_to_maj_min_patch_raise_InvalidVersion
```

CONCLUSION

- We now have ~ 49% unit test coverage of code for semantic.py. This is a great start but we still have more tests that we need to write.

Team Recommendations:

- You may not be able to get 100% code coverage. Why? Because of the way code coverage works and and how it determines when code is loaded and invoked. For example, code coverage in Django is difficult to reach 100%.
- Establish guide lines for your teams and best practices for checking in code. For example, make a game out of increasing code coverage, never merge a branch into your master or release branch if code coverage decreases.
- Write a failing test first for new code, then write the smallest unit to get the test to pass… Refactor. Rinse and repeat until you get the right operational result.
- Writing tests for code that does not have tests can be troublesome, but you can do it. Start small, write a failing test, make sure it fails, write a passing test, refactor… Continue writing tests. This is a highly debatable topic and just a recommendation.

WHERE TO DO FROM HERE?

- Yes, there are still a lot more tests that we need to write but I hope this demonstration help provide some examples of how to use Nose + Mock objects in your unit tests.

- **Nose Links:**
- http://pythontesting.net/framework/nose/nose-introduction/
- https://nose.readthedocs.org/en/latest/

Python mock library Links:

- https://docs.python.org/dev/library/unittest.mock.html