TestSlide Documentation

Release 1.3.0

Fabio Pugliese Ornellas

Jan 25, 2019

Contents:

TestSlide makes writing tests fluid and easy. Whether you prefer classic unit testing, TDD or BDD, it helps you be productive, with its easy to use well behaved mocks and its awesome test runner.

It is designed to work well with other test frameworks, so you can use it on top of existing `unittest.TestCase` without rewriting everything.

CHAPTER 1

Quickstart

Install the package:

```
pip install TestSlide
```

Scaffold the code you want to test `backup.py`:

```
class Backup(object):
  def delete(self, path):
    pass
```

Write a test case `backup_test.py` describing the expected behavior:

```
import testslide, backup, storage

class TestBackupDelete(testslide.TestCase):
  def setUp(self):
    super().setUp()
    self.storage_mock = testslide.StrictMock(storage.Client)
    # Makes storage.Client(timeout=60) return self.storage_mock
    self.mock_constructor(storage, 'Client')\
      .for_call(timeout=60)\
      .to_return_value(self.storage_mock)

  def test_delete_from_storage(self):
    # Set behavior and assertion for the call at the mock
    self.mock_callable(self.storage_mock, 'delete')\
      .for_call('/file/to/delete')\
      .to_return_value(True)\
      .and_assert_called_once()
    backup.Backup().delete('/file/to/delete')
```

TestSlide's *StrictMock* , *mock_callable()* and *mock_constructor()* are seamlessly integrated with Python's TestCase.

Run the test and see the failure:

```
$  testslide backup_test.py
backup_test.TestBackupDelete
  test_delete_from_storage: AssertionError: calls did not match assertion.

Failures:

  1) backup_test.TestBackupDelete: test_delete_from_storage
    1) AssertionError: calls did not match assertion.
    <StrictMock 0x7F6E88070F98 template=storage.Client>, 'delete':
      expected: called exactly 1 time(s) with arguments:
        ('/file/to/delete',)
        {}
      received: 0 call(s)
      File "/opt/python/lib/python3.6/unittest/case.py", line 59, in testPartExecutor
        yield
      File "/opt/python/lib/python3.6/unittest/case.py", line 646, in doCleanups
        function(*args, **kwargs)

Finished 1 example(s) in 0.0s:
  Failed: 1
```

TestSlide's mocks failure messages guide you towards the solution, that you can now implement:

```
import storage

class Backup(object):
  def __init__(self):
    self.storage = storage.Client(timeout=60)

  def delete(self, path):
    self.storage.delete(path)
```

And watch the test go green:

```
$  testslide backup_test.py
backup_test.TestBackupDelete
  test_delete_from_storage

Finished 1 example(s) in 0.0s:
  Successful: 1
```

It is all about letting the failure messages guide you towards the solution. There's a plethora of validation inside TestSlide's mocks, so you can trust they will help you iterate quickly when writing code and also cover you when breaking changes are introduced.

1.1 Test Runner

TestSlide has its own *DSL* that you can use to write tests, and so it comes with its own test runner. However, it can also execute tests written for Python's unittest, so you can have its benefits, without having to rewrite everything.

To use, simply give it a list of `.py` files containing the tests:

```
$ testslide calculator_test.py
calculator_test.TestCalculatorAdd
  test_add_negative: PASS
  test_add_positive: PASS
calculator_test.TestCalculatorSub
  test_sub_negative: PASS
  test_sub_positive: PASS

Finished 4 example(s) in 0.0s:
  Successful:  4
```

Note: For documentation simplicity, the output shown here is monochromatic and boring. When executing TestSlide from a terminal, it is **colored**, making it significantly easier to read. Eg: green for success, red for failure.

Whatever `unittest.TestCase` or *DSL* declared in the given files will be executed. You can even mix them in the same project or file.

Note: When using *mock_callable()* or *mock_constructor()* you must inherit your test class from `testslide.TestCase` to have access to those methods. The test runner does **not** require that, and is happy to run tests that inherit directly (or indirectly) from `unittest.TestCase`.

Note: Tests inheriting from `testslide.TestCase` can **also** be executed by Python's unittest CLI.

1.1.1 Listing Available Tests

You can use `--list` to run test discovery and list all tests found:

```
$ testslide backup_test.py
backup_test.TestBackupDelete: test_delete_from_storage
```

1.1.2 Multiple Failures Report

When using TestSlide's *mock_callable()* assertions, you can have a better signal on failures. For example, in this test we have two assertions:

```
def test_delete_from_storage(self):
  self.mock_callable(self.storage, 'delete')\
    .for_call('/file').to_return_value(True)\
    .and_assert_called_once()
  self.assertEqual(Backup().delete('/file'), True)
```

Normally when a test fails, you get only signal from the first failure. TestSlide's Test Runner can understand what you meant, and give you a more comprehensive signal, telling about each failed assertion:

```
$ testslide backup_test.py
backup_test.TestBackupDelete
  test_delete_from_storage: AssertionError: <StrictMock 0x7F55C5159B38␣
↪template=storage.Client>,   'delete':

Failures:

1) backup_test.TestBackupDelete: test_delete_from_storage
  1) AssertionError: None != True
    File "backup_test.py", line 47, in test_delete
      self.assertEqual(Backup().delete('/file'), True)
    File "/opt/python3.6/unittest/case.py", line 829, in assertEqual
      assertion_func(first, second, msg=msg)
    File "/opt/python3.6/unittest/case.py", line 822, in _baseAssertEqual
      raise self.failureException(msg)
  2) AssertionError: <StrictMock 0x7F55C5159B38 template=storage.Client>, 'delete':
    expected: called exactly 1 time(s) with arguments:
      ('/file',)
      {}
    received: 0 call(s)
    File "/opt/python3.6/unittest/case.py", line 59, in testPartExecutor
      yield
    File "/opt/python3.6/unittest/case.py", line 646, in doCleanups
      function(*args, **kwargs)
```

1.1.3 Failing Fast

When you change something and too many tests break, it is useful to stop the execution at the first failure, so you can iterate easier. To do that, use the `--fail-fast` option.

1.1.4 Focus and Skip

TestSlide allows you to easily focus execution of a single test, by simply adding `f` to the name of the test function:

```
def ftest_sub_positive(self):
  self.assertEqual(
    Calc().sub(1, 1), 0
  )
```

And then run your tests with `--focus`:

```
$ testslide calc_test.py
calc.TestCalcSub
  *ftest_sub_positive: PASS

Finished 1 example(s) in 0.0s:
  Successful: 1
  Not executed: 3
```

Only `ftest` tests will be executed. Note that it also tells you how many tests were not executed.

Similarly, you can skip a test with `x`:

```
def xtest_sub_positive(self):
  self.assertEqual(
    Calc().sub(1, 1), 0
  )
```

And this test will be skipped:

```
$ testslide calc_test.py
calc.TestCalcAdd
  test_add_negative: PASS
  test_add_positive: PASS
calc.TestCalcSub
  test_sub_negative: PASS
  xtest_sub_positive: SKIP

Finished 4 example(s) in 0.0s:
  Successful: 3
  Skipped: 1
```

1.1.5 Stack Trace Simplification

Stack traces can be hard to read. By default, TestSlide trims the working directory from file names on stack traces, simplifying the output. You can tweak this behavior with `--trim-strace-path-prefix`.

Also, stack trace lines that are from TestSlide's code base are hidden, as they are only useful when debugging TestSlide itself.

1.1.6 Shuffled Execution

Each test must be independent and isolated from each other. For example, if one test manipulates some module level object, that the next test depends on, we are leaking the context of one test to the next. To catch such cases, you can run your tests with `--shuffle`: tests will be executed in a random order every time. The test signal must always be the same, no matter in what order tests run. You can tweak the seed with `--seed`.

1.1.7 Slow Imports Profiler

As projects grow with more dependencies, running a test for a few lines of code can take several seconds. This is often cause by time spent on importing dependencies, rather that the tests themselves. If you run your tests with `--import-profiler $MS`, any imported module that took more that that the given amount of milliseconds will be reported in a nice and readable tree view. This helps you optimize your imports, so your unit tests can run faster. Frequently, the cause of slow imports is the construction of heavy objects at module level.

1.1.8 Tip: Automatic Test Execution

To help iterate even quicker, you can pair `testslide` execution with entr (or any similar):

```
find . -name \*.py | entr testslide tests/.py
```

This will automatically execute all your tests, whenever a file is saved. This is particularly useful when paired with focus and skip. This means **you don't have to leave your text editor, to iterate over your tests and code**.

1.2 StrictMock

When unit testing, mocks are often used in place of a real dependency, so tests can run independently. Mocks must behave exactly like the real thing, by returning configured canned responses, but rejecting anything else. If this is not true, it is hard to trust your test results.

Let's see a practical example of that:

```
In [1]: from unittest.mock import Mock

In [2]: class Calculator:
   ...:   def is_odd(self, x):
   ...:     return bool(x % 2)
   ...:

In [3]: mock = Mock(Calculator)

In [4]: mock.is_odd(2)
Out[4]: <Mock name='mock.is_odd()' id='140674180253512'>

In [5]: bool(mock.is_odd(2))
Out[5]: True

In [6]: mock.is_odd(2, 'invalid')
Out[6]: <Mock name='mock.is_odd()' id='140674180253512'>
```

The mock, right after being created, already has dangerous behavior. When `is_odd()` is called, another mock is returned. And it is unconditionally `True`. And this is wrong: 2 is **not** odd. When this happens in your test, it is hard to trust its results: it might go green, even with buggy code. Also note how the mock accepts calls with any arguments, even if they don't match the original method signature.

1.2.1 A Well Behaved Mock

StrictMock is **safe by default**: it only has configure behavior:

```
In [1]: from testslide import StrictMock

In [2]: class Calculator:
   ...:   def is_odd(self, x):
   ...:     return bool(x % 2)
   ...:

In [3]: mock = StrictMock(Calculator)

In [4]: mock.is_odd(2)
(...)
UndefinedBehavior: <StrictMock 0x7F290A3DD860 template=__main__.Calculator>:
  Attribute 'is_odd' has no behavior defined.
  You can define behavior by assigning a value to it.
```

Instead of guessing what `is_odd` should return, StrictMock clearly tells you it was not told what to do with it. In this case, the mock is clearly missing the behavior, that we can trivially add:

```
In [5]: mock.is_odd = lambda number: False
```

(continues on next page)

(continued from previous page)

```
In [6]: mock.is_odd(2)
Out[6]: False
```

1.2.2 API Validations

StrictMock does a lot of validation under the hood, so you can trust its behavior, even when breaking changes are introduced.

Attribute Existence

You won't be allowed to set an attribute to a StrictMock if the given template class does not have it:

```
In [1]: from testslide import StrictMock

In [2]: class Calculator:
   ...:   def is_odd(self, x):
   ...:     return bool(x % 2)
   ...:

In [3]: mock = StrictMock(Calculator)

In [4]: mock.invalid = 'whatever'
(...)
NoSuchAttribute: <StrictMock 0x7F7821920780 template=__main__.Calculator>:
  No such attribute 'invalid'.
  Can not set attribute invalid that is neither part of template class Calculator or␣
↪runtime_attrs=[].
```

Dynamic Attributes

StrictMock will introspect at the template class code, to detect attributes that are dynamically defined:

```
In [1]: from testslide import StrictMock
   ...:

In [2]: class DynamicAttr(object):
   ...:     def __init__(self):
   ...:         self.dynamic = 'set from __init__'
   ...:

In [3]: mock = StrictMock(DynamicAttr)

In [4]: mock.dynamic = 'something else'
```

Note: This feature is **not** available in Python 2!

The detection mechanism can only detect attributes defined from `__init__`. If you have attributes defined at other places, you will need to inform them explicitly:

```
StrictMock(TemplateClass, runtime_attrs=['attr_name'])
```

Method Signatures

StrictMock also ensures that method signatures match the ones from the template class:

```
In [1]: from testslide import StrictMock

In [2]: class Calculator:
   ...:   def is_odd(self, x):
   ...:     return bool(x % 2)
   ...:

In [3]: mock = StrictMock(Calculator)

In [4]: mock.is_odd = lambda number, invalid: False

In [5]: mock.is_odd(2, 'invalid')
(...)
TypeError: too many positional arguments
```

1.2.3 Magic Methods

Defining behavior for magic methods works out of the box:

```
In [1]: from testslide import StrictMock

In [2]: mock = StrictMock()

In [3]: mock.__str__ = lambda: 'mocked str'

In [4]: str(mock)
Out[4]: 'mocked str'
```

1.2.4 Naming

You can optionally name your mock, to make it easier to identify:

```
In [1]: from testslide import StrictMock

In [2]: str(StrictMock())
Out[2]: '<StrictMock 0x7F7A30FC0748>'

In [3]: str(StrictMock(name='whatever'))
Out[3]: "<StrictMock 0x7F7A30FDFF60 name='whatever'>"
```

1.2.5 Generic Mocks

It is recommended to use StrictMock giving it a template class, so you can leverage its API validation. There are situations however, that any "generic mock" is good enough. You can still use StrictMock, although you'll loose most validations:

```
In [1]: from testslide import StrictMock

In [2]: mock = StrictMock()

In [3]: mock.whatever
(...)
UndefinedBehavior: <StrictMock 0x7FED1C724C18>:
  Attribute 'whatever' has no behavior defined.
  You can define behavior by assigning a value to it.

In [4]: mock.whatever = 'something'

In [5]: mock.whatever
Out[5]: 'something'
```

It will accept setting any attributes, with any values.

1.2.6 Extra Functionality

- `copy.copy()` and `copy.deepcopy()` works, and give back another StrictMock, with the same behavior.
- Template classes that use `__slots__` are supported.
- If the template class is a context manager, the StrictMock instance will also define `__enter__`, yielding itself, and an empty `__exit__`.

1.3 mock_callable()

While *StrictMock* specializes in creating mocks that behave like some real object, mock_callable() focuses on mocking functions and instance/static/class methods. In a single shot, it allows you to:

- Create a callable mock.
- Define what call to accept.
- Define call behavior.
- Patch the callable mock somewhere.
- Define a call assertion (optional).

Sounds complicated, but it is not:

```
import os
from testslide import TestCase

def rm(path):
  os.remove(path)

class TestRm(TestCase):
  def test_remove_from_filesystem(self):
    path = '/some/file'
    self.mock_callable(os, 'remove')\
      .for_call(path)\
      .to_return_value(None)\
      .and_assert_called_once()
    rm(path)
```

This test will **only** pass if `os.remove` was called once with `path`. It will fail if `os.remove`:

- Is not called.
- Is called more than once.
- Is called with any other argument.

For example, if the code is broken and does `os.remove('/wrong/file')`:

```
$ testslide rm_test.py
rm_test.TestRm
  test_remove_from_filesystem: AggregatedExceptions: 2 failures.

Failures:

  1) rm_test.TestRm: test_remove_from_filesystem
    1) UnexpectedCallArguments: <module 'os' from '/opt/python/lib/python3.6/os.py'>,
↪'remove':
      Received call:
        ('/wrong/file',)
        {}
      But no behavior was defined for it.
      These are the registered calls:
        ('/some/file',)
        {}

      File "rm_test.py", line 14, in test_remove_from_filesystem
        rm(path)
      File "rm_test.py", line 5, in rm
        os.remove('/wrong/file')
      File "/opt/python/lib/python3.6/unittest/mock.py", line 939, in __call__
        return _mock_self._mock_call(*args, **kwargs)
      File "/opt/python/lib/python3.6/unittest/mock.py", line 1005, in _mock_call
        ret_val = effect(*args, **kwargs)
    2) AssertionError: calls did not match assertion.
    <module 'os' from '/opt/python/lib/python3.6/os.py'>, 'remove':
      expected: called at least 1 time(s) with arguments:
        ('/some/file',)
        {}
      received: 0 call(s)
      File "/opt/python/lib/python3.6/unittest/case.py", line 59, in testPartExecutor
        yield
      File "/opt/python/lib/python3.6/unittest/case.py", line 646, in doCleanups
        function(*args, **kwargs)

Finished 1 example(s) in 0.0s:
  Failed: 1
```

Note how you get two failed assertions, instead of just one:

- The mock was called with something unexpected.
- The expected call did not happen.

It is now pretty clear what is broken, and why it is broken.

1.3.1 Defining a Target

You always start mock_callable with:

```
self.mock_callable(target, 'attribute_name')
```

`target` can be:

- A *StrictMock*.
- A module.
  - The module can be given as a reference (eg: `time`) or as a string (eg: `"time"`). The latter allows you to avoid importing the module at the same file you use mock_callable.
- A Class
- Any object.

`attribute_name` is the name of the function / method you want to mock.

Note: You can mock instance methods at instances of classes but not at the class. This is by design, as mocking instance methods at the class affects every instance of that class, not just what's needed for the test, making it easy to introduce bugs. Assertions can be ambiguous: `.and_assert_called_twice()` means one instance called twice, or two instances called once each?

1.3.2 Defining Accepted Calls

By default, mock_callable accepts all call arguments:

```
self.mock_callable(os, 'remove')\
  .to_return_value(None)
for n in range(3):
  os.remove(str(n)) # => None
```

You can define precisely what arguments to accept:

```
self.mock_callable(os, 'remove')\
  .for_call('/some/file')\
  .to_return_value(None)
os.remove('/some/file') # => None
os.remove('/some/other/file') # => raises UnexpectedCallArguments
```

Note how it is **safe by default**: once `for_call` is used, other calls will not be accepted.

Composition

You can use mock_callable for the same target as many times as needed, so you can compose the behavior you need:

```
self.mock_callable(os, 'remove')\
  .to_raise(FileNotFoundError)
self.mock_callable(os, 'remove')\
  .for_call('/some/file')\
  .to_return_value(None)
self.mock_callable(os, 'remove')\
  .for_call('/some/other/file')\
  .to_return_value(None)
os.remove('/some/file') # => None
```

(continues on next page)

(continued from previous page)

```
os.remove('/some/other/file') # => None
os.remove('/anything/else') # => raises FileNotFoundError
```

mock_callable scans the list of registered calls **from last to first**, until it finds a match (`UnexpectedCallArguments` is raised if there's no match). In this example, `FileNotFoundError` essentially became the default behavior. This is particularly powerful when you configure it at the `setUp()` phase of your tests, then specialize the behavior inside each test function, for specific arguments.

1.3.3 Defining Call Behavior

The **safe by default** rational spans to call behavior. There's no default, and you are required to define what happens when the call is made.

Returning a value

Always return the same value:

```
self.mock_callable(os, 'remove')\
  .for_call('/some/file')\
  .to_return_value(None)
```

Returning a series of values

Return each value from a list until exhausted:

```
self.mock_callable(time, 'time')\
  .to_return_values([1.0, 2.0, 3.0])
time.time() => 1.0
time.time() => 2.0
time.time() => 3.0
time.time() => raises UndefinedBehaviorForCall
```

Yielding values

You can return a generator with:

```
self.mock_callable(some_object, 'some_method_name')\
  .to_yield_values([1, 2, 3])
for each_value in some_object.some_method_name():
  print(each_value)  # => 1, 2, 3
```

Raising exceptions

You can raise exceptions by either giving an exception class itself or an instance of it:

```
self.mock_callable(some_object, 'some_method_name')\
  .to_raise(RuntimeError)
some_object.some_method_name()  # => raise RuntimeError
```

Replacing the original implementation

Replace the original implementation with something else:

```
def func():
  return 33

self.mock_callable(some_object, 'some_method_name')\
  .with_implementation(func)
some_object.some_method_name()  # => 33
```

Note: `func` can be any callable (eg: a lambda).

Wrapping the original implementation

When the target is a real object (not a mock), it can be useful to still call the original method, process its return perhaps, and return something else:

```
def trim_query(original_callable):
  return original_callable()[0:5]

self.mock_callable(some_service, 'big_query')\
  .with_wrapper(trim_query)
some_service.big_query()  # => returns trimmed list
```

Calling the original implementation

Sometimes it is useful to mock only cherry picked calls for real targets and allow all other calls through:

```
self.mock_callable(some_object, 'some_method')\
  .to_call_original()
self.mock_callable(some_object, 'some_method')\
  .for_args('specific call')\
  .to_return_value('specific response')
some_object.some_method('any call')  # => returns whatever some_object.some_method()␣
↪returns
some_object.some_method('specific call')  # => 'specific response'
```

You can achieve the opposite (specific call goes through, mocked general case) with:

```
self.mock_callable(some_object, 'some_method_name')\
  .to_return_value('general case')
self.mock_callable(some_object, 'some_method_name')\
  .for_args('specific case')\
  .to_call_original()
some_object.some_method_name('whatever')  # => 'general case'
some_object.some_method_name('specific case')  # => Calls the original callable, and␣
↪return the value
```

1.3.4 Defining Call Assertions

When dealing with external dependencies, it is useful to assert on calls to them **when they have side-effects**. mock_callable allows you to do this:

```
self.mock_callable(os, 'remove')\
  .for_call(path)\
  .to_return_value(None)\
  .and_assert_called_once()
```

Here's a list of all assertions you can define:

```
.and_assert_called_exactly(times)
.and_assert_called_once()
.and_assert_called_twice()
.and_assert_called_at_least(times)
.and_assert_called_at_most(times)
.and_assert_called()
.and_assert_not_called()
```

Within the same test, you can define as many assertions as needed.

1.3.5 Cheat Sheet

It is a good idea to keep this at hand when using mock_callable:

```
self.mock_callable(target, 'callable_name')\
  # Call to accept
  .for_call(*args, **kwargs)\
  # Behavior
  .to_return_value(value)\
  .to_return_values(values_list)\
  .to_yield_values(values_list)\
  .to_raise(exception)\
  .with_implementation(func)\
  .with_wrapper(func)\
  .to_call_original()\
  # Assertion (optional)
  .and_assert_called_exactly(times)
  .and_assert_called_once()
  .and_assert_called_twice()
  .and_assert_called_at_least(times)
  .and_assert_called_at_most(times)
  .and_assert_called()
  .and_assert_not_called()
```

1.3.6 Magic Methods

Mocking magic methods (eg: `__str__`) for an instance can be quite tricky, as `str(obj)` requires the mock to be made at `type(obj)`. mock_callable implements the complicated mechanics required to make it work, so you can easily mock directly at instances:

```
import time
from testslide import TestCase
```

(continues on next page)

(continued from previous page)

```
class A:
  def __str__(self):
    return 'original'

class TestMagicMethodMocking(TestCase):
  def test_str(self):
    a = A()
    other_a = A()
    self.assertEqual(str(a), 'original')
    self.mock_callable(a, '__str__')\
      .to_return_value('mocked')
    self.assertEqual(str(a), 'mocked')
    self.assertEqual(str(other_a), 'original')
```

The mock works for the target instance, but does not affect other instances.

1.3.7 Signature Validation

mock_callable implements signature validation. When you use it, the mock will raise `TypeError` if it is called with a signature that does not match the original method:

```
import time
from testslide import TestCase

class A:
  def one_arg(self, arg):
    return 'original'

class TestSignature(TestCase):
  def test_signature(self):
    a = A()
    self.mock_callable(a, 'one_arg')\
      .to_return_value('mocked')
    self.assertEqual(a.one_arg('one'), 'mocked')
    with self.assertRaises(TypeError):
      a.one_arg('one', 'invalid')
```

This is particularly helpful when changes are introduced to the code: if a mocked method changes the signature, even when mocked, mock_callable will give you the signal that there's something broken.

1.3.8 Integration With Other Frameworks

mock_callable comes out of the box with support for Python's unittest (via `testslide.TestCase`) and *TestSlide's DSL*. You can easily integrate it with any other test framework you prefer:

- mock_callable calls `testslide.mock_callable.register_assertion` passing a callable object whenever an assertion is defined. You must set it to a function that will execute the assertion **after** the test code finishes. Eg: for Python's unittest: `testslide.mock_callable.register_assertion = lambda assertion: self.addCleanup(assertion)`.
- After each test execution, you must **unconditionally** call `testslide.mock_callable.unpatch_all_callable_mocks`. This will undo all patches, so the next test is not affected by them. Eg: for Python's unittest: `self.addCleanup(testslide.mock_callable.unpatch_all_callable_mocks)`.

- You can then call `testslide.mock_callable.mock_callable` directly from your tests.

1.4 mock_constructor()

Let's say we wan to unit test the `Backup.delete` method:

```
import storage

class Backup(object):
  def __init__(self):
    self.storage = storage.Client(timeout=60)

  def delete(self, path):
    self.storage.delete(path)
```

We want to ensure that when `Backup.delete` is called, it actually deletes `path` from the storage as well, by calling `storage.Client.delete`. We can leverage *StrictMock* and *mock_callable()* for that:

```
self.storage_mock = StrictMock(storage.Client)
self.mock_callable(self.storage_mock, 'delete')\
  .for_call('/file/to/delete')\
  .to_return_value(True)\
  .and_assert_called_once()
Backup().delete('/file/to/delete')
```

The question now is: how to put `self.storage_mock` inside `Backup.__init__`? This is where **mock_constructor** jumps in:

```
from testslide import TestCase, StrictMock, mock_callable
import storage
from backup import Backup

class TestBackupDelete(TestCase):
  def setUp(self):
    super().setUp()
    self.storage_mock = StrictMock(storage.Client)
    self.mock_constructor(storage, 'Client')\
      .for_call(timeout=60)\
      .to_return_value(self.storage_mock)

  def test_delete_from_storage(self):
    self.mock_callable(self.storage_mock, 'delete')\
      .for_call('/file/to/delete')\
      .to_return_value(True)\
      .and_assert_called_once()
    Backup().delete('/file/to/delete')
```

mock_constructor makes `storage.Client(timeout=60)` return `self.storage_mock`. It is similar to *mock_callable()*, accepting the same call, behavior and assertion definitions. Similarly, it will also fail if `storage.Client()` (missing timeout) is called.

Note how by using mock_constructor, not only you get all **safe by default** goodies, but also **totally decouples** your test from the code. This means that, no matter how `Backup` is refactored, the test remains the same.

1.4.1 Implementation Details

In principle, doing:

```
self.mock_callable(SomeClass, '__new__')\
  .for_call()\
  .to_return_value(some_class_mock)
```

Should be all you need. However, as of October 2018, Python 3 has a bug https://bugs.python.org/issue25731 that prevents this from working (it works in Python 2).

mock_callable is a way to not only solve this for Python 3, but also provide the same interface for both.

Internally, mock_callable will:

- Patch the class at its module with a subclass of it, that is dynamically created.
- `__new__` of this dynamic subclass is handled by mock_callable.

1.4.2 Integration With Other Frameworks

mock_constructor comes out of the box with support for Python's unittest (via `testslide.TestCase`) and *TestSlide's DSL*. You can easily integrate it with any other test framework you prefer:

- Integrate *mock_callable()* (used by mock_constructor under the hook).
- After each test execution, you must **unconditionally** call `testslide.mock_constructor.unpatch_all_callable_mocks`. This will undo all patches, so the next test is not affected by them. Eg: for Python's unittest: `self.addCleanup(testslide.mock_constructor.unpatch_all_callable_mocks)`.
- You can then call `testslide.mock_constructor.mock_constructor` directly from your tests.

1.5 TestSlide's DSL

When testing complex scenarios with lots of variations, or when doing BDD, TestSlide's DSL helps you break down your test cases close to spoken language. Composition of test scenarios enables covering more ground with less effort. Think of it as `unittest.TestCase` on steroids.

Let's say we want to test this class:

```
import storage

class Backup:
  def __init__(self):
    self.storage = storage.Client(timeout=60)

  def delete(self, path):
    self.storage.delete(path)
```

We can test use it with:

```
from testslide.dsl import context
from testslide import StrictMock
import storage
import backup
```

(continues on next page)

(continued from previous page)

```
@context
def Backup(context):

  context.memoize("backup", lambda self: backup.Backup())

  context.memoize("storage_mock", lambda self: StrictMock(storage.Client))

  @context.before
  def mock_storage_Client(self):
    self.mock_constructor(storage, 'Client')\
      .for_call(timeout=60)\
      .to_return_value(self.storage_mock)

  @context.sub_context
  def delete(context):
    context.memoize("path", lambda self: '/some/file')

    @context.after
    def call_backup_delete(self):
      self.backup.delete(self.path)

    @context.example
    def it_deletes_from_storage_backend(self):
      self.mock_callable(self.storage_mock, 'delete')\
        .for_call(self.path)\
        .to_return_value(True)\
        .and_assert_called_once()
```

And when we run it:

```
$ testslide backup_test.py
Backup
  delete
    it deletes from storage backend

Finished 1 example(s) in 0.0s:
  Successful: 1
```

As you can see, we can declare contexts for testing, and keep building on top of them:

- The top `Backup` context contains the object we want to test, and the common mocks needed.
- The nested `delete` context always calls `Backup.delete` after each example.
- The `it_deletes_from_storage_backend` example defines only the assertion needed for it.

As the `Backup` class grows, it is easy to nest new contexts, and reuse what's already defined.

1.5.1 Contexts and Examples

Within TestSlide's DSL language, a single test is called an **example**. All examples are declared inside a **context**. Contexts can be arbitrarily nested.

Contexts hold code that sets up and tear down the environment for each particular scenario. Things like instantiating objects and setting up mocks are usually part of the context.

Examples hold only code required to test the particular case.

Let's see it in action:

```
from testslide.dsl import context

@context
def calculator(context):

    @context.sub_context
    def addition(context):

        @context.example
        def sums_given_numbers(self):
            pass

    @context.sub_context
    def subtract(context):

        @context.example
        def subtracts_given_numbers(self):
            pass
```

This describes the basic behavior of a calculator class. Here's what you get when you run it:

```
calculator
  addition
    sums given numbers: PASS
  subtraction
    subtracts given numbers: PASS

Finished 2 examples in 0.0s:
  Successful: 2
```

Note how TestSlide parses the Python code, and yields a close to spoken language wersion of it.

Sub Examples

Sometimes, within the same example, you want to exercise your code multiple times for the same data. Sub examples allow you to do just that:

```
from testslide.dsl import context

@context
def Sub_examples(context):

  @context.example
  def shows_individual_failures(self):
    for i in range(5):
      with self.sub_example():
        if i %2:
          raise AssertionError('{} failed'.format(i))
    raise RuntimeError('Last Failure')
```

When executed, TestSlide understands all cases, and report them properly:

```
Sub examples
  shows individual failures: AggregatedExceptions: 3 failures.

```

(continues on next page)

(continued from previous page)

```
Failures:

  1) Sub examples: shows individual failures
    1) RuntimeError: Last Failure
      File "sub_examples_test.py", line 12, in shows_individual_failures
        raise RuntimeError('Last Failure')
    2) AssertionError: 1 failed
      File "sub_examples_test.py", line 11, in shows_individual_failures
        raise AssertionError('{} failed'.format(i))
    3) AssertionError: 3 failed
      File "sub_examples_test.py", line 11, in shows_individual_failures
        raise AssertionError('{} failed'.format(i))

Finished 1 example(s) in 0.0s:
  Failed: 1
```

Explicit names

TestSlide extracts the name for contexts and examples from the function name, just swapping _ for a space. If you need special characters at your context or example names, you can do it like this:

```
from testslide.dsl import context

@context('Top-level context name')
def top(context):
  @context.sub_context('sub-context name')
  def sub(context):
    @context.example('example with weird-looking name')
    def ex(self):
      pass
```

Note: When explicitly naming, the function name is irrelevant, just make sure there's no name collision.

1.5.2 Sharing Contexts

You can use shared contexts to avoid code duplication, and share common logic applicable to multiple contexts:

```
from testslide.dsl import context

@context
def Sharing_contexts(context):

    # This context will not be evaluated immediately, and can be reused later
    @context.shared_context
    def Shared_context(context):

        @context.example
        def shared_example(self):
            pass

    @context.sub_context
```

(continues on next page)

(continued from previous page)

```
    def Merging_shared_contexts(context):
        # The shared context will me merged into current context
        context.merge_context('Shared context')

    @context.sub_context
    def Nesting_shared_contexts(context):
        # The shared context will be nested below the current context
        context.nest_context('Shared context')
```

And when we execute them:

```
Sharing contexts
  Merging shared contexts
    shared example: PASS
  Nesting shared contexts
    Shared context
      shared example: PASS

Finished 2 examples in 0.0s:
  Successful: 2
```

Note the difference between merging and nesting a shared context: when you merge, no new sub context is created, when you nest, a new sub context will be created below where it was nested.

Parameterized shared contexts

Your shared contexts can accept optional arguments, that can be used to control its declarations:

```
from testslide.dsl import context

@context
def Sharing_contexts(context):

    # This context will not be evaluated immediately, and can be reused later
    @context.shared_context
    def Shared_context(context, extra_example=False):

        @context.example
        def shared_example(self):
            pass

        if extra_example:

            @context.example
            def extra_shared_example(self):
                pass

    @context.sub_context
    def With_extra_example(context):
        context.merge_context('Shared context', extra_example=True)

    @context.sub_context
    def Without_extra_example(context):
        context.nest_context('Shared context')
```

Note: It is an anti-pattern to reference shared context arguments inside hooks or examples, as there's chance of leaking context from one example to the next.

1.5.3 Context Hooks

Contexts must prepare the test scenario according to its description. To do that, you can configure hooks to run before, after or around individual examples.

Before

Before hooks are executed in the order defined, before each example:

```
from testslide.dsl import context

@context
def before_hooks(context):

  @context.before
  def define_list(self):
    self.value = []

  @context.before
  def append_one(self):
    self.value.append(1)

  def append_two(self):
    self.value.append(2)

  @context.example
  def before_hooks_are_executed_in_order(self):
    self.assertEqual(self.value, [1, 2])
```

Note: The name of the before functions does not matter. It is however useful to give them meaningful names, so they are easier to debug.

If code at a before hook fails (raises), test execution stops with a failure.

Typically, before hooks are used to:

- Setup the object being tested.
- Setup any dependencies, including mocks.

You can alternatively use lambdas as well:

```
@context
def before_hooks(context):

  context.before(lambda self: self.value = [])
```

After

The after hook is pretty much the opposite of before hooks: they are called *after* each example, in the **opposite** order defined:

```
from testslide.dsl import context
import os

@context
def After_hooks(context):

  @context.after
  def do_call(self):
    os.remove('/tmp/something')

  @context.example
  def passes(self):
    self.mock_callable(os, 'remove')\
      .for_call('/tmp/something')\
      .to_return_value(None)\
      .and_assert_called_once()

  @context.example
  def falis(self):
    self.mock_callable(os, 'remove')\
      .for_call('/tmp/WRONG')\
      .to_return_value(None)\
      .and_assert_called_once()
```

After hooks are typically used for:

- Executing things common to all examples (eg: calling the code that is being tested).
- Doing assertions common to all examples.
- Doing cleanup logic (eg: closing file descriptors).

You can also define after hooks from within examples:

```
@context.example
def can_define_after_hook(self):
  do_first_thing()

  @self.after
  def run_after_example_finishes():
    do_something_after_last_thing()

  do_last_thing()
```

Will run `do_first_thing`, `do_last_thing` **then** `do_something_after_last_thing`.

Aggregated failures

One important behavior of after hooks, is that they are **always** executed, regardless of any other failures in the test. This means, we get detailed result of each after hook failure:

```
from testslide.dsl import context

@context
def Show_aggregated_failures(context):

  @context.example
  def example_with_after_hooks(self):
    @self.after
    def assert_something(self):
      assert 1 == 2

    @self.after
    def assert_other_thing(self):
      assert 1 == 3
```

And its output:

```
Show aggregated failures
  example with after hooks: FAIL: AggregatedExceptions: empty example

Failures:

  1) Show aggregated failures: example with after hooks
    1) AssertionError:
      (...)
    2) AssertionError:
      (...)

Finished 1 examples in 0.0s:
  Failed: 1
```

Around

Around hooks execute around all **before hooks**, **example code** and all **after hooks**:

```
from testslide.dsl import context
import os, tempfile

@context
def Around_hooks(context):

  @context.around
  def inside_tmp_dir(self, example):
    with tempfile.TemporaryDirectory() as path:
      self.path = path
      original_path = os.getcwd()
      try:
        os.chdir(path)
        example()
      finally:
        os.chdir(original_path)

  @context.example
  def code_inside_temporary_dir(self):
    assert os.getcwd() == self.path
```

In this example, every example in the context will run inside a temporary directory.

If you declare multiple around hooks, the first around hook wraps the next one and so on.

Typical use for around hooks are similar to when context manager would be useful:

- Rolling back DB transactions after each test.
- Closing open file descriptors.
- Removing temporary files.

1.5.4 Context Attributes and Functions

Other than *Context Hooks*, you can also configure contexts with any attributes or functions.

Attributes

You can set any arbitrary attribute from within any hook:

```
@context.before
def before(self):
  self.calculator = Calculator()
```

and refer it later on:

```
@context.example
def is_a_calculaor(self):
  assert type(self.calculator) == Calculator
```

Memoized Attributes

Memoized attributes allow for lazy construction of attributes needed during a test. The attribute value will be constructed and remembered only at the first attribute access:

```
@context
def Memoized_attributes(context):

  # This function will be used to lazily set a memoized attribute with the same name
  @context.memoize
  def memoized_value(self):
    return []

  # Lambdas are also OK
  context.memoize('another_memoized_value', lambda self: [])

  # Or in bulk
  context.memoize(
    yet_another=lambda self: 'one',
    and_one_more=lambda self: 'attr',
  )

  @context.example
  def can_access_memoized_attributes(self):
    # memoized_value
    assert len(self.memoized_value) == 0
    self.memoized_value.append(True)
```

(continues on next page)

(continued from previous page)

```
    assert len(self.memoized_value) == 1

    # another_memoized_value
    assert len(self.another_memoized_value) == 0
    self.another_memoized_value.append(True)
    assert len(self.another_memoized_value) == 1

    # these were declared in bulk
    assert self.yet_anoter == 'one'
    assert self.and_one_more == 'attr'
```

Note in the example that the list built by `memoized_value()`, is memoized, and is the same object for every access.

Another option is to force memoization to happen at a before hook, instead of at the moment the attribute is accessed:

```
@context.memoize_before
def attribute_name(self):
  return []
```

In this case, the attribute will be set, regardless if it is used or not.

Composition

The big value of using memoized attributes as opposed to a regular attribute, is that you can easily do composition:

```
from testslide.dsl import context
from testslide import StrictMock

@context
def Composition(context):

  context.memoize('attr_value', lambda self: 'default value')

  @context.memoize
  def mock(self):
    mock = StrictMock()
    mock.attr = self.attr_value
    return mock

  @context.example
  def sees_default_value(self):
    self.assertEqual(self.mock.attr, 'default value')

  @context.sub_context
  def With_different_value(context):

    context.memoize('attr_value', lambda self: 'different value')

    @context.example
    def sees_different_value(self):
      self.assertEqual(self.mock.attr, 'different value')
```

Functions

You can define arbitrary functions that can be called from test code with the `@context.function` decorator:

```
@context
def Arbitrary_helper_functions(context):

  @context.memoize
  def some_list(self):
    return []

  # You can define arbitrary functions to call later
  @context.function
  def my_helper_function(self):
    self.some_list.append('item')
    return "I'm helping!"

  @context.example
  def can_call_helper_function(self):
    assert "I'm helping!" == self.my_helper_function()
    assert ['item'] == self.some_list
```

1.5.5 Skip and Focus

The *Test Runner* supports focusing and skipping examples. Let's see how to do it with TestSlide's DSL.

Focus

You can focus either the top level context, sub contexts or examples by prefixing their declaration with a `f`:

```
from testslide.dsl import context, fcontext, xcontext

@context
def Focusing(context):

  @context.example
  def not_focused_example(self):
    pass

  @context.fexample
  def focused_example(self):
    pass

  @context.sub_context
  def Not_focused_subcontext(context):

    @context.example
    def not_focused_example(self):
      pass

  @context.fsub_context
  def Focused_context(context):

    @context.example
    def inherits_focus_from_context(self):
      pass
```

And when run with `--focus`:

```
Focusing
  *focused example: PASS
  *Focused context
    *inherits focus from context: PASS

Finished 2 example(s) in 0.0s:
  Successful:  2
  Not executed:  2
```

Skip

Skipping works just the same, but you have to use a `x`:

```
from testslide.dsl import context, fcontext, xcontext

@context
def Skipping(context):

  @context.example
  def not_skipped_example(self):
    pass

  @context.xexample
  def skipped_example(self):
    pass

  @context.sub_context
  def Not_skipped_subcontext(context):

    @context.example
    def not_skipped_example(self):
      pass

  @context.xsub_context
  def Skipped_context(context):

    @context.example
    def inherits_skip_from_context(self):
      pass
```

```
Skipping
  not skipped example: PASS
  skipped example: SKIP
  Not skipped subcontext
    not skipped example: PASS
  Focused context
    inherits focus from context: SKIP

Finished 4 example(s) in 0.0s:
  Successful:  2
  Skipped:  2
```

1.5.6 `unittest.TestCase` Integration

TestSlide's DSL builtin integration with Python's unittest.

Assertions

TestSlide (currently) has on assertion framework. It comes however, with all `self.assert*` methods that you find at `unittest.TestCase` (see the docs):

```
@context
def unittest_assert_methods(context):

  @context.example
  def has_assert_true(self):
    self.assertTrue(True)
```

Reusing existing `unittest.TestCase` setUp

You can leverage existing `unittest.TestCase` classes, and use their setup logic to with TestSlide's DSL:

```
@context
def merging_test_cases(context):

  context.merge_test_case(SomePreExistingTestCase, 'legacy_test_case')

  @context.example
  def can_access_the_test_case(self):
    self.legacy_test_case  # => SomePreExistingTestCase instance
```

`merge_test_case` will call all `SomePreExistingTestCase` test hooks (`setUp`, `tearDown` etc) for each example.

From each example (or hooks), you will have access to the `TestCase` instance, so you can access any of its methods or attributes.

Note: Only hooks are executed, no existing tests will be imported!

1.6 Code Snippets

Here are code snippets, to save you time when writing tests.

1.6.1 Atom

Please refer Atom's documentation on how to use these.

```
'.source.python':
  ##
  ## TestSlide
  ##

  # Context
  '@context':
    'prefix': 'cont'
    'body': '@context\ndef ${1:context_description}(context):\n    ${2:pass}'
  '@fcontext':
```

(continues on next page)

(continued from previous page)

```
    'prefix': 'fcont'
    'body': '@fcontext\ndef ${1:context_description}(context):\n    ${2:pass}'
  '@xcontext':
    'prefix': 'xcont'
    'body': '@xcontext\ndef ${1:context_description}(context):\n    ${2:pass}'
  '@context.sub_context':
    'prefix': 'scont'
    'body': '@context.sub_context\ndef ${1:context_description}(context):\n    $
↪{2:pass}'
  '@context.fsub_context':
    'prefix': 'fscont'
    'body': '@context.fsub_context\ndef ${1:context_description}(context):\n    $
↪{2:pass}'
  '@context.xsub_context':
    'prefix': 'xscont'
    'body': '@context.xsub_context\ndef ${1:context_description}(context):\n    $
↪{2:pass}'
  '@context.shared_context':
    'prefix': 'shacont'
    'body': '@context.shared_context\ndef ${1:shared_context_description}(context):\n␣
↪    ${2:pass}'

  # Example
  '@context.example':
    'prefix': 'exp'
    'body': '@context.example\ndef ${1:example_description}(self):\n    ${2:pass}'
  '@context.fexample':
    'prefix': 'fexp'
    'body': '@context.fexample\ndef ${1:example_description}(self):\n    ${2:pass}'
  '@context.xexample':
    'prefix': 'xexp'
    'body': '@context.xexample\ndef ${1:example_description}(self):\n    ${2:pass}'

  # Hooks
  '@context.before':
    'prefix': 'befo'
    'body': '@context.before\ndef ${1:before}(self):\n    ${2:pass}'
  '@context.after':
    'prefix': 'aft'
    'body': '@context.after\ndef ${1:after}(self):\n    ${2:pass}'
  '@context.around':
    'prefix': 'aro'
    'body': '@context.around\ndef ${1:around}(self, bef_aft_example):\n    ${2:pass
↪# before example}\n    bef_aft_example()\n    ${3:pass  # after example}'

  # Attributes
  '@context.memoize':
    'prefix': 'memo'
    'body': '@context.memoize\ndef ${1:attribute_name}(self):\n    ${2:pass}'
  '@context.function':
    'prefix': 'cfunc'
    'body': '@context.function\ndef ${1:function_name}(self):\n    ${2:pass}'
```