Specter Documentation

Release 0.1.15

John Vrbanac

September 07, 2014

Contents

Specter is a Python testing framework inspired from RSpec and Jasmine. The library was created out of a desire to have a relatively flexible Python testing framework that adopted a more code-centric approach to BDD.

Specter is open-source and is available on GitHub. We love contributions!

Note: Questions? Join us on Freenode on the #specterframework channel

CHAPTER 1

Documentation

1.1 Using Specter

1.1.1 Installation

You can download Specter from PyPI for easy installation. It is recommended that you use pip or easy_install to install the bindings:

```
pip install specter
```

or:

```
easy_install specter
```

You may consider using virtualenv or pyenv to create isolated Python environments.

1.1.2 Setup

By default, Specter looks within the current directory for a folder called "spec" which contains your test files

Example Default Test Structure:

```
Project_Folder
    -- spec
        -- submodule
            -- another.py
            -- __init__.py
        -- example.py
        -- __init__.py
```

If you do not which to use the default folder, you can specific an alternative using the command-line argument:

```
specter --search /path/to/folder
```

1.1.3 Runner

Specter allows for quick and easy execution of your tests by just calling 'specter' within your project folder:

```
specter
          ___
        _/ @@\
    ~- ( \  O/__      Specter
    ~-  \     \__)    ~~~~~~~~~~
    ~-  /      \      Keeping the Bogeyman away from your code!
    ~- /       _\
       ~~~~~~~~~

ExampleSpec
   this is a test spec
     expect 'test' to equal 'test'


------------------------
------- Summary --------
Pass             | 1
Skip             | 0
Fail             | 0
Error            | 0
Incomplete       | 0
Test Total       | 1
 - Expectations | 1

------------------------
```

Command-line Arguments

Specter is a spec-based testing library to help facilitate BDD in Python.

Argument	Description
-h, –help	Show console help
–search PATH	Specifies the search path for spec files
–no-art	Disables the ASCII art on the runner
–coverage	Enables coverage.py integration. Configure using .coveragerc
–select-module	Selects a module path to run. Ex: spec.sample.TestClass
–select-by-metadata	Selects tests to run by specifying a list of key=value pairs
–xunit-results	Output xUnit XML results into a specified file
–no-color	Disables ASCII color codes
–parallel	Activates parallel testing mode
–num-processes	Specifies the number of processes to use under parallel mode (default: 6)

1.2 Writing Specter Tests

1.2.1 Naming Rules

Most frameworks require you to start your test with a given prefix such as . Specter does not impose any prefix rules on test functions. We believe that it is better to give the developer more flexibility in naming so that their test names better describe what they are actually testing. However, while Specter does have a couple rules that should be followed.

- All helper functions should start with an underscore (_). Just as Python treats a single underscore as "protected", so does Specter.

- "before_each", "after_each", "before_all", and "after_all" are reserved for setup functions on your test suites (Specs).

1.2.2 Writing Tests

Writing a test in Specter is simple.

1. Create a class which extends Spec
2. Create a function in that class that calls expect or require once

```
from specter import Spec, expect

class SampleSpec(Spec):
    """Docstring describing the specification"""
    def it_can_create_an_object(self):
        """ Test docstring"""
        expect('something').to.equal('something')
```

1.2.3 Test Setup / Teardown

```
from specter import Spec, expect

class SampleSpec(Spec):
    """Docstring describing the specification"""

    # Called once before any tests or child Specs are called
    def before_all(self):
        pass

    # Called after all tests and child Specs have been called
    def after_all(self):
        pass

    # Called before each test
    def before_each(self):
        pass

    # Called after each test
    def after_each(self):
        pass

    def it_can_create_an_object(self):
        """ Test docstring"""
        expect('something').to.equal('something')
```

1.2.4 Nested Tests

Specter tests utilizes the concept of nested test suites. This allows for you to provide a clearer picture of what you are testing within your test suites. For those who have used Jasmine or RSpec should be relatively familiar with this concept from their implementation of Spec.

Within Specter you can create a nested test description (suite) in the form of a class that inherits from the Spec class.

```
from specter import Spec, expect

class SampleSpec(Spec):

    class OtherFunctionalityOfSample(Spec):
        """ Docstring goes here """

        def it_should_do_something(self):
            """ Test Docstring """
            expect('trace').to.equal('trace')
```

1.2.5 Test Fixtures

In Specter, a test fixture is defined as a test base class that is not treated as a runnable test specification. This allows for you to build reusable test suites through inheritance. To facilitate this, there is a decorator named "fixture" available in the spec module.

```
from specter import Spec, fixture, expect

@fixture
class ExampleTestFixture(Spec):

    def _random_helper_func(self):
        pass

    def sample_test(self):
        """This test will be on every Spec that inherits this fixture"""
        expect('something').to.equal('something')


class UsingFixture(ExampleTestFixture):

    def another_test(self):
        expect('this').not_to.equal('that')
```

```
UsingFixture
   sample test
     'something' to equal 'something'
   another test
     'this' not to equal 'that'
```

1.2.6 Test State and Inheritance

Each test spec executes its tests under a clean state that does not contain the attributes of the actual Spec class. This allows for users to not worry about conflicting with the Specter infrastructure. However, the drawback to this is that the instance of "self" within a test is not actually an instance of the type defined in your hardcoded tests. This makes calling super a little bit unconventional as you can see in the example below.

```
from specter import Spec

class FirstSpec(Spec):
    def before_all(self):
        # Do something
        pass
```

```
class SecondSpec(FirstSpec):
    def before_all(self):
        # self is actually an instance of the state object and not an instance of SecondSpec
        super(type(self), self).before_all()

        # Do something else
```

As you can see in the example, you still can inherit the attributes of your other spec classes. However, you just have to keep in mind, that "self" is actually the state object and not the actual instance of the spec.

1.2.7 Assertions / Expectations

Assertions or expectations in specter attempt to be as expressive as possible. This allows for cleaner and more expressive tests which can help with overall code-awareness and effectiveness. It is important to note that an expectation does not fast-fail the test; it will continue executing the test even if the expectation fails.

Expecations follow this flow expect [target object] [to or not_to] [comparison] [expected object]

If you were expecting a status_code object was equal to 200 you would write: expect(request.status_code).to.equal(200)

Available Comparisons

- equal(expected_object)
- be_greater_than(expected_object)
- be_less_than(expected_object)
- be_none()
- be_true()
- be_false()
- contain(expected_object)
- be_in(expected_object)
- raise_a(expected_exception_type)

Asserting a raised exception

```
expect(example_func, ['args_here']).to.raise_a(Exception)
```

Fast-fail expectations

In some cases, you need to stop the execution of a test immediately upon the failure of an expectation. With specter, we call these requirements. While they follow the same flow as expectations, the name for this action is "require".

Lets say you are writing a test that checks for valid content within a request body. You could do something like:

```
expect(request.status_code).to.equal(200)
require(request.content).not_to.be_none()
# ... continue processing content
```

Utilizing this concept can allow for better visibility into an issue when a test fails. For example, if in the given example, the request status code was 202, but the rest of the test passes, you will instantly can see the problem is with the response code and not the body of the message. This has the ability to save you quite a bit of time; especially if you are testing web APIs.

1.2.8 Data-Driven Tests

Often times you find that you need to run numerous types of data through a given test case. Rather than having to duplicate your tests a large number of times, you can utilize the concept of Data-Driven Tests. This will allow for you to subject your test cases to specified dataset.

```
from specter import DataSpec

class ExampleData(DataSpec):
    DATASET = {
        'test': {'data_val': 'sample_text'},
        'second_test': {'data_val': 'sample_text2'}
    }

    def sample_data(self, data_val):
        expect(data_val).to.equal('sample_text')
```

This dataset will produce a Spec with two tests: "sample_data_test" and "sample_data_second_test" each passed in "sample_text" under the data_val parameter.

```
Example Data
   sample data test
     "sample_text" to equal "sample_text"
   sample data second test
     "sample_text2" to equal "sample_text"
```

Note: Specter will automatically remove duplicated tests within a dataset and will inform you about how many duplications were found.

Metadata in Data-Driven

There are two different methods of adding metadata to your data-driven tests. The first method is to assign metadata to the entire set of data-driven tests.

```
from specter import DataSpec

class ExampleData(DataSpec):
    DATASET = {
        'test': {'data_val': 'sample_text'},
        'second_test': {'data_val': 'sample_text'}
    }

    @metadata(test='smoke')
    def sample_data(self, data_val):
        expect(data_val).to.equal('sample_text')
```

This will assign the metadata attributes to all tests that are generated from the decoratored instance method. The second way of assigning metadata is by creating a more complex dataset item. A complex dataset item contains two keys; args and meta.

```
from specter import DataSpec

class ExampleData(DataSpec):
    DATASET = {
        'test': {'data_val': 'sample_text'},
        'second_test': {'args': {'data_val': 'sample_text'}, 'meta': {'network': 'yes'}
    }

    def sample_data(self, data_val):
        expect(data_val).to.equal('sample_text')
```

By doing this, only the 'second_test' will contain metadata. It is important to remember that you can use this format in conjunction with standard metadata tags as mentioned above.

1.2.9 Skipping Tests

Specter provided a few different ways of skipping tests.

specter.**skip**(*reason*)
The skip decorator allows for you to always bypass a test.

Parameters **reason** – Expects a string

specter.**skip_if**(*condition*, *reason=None*)
The skip_if decorator allows for you to bypass a test given that a specific condition is met.

Parameters

- **condition** – Expects a boolean
- **reason** – Expects a string

specter.**incomplete**()
The incomplete decorator behaves much like a normal skip; however, tests that are marked as incomplete get tracked under a different metric. This allows for you to create a skeleton around all of your features and specifications, and track what tests have been written and what tests are left outstanding.

```
# Example of using the incomplete decorator
@incomplete
def it_should_do_something(self):
    pass
```

1.2.10 Adding Metadata to Tests

Specter allows for you to tag tests with metadata. The primary purpose of this is to be able to carry misc information along with your test. At some point in the future, Specter will be able to output this information for consumption and processing. However, currently, metadata information can be used to select which tests you want to run.

specter.**metadata**(***key_value_pairs*)
The metadata decorator allows for you to tag specific tests with key/value data for run-time processing or reporting. The common use case is to use metadata to tag a test as a positive or negative test type.

```
# Example of using the metadata decorator
@metadata(type='negative')
def it_shouldnt_do_something(self):
    pass
```

1.3 Parallel Testing in Specter

For those who need their tests run in a parallel processes, Specter provides a parallel mode. Parallel mode distributes all tests across multiple python processes. This is especially useful if you have a vast quantity of tests or many tests that take a long amount of time.

1.3.1 Usage

Activating parallel mode is simple:

```
specter --parallel
```

Note: Keep in mind that you can tune how many processes are spawned through the –num-processes argument.

1.3.2 Differences using the parallel runner

Reporting

One of the key differences you'll notice is that the reporting is entirely different. Due to the nature of the parallel testing, the normal verbose/pretty Specter output is really not feasible. As a result, we currently provide a simple "dot" reporter with failed output in the pretty format. This allows for us to maintain a decent level of performance for users with very large numbers of tests. In the future, we plan on having specialty reporters for the parallel runner that provide more information.

Note: The xUnit reporter is available in parallel mode as well.

State

Due to the concept of parallelism, sharing live state between tests through the class instance is very costly and quite impractical. As a result, Specter does not sync state between tests during test execution. However, each Spec provides before_all() and after_all() functions to which is called before and after test execution, so that state is carried into the tests.

1.4 Release Notes

1.4.1 Release: 0.1.15

Features and bug fixes

1. Fixing PyPI package number - gh-#43

1.4.2 Release: 0.1.14

Features and bug fixes

1. Fixed Coverage.py integration - gh-#36 gh-#40

2. Fixed coverage reporting in parallel mode - gh-#40
3. Fixed duplicated traceback information on errors - gh-#42
4. Fixed difficult to trace error messages with expected parameters - gh-#41
5. Added support for execution of specter through Coverage (i.e. coverage run -m specter)

1.4.3 Release: 0.1.13

Features and bug fixes

1. Added clean test state per suite - gh-#37 gh-#13
2. Added basic parallel testing - gh-#3
3. Fixed xUnit test class path
4. Fixed standard reporter to not be red all the time - gh-#28
5. Fixed be_in() assertion - gh-#34
6. Fixed metadata decorator not re-raising assertions - gh-#35

1.4.4 Release: 0.1.12

Features and bug fixes

1. Fixing packaging issue where it wasn't including the specter.reporting package.

1.4.5 Release: 0.1.11

Special thanks to John Wood for his contributions to this release!

Features and bug fixes

1. Fixed Jenkins unicode error - gh-#27
2. Refactored reporting system to be plugin centric - gh-#21
3. Added no-color mode for CI systems - gh-#19
4. Added xUnit output reporter - gh-#10
5. Added duplication filter on data-driven dataset items - gh-#6
6. Added console output of parameters on a failed data-driven test - gh-#2
7. Added error line indicator on tracebacks
8. Added checks and x's as pass/fail indicators

CHAPTER 2

Continuous Integration

Travis CI builds - https://travis-ci.org/jmvrbanac/Specter

Coveralls Coverage - https://coveralls.io/r/jmvrbanac/Specter?branch=master

CHAPTER 3

Tested Python Versions

- 2.7.x
- 3.3.x
- PyPy 1.9.0