Sure Documentation

Release 1.4.4

Gabriel Falcão

Mar 21, 2017

Contents

Contents:

CHAPTER 1

Introduction

Sure is a python library for python that leverages a DSL for writing assertions.

In CPython it monkey-patches the `object` type, adding some methods and properties purely for test purposes.

Any python code writen after `import sure` gains testing superpowers, so you can write assertions like this:

```
import sure


def some_bratty_function(parameter):
    raise ValueError("Me no likey {0}".format(parameter))


some_bratty_function.when.called_with("Scooby").should.throw(ValueError, "Me no likey␣
↪Scooby")
```

Let's get it started

CHAPTER 2

Getting Started

Installing

It is available in PyPi, so you can install through pip:

```
pip install sure
pip3 install sure
```

Activating

Sure is activated upon importing it, unless the environment variable `SURE_DISABLE_NEW_SYNTAX` is set to any non-falsy value. (You could just use `true`)

For test code cleaningness it's recommended to import sure only once in the `__init__.py` of your root test package.

Here is an example:

```
mymodule.py
tests/
tests/__init__.py  # this is our guy
tests/unit/__init__.py
tests/unit/test_mymodule_unit1.py
tests/functional/__init__.py
tests/functional/test_mymodule_functionality.py
```

That is unless, of course, you want to explicitly import the assertion helpers from sure in every module.

CHAPTER 3

Python version compatibility

Sure is continuously tested against python versions 2.7, 3.3, 3.4 and 3.5, but its assertion API is most likely to work anywhere. The only real big difference of sure in cpython and even other implementations such as PyPy is that the monkey-patching only happens in CPython.

You can always get around beautifully with `expect`:

```
from sure import expect

expect("this".replace("is", "at")).to.equal("that")
```

where in cpython you could do:

```
"this".replace("is", "at").should.equal("that")
```

CHAPTER 4

Disabling the monkey patching

Just export the `SURE_DISABLE_NEW_SYNTAX` environment variable before running your tests.

```
export SURE_DISABLE_NEW_SYNTAX=true
```

CHAPTER 5

How sure works

The class `sure.AssertionBuilder` creates objects capable of doing assertions. The AssertionBuilder simply arranges a vast set of possible assertions that are composed by a `source` object and a `destination` object.

Every assertion, even implicitly if implicitly like in `(2 < 3).should.be.true`, is doing a source/destination matching.

Chainability

Some specific assertion methods are chainable, it can be useful for short assertions like:

```
PERSON = {
  "name": "John",
  "facebook_info": {
    "token": "abcd"
  }
}

PERSON.should.have.key("facebook_info").being.a(dict)
```

CHAPTER 6

Monkey-patching

Lincoln Clarete has written the module [`sure/magic.py`] which I simply added to sure. The most exciting part of the story is that Lincoln exposed the code with a super clean API, it's called forbidden fruit

Why CPython-only ?

Sure uses the ctypes module to break in python protections against monkey patching.

Although ctypes might also be available in other implementations such as Jython, only the CPython provide `'ctypes.pythonapi <http://docs.python.org/library/ctypes#loading-shared-libraries>'__` the features required by sure.

CHAPTER 7

API Reference

Setup/Teardown

You might be familiar with to how the `unittest` module suggests to implement setup and teardown callbacks for your tests.

But if you prefer to define test cases as functions and use a runner like nose then *sure* can help you define and activate modular fixtures.

In *sure's* parlance, we call it a *Scenario*

Example: Setup a Flask app for testing

my_flask_app.py

```
import json
from flask import Response, Flask

webapp = Flask(__name__)

@webapp.route('/')
def index():
    data = json.dumps({'hello': 'world'})
    return Response(data, headers={'Content-Type': 'application/json'})
```

tests/scenarios.py

```
from sure import scenario
from my_flask_app import webapp

def prepare_webapp(context):
    context.server = webapp.test_client()

web_scenario = scenario(prepare_webapp)
```

tests/test_webapp.py

```
import json
from sure import scenario
from tests.scenarios import web_scenario

@web_scenario
def test_hello_world(context):
    # Given that I GET /
    response = context.server.get('/')

    # Then it should have returned a successful json response
    response.headers.should.have.key('Content-Type').being.equal('application/json')
    response.status_code.should.equal(200)

    json.loads(response.data).should.equal({'hello': 'world'})
```

Example: Multiple Setup and Teardown functions

tests/scenarios.py

```
import os
import shutil
from sure import scenario

def prepare_directories(context):
    context.root = os.path.dirname(os.path.abspath(__file__))
    context.fixture_path = os.path.join(context.root, 'input_data')
    context.result_path = os.path.join(context.root, 'output_data')
    context.directories = [
        context.fixture_path,
        context.result_path,
    ]

    for path in context.directories:
        if os.path.isdir(path):
            shutil.rmtree(path)

        os.makedirs(path)


def cleanup_directories(context):
    for path in context.directories:
        if os.path.isdir(path):
            shutil.rmtree(path)


def create_10_dummy_hex_files(context):
    for index in range(10):
        filename = os.path.join(context.fixture_path, 'dummy-{}.hex'.format(index))
        open(filename, 'wb').write(os.urandom(32).encode('hex'))


dummy_files_scenario = scenario([create_directories, create_10_dummy_hex_files],␣
↪[cleanup_directories])
```

tests/test_filesystem.py

```
import os
from tests.scenarios import dummy_files_scenario

@dummy_files_scenario
def test_files_exist(context):
    os.listdir(context.fixture_path).should.equal([
        'dummy-0.hex',
        'dummy-1.hex',
        'dummy-2.hex',
        'dummy-3.hex',
        'dummy-4.hex',
        'dummy-5.hex',
        'dummy-6.hex',
        'dummy-7.hex',
        'dummy-8.hex',
        'dummy-9.hex',
    ])
```

Number Equality

(2 + 2).should.equal(4)

```
import sure

(4).should.be.equal(2 + 2)
(7.5).should.eql(3.5 + 4)
(2).should.equal(8 / 4)

(3).shouldnt.be.equal(5)
```

.equal(float, epsilon)

```
import sure

(4.242423).should.be.equal(4.242420, epsilon=0.000005)
(4.01).should.be.eql(4.00, epsilon=0.01)
(6.3699999).should.equal(6.37, epsilon=0.001)

(4.242423).shouldnt.be.equal(4.249000, epsilon=0.000005)
```

String Equality

.should_not.be.different_of(string)

```
import sure

XML1 = '''<root>
```

```
  <a-tag with-attribute="one">AND A VALUE</a-tag>
</root>'''


XML1.should_not.be.different_of(XML1)

XML2 = '''<root>
  <a-tag with-attribute="two">AND A VALUE</a-tag>
</root>'''

XML2.should.be.different_of(XML1)
```

this will give you and output like

```
Difference:

  <root>
-   <a-tag with-attribute="one">AND A VALUE</a-tag>
?                             --
+   <a-tag with-attribute="two">AND A VALUE</a-tag>
?                            ++
  </root>'''
```

.should.equal("a string")

```
"Awesome ASSERTIONS".lower().split().should.equal(['awesome', 'assertions'])
```

String Similarity

.look_like()

```
"""

THIS IS MY loose string
""".should.look_like('this is my loose string')

"""this one is different""".should_not.look_like('this is my loose string')
```

Strings Matching Regular-Expressions

should.match()

You can also use the modifiers:

- re.DEBUG
- re.I and re.IGNORECASE
- re.M and re.MULTILINE
- re.S re.DOTALL

- re.U and re.UNICODE
- re.X and re.VERBOSE

```
import re

"SOME STRING".should.match(r'some \w+', re.I)
"FOO BAR CHUCK NORRIS".should_not.match(r'some \w+', re.M)
```

Collections and Iterables

Works with:

- Lists, Tuples, Sets
- Dicts, OrderedDicts
- Anything that implements `__iter__()` / `next()`

.equal({'a': 'collection'})

```
{'foo': 'bar'}.should.equal({'foo': 'bar'})
{'foo': 'bar'}.should.eql({'foo': 'bar'})
{'foo': 'bar'}.must.be.equal({'foo': 'bar'})
```

.contain()

`expect(collection).to.contain(item)` is a shorthand to `expect(item).to.be.within(collection)`

```
['1.2.5', '1.2.4'].should.contain('1.2.5')
'1.2.4'].should.be.within(['1.2.5', '1.2.4'])

# also works with strings
"My bucket of text".should.contain('bucket')
"life".should_not.contain('anger')
'1.2.3'.should.contain('2')
```

.should.be.empty

```
[].should.be.empty;
{}.should.be.empty;
set().should.be.empty;
"".should.be.empty;
().should.be.empty
range(0).should.be.empty;

## negate with:

[1, 2, 3].shouldnt.be.empty;
"Dummy String".shouldnt.be.empty;
"Dummy String".should_not.be.empty;
```

{number}.should.be.within(0, 10)

asserts inclusive numeric range

```
(1).should.be.within(0, 2)
(5).should.be.within(0, 10)

## negate with:

(1).shouldnt.be.within(5, 6)
```

.be.within({iterable})

asserts that a member is part of the iterable

```
"g".should.be.within("gabriel")
'name'.should.be.within({'name': 'Gabriel'})
'Lincoln'.should.be.within(['Lincoln', 'Gabriel'])

## negate with:

'Bug'.shouldnt.be.within(['Sure 1.0'])
'Bug'.should_not.be.within(['Sure 1.0'])
```

.be.none

Assert whether an object is or not `None`

```
value = None
value.should.be.none
None.should.be.none

"".should_not.be.none
(not None).should_not.be.none
```

.be.ok

Assert truthfulness:

```
from sure import this

True.should.be.ok
'truthy string'.should.be.ok
{'truthy': 'dictionary'}.should.be.ok
```

And negate truthfulness:

```
from sure import this

False.shouldnt.be.ok
''.should_not.be.ok
{}.shouldnot.be.ok
```

.have.property()

```
class Basket(object):
    fruits = ["apple", "banana"]


basket1 = Basket()

basket1.should.have.property("fruits")
```

If the programmer calls `have.property()` it returns an assertion builder of the property if it exists, so that you can chain up assertions for the property value itself.

```
class Basket(object):
    fruits = ["apple", "banana"]

basket2 = Basket()
basket2.should.have.property("fruits").which.should.be.equal(["apple", "banana"])
basket2.should.have.property("fruits").being.equal(["apple", "banana"])
basket2.should.have.property("fruits").with_value.equal(["apple", "banana"])
basket2.should.have.property("fruits").with_value.being.equal(["apple", "banana"])
```

.have.key()

```
basket3 = dict(fruits=["apple", "banana"])
basket3.should.have.key("fruits")
```

If the programmer calls `have.key()` it returns an assertion builder of the key if it exists, so that you can chain up assertions for the dictionary key value itself.

```
person = dict(name=None)
person.should.have.key("name").being.none
person.should.have.key("name").being.equal(None)
```

.have.length_of(2)

Assert the length of objects

```
[3, 4].should.have.length_of(2)

"Python".should.have.length_of(6)

{'john': 'person'}.should_not.have.length_of(2)
```

{X}.should.be.greater_than(Y) and {Y}.should.be.lower_than(X)

Assert the magnitude of objects with `{X}.should.be.greater_than(Y)` and `{Y}.should.be.lower_than(X)` as well as `{X}.should.be.greater_than_or_equal_to(Y)` and `{Y}.should.be.lower_than_or_equal_to(X)`.

```
(5).should.be.greater_than(4)
(5).should_not.be.greater_than(10)
(1).should.be.lower_than(2)
(1).should_not.be.lower_than(0)

(5).should.be.greater_than_or_equal_to(4)
(5).should_not.be.greater_than_or_equal_to(10)
(1).should.be.lower_than_or_equal_to(2)
(1).should_not.be.lower_than_or_equal_to(0)
```

Callables

callable.when.called_with(arg1, kwarg1=2).should.have.raised(Exception)

You can use this feature to assert that a callable raises an exception:

```
range.when.called_with("chuck norris").should.have.raised(TypeError)
range.when.called_with(10).should_not.throw(TypeError)
```

You can also match regular expressions with to the expected exception messages:

```
import re
range.when.called_with(10, step=20).should.have.raised(TypeError, re.compile(r'(does␣
↪not take|takes no) keyword arguments'))
range.when.called_with("chuck norris").should.have.raised(TypeError, re.compile(r
↪'(cannot be interpreted as an integer|integer end argument expected)'))
```

.should.throw(Exception)

An idiomatic alias to `.should.have.raised`.

```
range.when.called_with(10, step="20").should.throw(TypeError, "range() takes no␣
↪keyword arguments")
range.when.called_with(b"chuck norris").should.throw("range() integer end argument␣
↪expected, got str.")
```

function.when.called_with(arg1, kwarg1=2).should.return_value(value)

This is a shorthand for testing that a callable returns the expected result

```
list.when.called_with([0, 1]).should.have.returned_the_value([0, 1])
```

which equates to:

```
value = range(2)
value.should.equal([0, 1])
```

there are no differences between those 2 possibilities, use at will

.be.a('typename')

this takes a type name and checks if the class matches that name

```
import sure

{}.should.be.a('dict')
(5).should.be.an('int')

## also works with paths to modules

range(10).should.be.a('collections.Iterable')
```

.be.a(type)

this takes the class (type) itself and checks if the object is an instance of it

```
import sure
from six import PY3

if PY3:
    u"".should.be.an(str)
else:
    u"".should.be.an(unicode)
[].should.be.a(list)
```

.be.above(num) and .be.below(num)

assert the instance value above and below `num`

```
import sure

(10).should.be.below(11)
(10).should.be.above(9)
(10).should_not.be.above(11)
(10).should_not.be.below(9)
```

it(), this(), those(), these()

`.should` aliases to make your tests more idiomatic.

Whether you don't like the `object.should` syntax or you are simply not running CPython, sure still allows you to use any of the assertions above, all you need to do is wrap the object that is being compared in one of the following options: `it`, `this`, `those` and `these`.

Too long, don't read

```
from sure import it, this, those, these

(10).should.be.equal(5 + 5)
```

```
this(10).should.be.equal(5 + 5)
it(10).should.be.equal(5 + 5)
these(10).should.be.equal(5 + 5)
those(10).should.be.equal(5 + 5)
```

Every assertion returns `True` when succeeded, and if failed the AssertionError is already raised internally by sure, with a nice description of what failed to match, too.

```
from sure import it, this, those, these, expect

assert (10).should.be.equal(5 + 5)
assert this(10).should.be.equal(5 + 5)
assert it(10).should.be.equal(5 + 5)
assert these(10).should.be.equal(5 + 5)
assert those(10).should.be.equal(5 + 5)

expect(10).to.be.equal(5 + 5)
expect(10).to.not_be.equal(8)
```

(lambda: None).should.be.callable

Test if something is or not callable

```
import sure

range.should.be.callable
(lambda: None).should.be.callable;
(123).should_not.be.callable
```

Note: *you can use or not the* `assert` *keyword, sure internally already raises an appropriate* `AssertionError` *with an assertion message so that you don't have to specify your own, but you can still use* `assert` *if you find it more semantic*

Example:

```
import sure

"Name".lower().should.equal('name')

## or you can also use

assert "Name".lower().should.equal('name')

## or still

from sure import this

assert this("Name".lower()).should.equal('name')

## also without the assert

this("Name".lower()).should.equal('name')
```

Any of the examples above will raise their own `AssertionError` with a meaningful error message.

Synonyms

Sure provides you with a lot of synonyms so that you can pick the ones that makes more sense for your tests.

Note that the examples below are merely illustrative, they work not only with numbers but with any of the assertions you read early in this documentation.

Positive synonyms

```
(2 + 2).should.be.equal(4)
(2 + 2).must.be.equal(4)
(2 + 2).does.equals(4)
(2 + 2).do.equals(4)
```

Negative synonyms

```
from sure import expect

(2).should_not.be.equal(3)
(2).shouldnt.be.equal(3)
(2).doesnt.equals(3)
(2).does_not.equals(3)
(2).doesnot.equals(3)
(2).dont.equal(3)
(2).do_not.equal(3)

expect(3).to.not_be.equal(1)
```

Chain-up synonyms

Any of those synonyms work as an alias to the assertion builder:

- `be`
- `being`
- `to`
- `when`
- `have`
- `with_value`

```
from sure import expect

{"foo": 1}.must.with_value.being.equal({"foo": 1})
{"foo": 1}.does.have.key("foo").being.with_value.equal(1)
```

Equality synonyms

```
(2).should.equal(2)
(2).should.equals(2)
(2).should.eql(2)
```

Positive boolean synonyms

```
import sure
(not None).should.be.ok
(not None).should.be.truthy
(not None).should.be.true
```

Negative boolean synonyms

```
import sure
False.should.be.falsy
False.should.be.false
False.should_not.be.true
False.should_not.be.ok
None.should_not.be.true
None.should_not.be.ok
```

Differently of ruby python doesn't have open classes, but sure uses a technique involving the module `ctypes` to write directly in the private `__dict__` of in-memory objects. For more information check out the Forbidden Fruit project.

Yes, it is dangerous, non-pythonic and should not be used in production code.

Although `sure` is here to be used **ONLY** in test code, therefore it should be running in **ONLY** possible environments: your local machine or your continuous-integration server.

API Builtin Documentation

`sure.`**`assertion`**(*func*)
Extend sure with a custom assertion method.

`sure.`**`build_assertion_property`**(*name*, *is_negative*, *prop=True*)
Build assertion property

This is the assertion property which is usually patched to the built-in `object` and `NoneType`.

`sure.`**`chain`**(*func*)
Extend sure with a custom chaining method.

`sure.`**`chainproperty`**(*func*)
Extend sure with a custom chain property.

class `sure.`**`ensure`**(*msg*, **args*, ***kwargs*)
Contextmanager to ensure that the given assertion message is printed upon a raised `AssertionError` exception.

The `args` and `kwargs` are used to format the message using `format()`.

class `sure.core.`**`Anything`**
Represents any possible value. Its existence is solely for idiomatic purposes.

Add custom assertions, chains and chain properties

`sure` allows to add custom assertion methods, chain methods and chain properties.

Custom assertion methods

By default `sure` comes with a good amount of *assertion methods*. For example:

- `equals()`
- `within()`
- `contains()`

And plenty more.

However, in some cases it makes sense to add custom *assertion methods* to improve the test experience.

Let's assume you want to test your web application. Somewhere there is a `Response` class with a `return_code` property. We could do the following:

```
response = Response(...)
response.return_code.should.be.equal(200)
```

This is already quiet readable, but wouldn't it be awesome do to something like this:

```
response = Response(...)
response.should.have.return_code(200)
```

To achieve this the custom assertion methods come into play:

```
from sure import assertion

@assertion
def return_code(self, expected_return_code):
    if self.negative:
        assert expected_return_code != self.obj.return_code, \
            'Expected return code matches'
    else:
        assert expected_return_code == self.obj.return_code, \
            'Expected return code does not match'


response = Response(...)
response.should.have.return_code(200)
```

I'll admit you have to write the assertion method yourself, but the result is a great experience you don't want to miss.

Chain methods

chain methods are similar to *assertion methods*. The only difference is that the *chain methods*, as the name implies, can be chained with further chains or assertions:

```
from sure import chain

@chain
def header(self, header_name):
```

```
    # check if header name actually exists
    self.obj.headers.should.have.key(header_name)
    # return header value
    return self.obj.headers[header_name]


response = Response(200, headers={'Content-Type': 'text/python'})
response.should.have.header('Content-Type').equals('text/python')
```

Chain properties

chain properties are simple properties which are available to build an assertion. Some of the default chain properties are:

- `be`
- `to`
- `when`
- `have`
- ...

Use the `chainproperty` decorator like the following to build your own *chain*:

```
from sure import chainproperty, assertion


class Foo:
    magic = 42


@chainproperty
def having(self):
    return self


@chainproperty
def implement(self):
    return self


@assertion
def attribute(self, name):
    has_it = hasattr(self.obj, name)
    if self.negative:
        assert not has_it, 'Expected was that object {0} does not have attr {1}'.
↪format(
            self.obj, name)
    else:
        assert has_it, 'Expected was that object {0} has attr {1}'.format(
            self.obj, name)

# Build awesome assertion chains
expect(Foo).having.attribute('magic')
Foo.doesnt.implement.attribute('nomagic')
```

Use custom assertion messages with `ensure`

With the `ensure` context manager *sure* provides an easy to use way to override the `AssertionError` message raised by `sure`'s assertion methods. See the following example:

```
import sure

name = myapi.do_something_that_returns_string()

with sure.ensure('the return value actually looks like: {0}', name):
    name.should.contain('whatever')
```

In case `name` does not contain the string `whatever` it will raise an `AssertionError` exception with the message *the return value actually looks like: <NAME>* (where *<NAME>* would be the actual value of the variable `name`) instead of *sure*'s default error message in that particular case.

Only `AssertionError` exceptions are re-raised by `sure.ensure()` with the custom provided message. Every other exception will be ignored and handled as expected.

CHAPTER 8

About sure

The assertion library is 100% inspired be the awesomeness of should.js which is simple, declarative and fluent.

Sure strives to provide everything a python developer needs in an assertion:

- Assertion messages are easy to understand
- When comparing iterables the comparation is recursive and shows exactly where is the error
- Fluency: the builtin types are changed in order to provide awesome simple assertions

CHAPTER 9

Indices and tables

- genindex
- modindex
- search

S

A

B

C

E

S