Case Documentation

Release 1.1.0

Ask Solem contributors

April 09, 2016

Contents

Version 1.1.0

Web http://case.readthedocs.org/

Download http://pypi.python.org/pypi/case/

Source http://github.com/celery/case/

Keywords testing utilities, python, unittest, mock

CHAPTER 1

About

CHAPTER 2

Contents

2.1 API Reference

Release 1.1

Date April 09, 2016

2.1.1 case

Python unittest Utilities

class `case.`**`Case`**(*methodName='runTest'*)

Test Case

Subclass of `unittest.TestCase` adding convenience methods.

setup / teardown

New *`setup()`* and *`teardown()`* methods can be defined in addition to the core *`setUp()`* + *`tearDown()`* methods.

Note: If you redefine the core *`setUp()`* + *`tearDown()`* methods you must make sure `super` is called. `super` is not necessary for the lowercase versions.

Python 2.6 compatibility

This class also implements *`assertWarns()`*, *`assertWarnsRegex()`*, *`assertDictContainsSubset()`*, and *`assertItemsEqual()`* which are not available in the original Python 2.6 unittest implementation.

exception `DeprecationWarning`

Base class for warnings about deprecated features.

exception `Case.`**`PendingDeprecationWarning`**

Base class for warnings about features which will be deprecated in the future.

`Case.`**`assertDeprecated`**(**args*, ***kwds*)

`Case.`**`assertDictContainsSubset`**(*expected*, *actual*, *msg=None*)

`Case.`**`assertItemsEqual`**(*expected_seq*, *actual_seq*, *msg=None*)

`Case.`**`assertPendingDeprecation`**(**args*, ***kwds*)

`Case.`**`assertWarns`**(*expected_warning*)

Case.**assertWarnsRegex**(*expected_warning*, *expected_regex*)

Case.**setUp**()

Case.**setup**()

Case.**tearDown**()

Case.**teardown**()

case.**ContextMock**(**args*, ***kwargs*)
Mock that mocks `with` statement contexts.

class case.**MagicMock**(**args*, ***kwargs*)

class case.**Mock**(**args*, ***kwargs*)

case.**call**
A tuple for holding the results of a call to a mock, either in the form *(args, kwargs)* or *(name, args, kwargs)*.

If args or kwargs are empty then a call tuple will compare equal to a tuple without those values. This makes comparisons less verbose:

```
_Call(('name', (), {})) == ('name',)
_Call(('name', (1,), {})) == ('name', (1,))
_Call(((), {'a': 'b'})) == ({'a': 'b'},)
```

The *_Call* object provides a useful shortcut for comparing with call:

```
_Call(((1, 2), {'a': 3})) == call(1, 2, a=3)
_Call(('foo', (1, 2), {'a': 3})) == call.foo(1, 2, a=3)
```

If the _Call has no name then it will match any name.

case.**patch**(*target*, *new=sentinel.DEFAULT*, *spec=None*, *create=False*, *spec_set=None*, *autospec=None*, *new_callable=None*, ***kwargs*)
patch acts as a function decorator, class decorator or a context manager. Inside the body of the function or with statement, the *target* is patched with a *new* object. When the function/with statement exits the patch is undone.

If *new* is omitted, then the target is replaced with a *MagicMock*. If *patch* is used as a decorator and *new* is omitted, the created mock is passed in as an extra argument to the decorated function. If *patch* is used as a context manager the created mock is returned by the context manager.

target should be a string in the form *'package.module.ClassName'*. The *target* is imported and the specified object replaced with the *new* object, so the *target* must be importable from the environment you are calling *patch* from. The target is imported when the decorated function is executed, not at decoration time.

The *spec* and *spec_set* keyword arguments are passed to the *MagicMock* if patch is creating one for you.

In addition you can pass *spec=True* or *spec_set=True*, which causes patch to pass in the object being mocked as the spec/spec_set object.

new_callable allows you to specify a different class, or callable object, that will be called to create the *new* object. By default *MagicMock* is used.

A more powerful form of *spec* is *autospec*. If you set *autospec=True* then the mock with be created with a spec from the object being replaced. All attributes of the mock will also have the spec of the corresponding attribute of the object being replaced. Methods and functions being mocked will have their arguments checked and will raise a *TypeError* if they are called with the wrong signature. For mocks replacing a class, their return value (the 'instance') will have the same spec as the class.

Instead of *autospec=True* you can pass *autospec=some_object* to use an arbitrary object as the spec instead of the one being replaced.

By default *patch* will fail to replace attributes that don't exist. If you pass in *create=True*, and the attribute doesn't exist, patch will create the attribute for you when the patched function is called, and delete it again afterwards. This is useful for writing tests against attributes that your production code creates at runtime. It is off by by default because it can be dangerous. With it switched on you can write passing tests against APIs that don't actually exist!

Patch can be used as a *TestCase* class decorator. It works by decorating each test method in the class. This reduces the boilerplate code when your test methods share a common patchings set. *patch* finds tests by looking for method names that start with *patch.TEST_PREFIX*. By default this is *test*, which matches the way *unittest* finds tests. You can specify an alternative prefix by setting *patch.TEST_PREFIX*.

Patch can be used as a context manager, with the with statement. Here the patching applies to the indented block after the with statement. If you use "as" then the patched object will be bound to the name after the "as"; very useful if *patch* is creating a mock object for you.

patch takes arbitrary keyword arguments. These will be passed to the *Mock* (or *new_callable*) on construction.

patch.dict(...), *patch.multiple(...)* and *patch.object(...)* are available for alternate use-cases.

2.1.2 case.case

class `case.case.`**`CaseMixin`**
Mixin class that adds the utility methods to any unittest TestCase class.

class `case.case.`**`Case`**(*methodName='runTest'*)
Test Case

Subclass of `unittest.TestCase` adding convenience methods.

setup / teardown

New `setup()` and `teardown()` methods can be defined in addition to the core `setUp()` + `tearDown()` methods.

Note: If you redefine the core `setUp()` + `tearDown()` methods you must make sure `super` is called. `super` is not necessary for the lowercase versions.

Python 2.6 compatibility

This class also implements `assertWarns()`, `assertWarnsRegex()`, `assertDictContainsSubset()`, and `assertItemsEqual()` which are not available in the original Python 2.6 unittest implementation.

2.1.3 case.skip

`case.skip.`**`todo`**(*reason*)
Skip test flagging case as TODO.

Example:

```
@skip.todo(reason='broken test')
```

`case.skip.`**`if_darwin`**()
Skip test if running under OS X.

Example:

```
@skip.if_darwin()
```

case.skip.**unless_darwin**()
Skip test if not running under OS X.

case.skip.**if_environ**(*env_var_name*)
Skip test if environment variable `env_var_name` is defined.

Example:

```
@skip.if_environ('SKIP_SLOW_TESTS')
```

case.skip.**unless_environ**(*env_var_name*)
Skip test if environment variable `env_var_name` is undefined.

Example:

```
@skip.unless_environ('LOCALE')
```

case.skip.**if_module**(*module*, *name=None*, *import_errors=(<type 'exceptions.ImportError'>,)*)
Skip test if `module` can be imported.

Parameters

- **module** – Module to import.
- **name** – Alternative module name to use in reason.
- **import_errors** – Tuple of import errors to check for. Default is `(ImportError,)`.

Example:

```
@skip.if_module('librabbitmq')
```

case.skip.**unless_module**(*module*, *name=None*, *import_errors=(<type 'exceptions.ImportError'>,)*)
Skip test if `module` can not be imported.

Parameters

- **module** – Module to import.
- **name** – Alternative module name to use in reason.
- **import_errors** – Tuple of import errors to check for. Default is `(ImportError,)`.

Example:

```
@skip.unless_module('librabbitmq')
```

case.skip.**if_jython**()
Skip test if running under Jython.

Example:

```
@skip.if_jython()
```

case.skip.**unless_jython**()
Skip test if not running under Jython.

case.skip.**if_platform**(*platform_name*, *name=None*)
Skip test if `sys.platform` name matches `platform_name`.

Parameters

- **platform_name** – Name to match with `sys.platform`.

- **name** – Alternative name to use in reason.

Example:

```
@skip.if_platform('netbsd', name='NetBSD')
```

case.skip.**unless_platform**(*platform_name*, *name=None*)
Skip test if sys.platform name does not match platform_name.

Parameters

- **platform_name** – Name to match with sys.platform.
- **name** – Alternative name to use in reason.

Example:

```
@skip.unless_platform('netbsd', name='NetBSD')
```

case.skip.**if_pypy**(*reason=u'does not work on PyPy'*)
Skip test if running on PyPy.

Example:

```
@skip.if_pypy()
```

case.skip.**unless_pypy**(*reason=u'only applicable for PyPy'*)
Skip test if not running on PyPy.

Example:

```
@skip.unless_pypy()
```

case.skip.**if_python3**(**version*, ***kwargs*)
Skip test if Python version is 3 or later.

Example:

```
@skip.if_python3(reason='does not have buffer type')
```

case.skip.**unless_python3**(**version*, ***kwargs*)
Skip test if Python version is Python 2 or earlier.

Example:

```
@skip.unless_python3()
```

case.skip.**if_win32**()
Skip test if running under Windows.

Example:

```
@skip.if_win32()
```

case.skip.**unless_win32**()
Skip test if not running under Windows.

case.skip.**if_symbol**(*symbol*, *name=None*, *import_errors=(<type 'exceptions.AttributeError'>, <type 'exceptions.ImportError'>)*)
Skip test if symbol can be imported.

Parameters

- **module** – Symbol to import.

- **name** – Alternative symbol name to use in reason.
- **import_errors** – Tuple of import errors to check for. Default is `(AttributeError, ImportError,)`.

Example:

```
@skip.if_symbol('django.db.transaction:on_commit')
```

case.skip.**unless_symbol**(*symbol*, *name=None*, *import_errors=(<type 'exceptions.AttributeError'>, <type 'exceptions.ImportError'>)*)

Skip test if `symbol` cannot be imported.

Parameters

- **module** – Symbol to import.
- **name** – Alternative symbol name to use in reason.
- **import_errors** – Tuple of import errors to check for. Default is `(AttributeError, ImportError,)`.

Example:

```
@skip.unless_symbol('django.db.transaction:on_commit')
```

case.skip.**if_python_version_after**(**version*, ***kwargs*)

Skip test if Python version is greater or equal to `*version`.

Example:

```
# skips test if running on Python >= 3.5
@skip.if_python_version_after(3, 5)
```

case.skip.**if_python_version_before**(**version*, ***kwargs*)

Skip test if Python version is less than `*version`.

Example:

```
# skips test if running on Python < 3.1
@skip.if_python_version_before(3, 1)
```

2.1.4 case.mock

case.mock.**ContextMock**(**args*, ***kwargs*)

Mock that mocks `with` statement contexts.

class case.mock.**MagicMock**(**args*, ***kwargs*)

class case.mock.**Mock**(**args*, ***kwargs*)

class case.mock.**MockCallbacks**

case.mock.**call**

A tuple for holding the results of a call to a mock, either in the form *(args, kwargs)* or *(name, args, kwargs)*.

If args or kwargs are empty then a call tuple will compare equal to a tuple without those values. This makes comparisons less verbose:

```
_Call(('name', (), {})) == ('name',)
_Call(('name', (1,), {})) == ('name', (1,))
_Call(((), {'a': 'b'})) == ({'a': 'b'},)
```

The *_Call* object provides a useful shortcut for comparing with call:

```
_Call(((1, 2), {'a': 3})) == call(1, 2, a=3)
_Call(('foo', (1, 2), {'a': 3})) == call.foo(1, 2, a=3)
```

If the _Call has no name then it will match any name.

case.mock.**patch**(*target*, *new=sentinel.DEFAULT*, *spec=None*, *create=False*, *spec_set=None*, *autospec=None*, *new_callable=None*, ***kwargs*)

patch acts as a function decorator, class decorator or a context manager. Inside the body of the function or with statement, the *target* is patched with a *new* object. When the function/with statement exits the patch is undone.

If *new* is omitted, then the target is replaced with a *MagicMock*. If *patch* is used as a decorator and *new* is omitted, the created mock is passed in as an extra argument to the decorated function. If *patch* is used as a context manager the created mock is returned by the context manager.

target should be a string in the form *'package.module.ClassName'*. The *target* is imported and the specified object replaced with the *new* object, so the *target* must be importable from the environment you are calling *patch* from. The target is imported when the decorated function is executed, not at decoration time.

The *spec* and *spec_set* keyword arguments are passed to the *MagicMock* if patch is creating one for you.

In addition you can pass *spec=True* or *spec_set=True*, which causes patch to pass in the object being mocked as the spec/spec_set object.

new_callable allows you to specify a different class, or callable object, that will be called to create the *new* object. By default *MagicMock* is used.

A more powerful form of *spec* is *autospec*. If you set *autospec=True* then the mock with be created with a spec from the object being replaced. All attributes of the mock will also have the spec of the corresponding attribute of the object being replaced. Methods and functions being mocked will have their arguments checked and will raise a *TypeError* if they are called with the wrong signature. For mocks replacing a class, their return value (the 'instance') will have the same spec as the class.

Instead of *autospec=True* you can pass *autospec=some_object* to use an arbitrary object as the spec instead of the one being replaced.

By default *patch* will fail to replace attributes that don't exist. If you pass in *create=True*, and the attribute doesn't exist, patch will create the attribute for you when the patched function is called, and delete it again afterwards. This is useful for writing tests against attributes that your production code creates at runtime. It is off by by default because it can be dangerous. With it switched on you can write passing tests against APIs that don't actually exist!

Patch can be used as a *TestCase* class decorator. It works by decorating each test method in the class. This reduces the boilerplate code when your test methods share a common patchings set. *patch* finds tests by looking for method names that start with *patch.TEST_PREFIX*. By default this is *test*, which matches the way *unittest* finds tests. You can specify an alternative prefix by setting *patch.TEST_PREFIX*.

Patch can be used as a context manager, with the with statement. Here the patching applies to the indented block after the with statement. If you use "as" then the patched object will be bound to the name after the "as"; very useful if *patch* is creating a mock object for you.

patch takes arbitrary keyword arguments. These will be passed to the *Mock* (or *new_callable*) on construction.

patch.dict(...), *patch.multiple(...)* and *patch.object(...)* are available for alternate use-cases.

case.mock.**wrap_logger**(*logger*, *loglevel=40*)

Wrap `logging.Logger` with a StringIO() handler.

yields a StringIO handle.

Example:

```
    with mock.wrap_logger(logger, loglevel=logging.DEBUG) as sio:
        ...
        sio.getvalue()
```

case.mock.**environ**(*env_name*, *env_value*)
Mock environment variable value.

Example:

```
    @mock.environ('DJANGO_SETTINGS_MODULE', 'proj.settings')
    def test_other_settings(self):
        ...
```

case.mock.**sleepdeprived**(*module=<module 'time' (built-in)>*)
Mock time.sleep to do nothing.

Example:

```
    @mock.sleepdeprived()  # < patches time.sleep
    @mock.sleepdeprived(celery.result)  # < patches celery.result.sleep
```

case.mock.**mask_modules**(**modnames*)
Ban some modules from being importable inside the context

For example:

```
    >>> with mask_modules('sys'):
    ...     try:
    ...         import sys
    ...     except ImportError:
    ...         print('sys not found')
    sys not found

    >>> import sys  # noqa
    >>> sys.version
    (2, 5, 2, 'final', 0)
```

Or as a decorator:

```
    @mask_modules('sys')
    def test_foo(self):
        ...
```

case.mock.**stdouts**()
Override *sys.stdout* and *sys.stderr* with *StringIO* instances.

Decorator example:

```
    @mock.stdouts
    def test_foo(self, stdout, stderr):
        something()
        self.assertIn('foo', stdout.getvalue())
```

Context example:

```
    with mock.stdouts() as (stdout, stderr):
        something()
        self.assertIn('foo', stdout.getvalue())
```

case.mock.**replace_module_value**(*module*, *name*, *value=None*)
Mock module value, given a module, attribute name and value.

Decorator example:

```
@mock.replace_module_value(module, 'CONSTANT', 3.03)
def test_foo(self):
    ...
```

Context example:

```
with mock.replace_module_value(module, 'CONSTANT', 3.03):
    ...
```

case.mock.**sys_version**(*value=None*)
Mock sys.version_info

Decorator example:

```
@mock.sys_version((3, 6, 1))
def test_foo(self):
    ...
```

Context example:

```
with mock.sys_version((3, 6, 1)):
    ...
```

case.mock.**pypy_version**(*value=None*)
Mock sys.pypy_version_info

Decorator example:

```
@mock.pypy_version((3, 6, 1))
def test_foo(self):
    ...
```

Context example:

```
with mock.pypy_version((3, 6, 1)):
    ...
```

case.mock.**platform_pyimp**(*value=None*)
Mock platform.python_implementation

Decorator example:

```
@mock.platform_pyimp('PyPy')
def test_foo(self):
    ...
```

Context example:

```
with mock.platform_pyimp('PyPy'):
    ...
```

case.mock.**sys_platform**(*value=None*)
Mock sys.platform

Decorator example:

```
@mock.sys_platform('darwin')
def test_foo(self):
    ...
```

Context example:

```
with mock.sys_platform('darwin'):
    ...
```

case.mock.**reset_modules**(**modules*)

Remove modules from `sys.modules` by name, and reset back again when the test/context returns.

Decorator example:

```
@mock.reset_modules('celery.result', 'celery.app.base')
def test_foo(self):
    pass
```

Context example:

```
with mock.reset_modules('celery.result', 'celery.app.base'):
    pass
```

case.mock.**module**(**names*)

Mock one or modules such that every attribute is a *Mock*.

case.mock.**open**(*typ=<class 'case.utils.WhateverIO'>*, *side_effect=None*)

Patch builtins.open so that it returns StringIO object.

Parameters

- **typ** – File object for open to return. Defaults to `WhateverIO` which is the bastard child of `io.StringIO` and `io.BytesIO` accepting both bytes and unicode input.
- **side_effect** – Additional side effect for when the open context is entered.

Decorator example:

```
@mock.open()
def test_foo(self, open_fh):
    something_opening_and_writing_a_file()
    self.assertIn('foo', open_fh.getvalue())
```

Context example:

```
with mock.open(io.BytesIO) as open_fh:
    something_opening_and_writing_bytes_to_a_file()
    self.assertIn(b'foo', open_fh.getvalue())
```

case.mock.**restore_logging**()

Restore root logger handlers after test returns.

Decorator example:

```
@mock.restore_logging()
def test_foo(self):
    setup_logging()
```

Context example:

```
with mock.restore_logging():
    setup_logging()
```

case.mock.**module_exists**(**modules*)

Patch one or more modules to ensure they exist.

A module name with multiple paths (e.g. gevent.monkey) will ensure all parent modules are also patched (`gevent` + `gevent.monkey`).

Decorator example:

```
@mock.module_exists('gevent.monkey')
def test_foo(self):
    pass
```

Context example:

```
with mock.module_exists('gevent.monkey'):
    gevent.monkey.patch_all = Mock(name='patch_all')
    ...
```

`case.mock.`**`create_patcher`**(**partial_path*)

2.1.5 case.utils

class `case.utils.`**`WhateverIO`**(*v=None*, **a*, ***kw*)

`write`(*data*)

`case.utils.`**`decorator`**(*predicate*)

`case.utils.`**`get_logger_handlers`**(*logger*)

`case.utils.`**`noop`**(**args*, ***kwargs*)

`case.utils.`**`symbol_by_name`**(*name*, *aliases={}*, *imp=None*, *package=None*, *sep=u'.'*, *default=None*, ***kwargs*)

Get symbol by qualified name.

The name should be the full dot-separated path to the class:

```
modulename.ClassName
```

Example:

```
celery.concurrency.processes.TaskPool
                            ^- class name
```

or using ':' to separate module and symbol:

```
celery.concurrency.processes:TaskPool
```

If *aliases* is provided, a dict containing short name/long name mappings, the name is looked up in the aliases first.

Examples:

```
>>> symbol_by_name('celery.concurrency.processes.TaskPool')
<class 'celery.concurrency.processes.TaskPool'>
```

```
>>> symbol_by_name('default', {
...     'default': 'celery.concurrency.processes.TaskPool'})
<class 'celery.concurrency.processes.TaskPool'>
```

Does not try to look up non-string names. >>> from celery.concurrency.processes import TaskPool >>> symbol_by_name(TaskPool) is TaskPool True

2.2 Changes

2.3 1.0.3

release-date 2016-04-08 10:00 P.M PDT

- Adds new Mock methods from Python 3.6:
  - `Mock.assert_called()`
  - `Mock.assert_not_called()`
  - `Mock.assert_called_once()`
- Adds `Mock.create_patcher()`

  Example:

```
from case import Case, mock

patch_commands = mock.create_patcher('long_name.management.commands')

class test_FooCommand(Case):

    @patch_commands('FooCommand.authenticate')
    def test_foo(self, authenticate):
        pass
```

2.4 1.0.3

release-date 2016-04-06 04:00 P.M PDT

- Python 2.6 compatibility.
- `mock.platform_pyimp` no longer accepted default value.

2.5 1.0.2

release-date 2016-04-06 03:46 P.M PDT

- Adds docstrings

2.6 1.0.1

release-date 2016-04-05 04:00 P.M PDT

- Fixed issues with Python 3

2.7 1.0.0

release-date 2016-04-05 02:00 P.M PDT

release-by Ask Solem

- Initial release

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

C

T

U

W