TestSlide Documentation

Release 2.5.7

Fabio Pugliese Ornellas

Jul 16, 2020

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 *patch_attribute()*, *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 --list 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 --focus 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.

When you are committing tests to a continuous integration system, focusing tests may not be the best choice. You can use the cli option `--fail-if-focused` which will cause TestSlide to fail if any focused examples are run.

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 Path Simplification

The option `--trim-path-prefix` selects a path prefix to remove from stack traces and error messages. This makes parsing error messages easier. It defaults to the working directory, so there's seldom need to tweak it.

1.1.6 Internal Stack Trace

By default, stack trace lines that are from TestSlide's code base are hidden, as they are only useful when debugging TestSlide itself. You can see them if you wish, by using `--show-testslide-stack-trace`.

1.1.7 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`.

1.1.8 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.

1.1.9 Code Coverage

Coverage.py integration is simple. Make sure your `.coveragerc` file has this set:

```
[run]
parallel = True
```

and then you can run all your tests and get a report like this

```
$ coverage erase
$ COVERAGE_PROCESS_START=.coveragerc testslide some.py tests.py
$ COVERAGE_PROCESS_START=.coveragerc testslide some_more_tests.py
$ coverage combine
$ coverage report
```

1.1.10 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

Often code we write depends on external things such as a database or a REST API. We can test our code allowing it to talk directly to those dependencies, but there are different reasons why we wouldn't want to:

- The dependency is available as a production environment only and we can't let a test risk breaking production.
- The dependency is not available on all environments the test is being executed, for example during a Continuous Integration build.
- We want to test different scenarios, such as a valid response, error response or a timeout.

Mocks helps us achieve this goal when used in place of a real dependency. They need to respond conforming to the same interface exposed by the dependency, allowing us to configure canned responses to simulate the different scenarios we need. This must be true if we want to trust our test results.

1.2.1 Yet Another Mock?

Python unittest already provides us with `Mock`, `PropertyMock`, `AsyncMock`, `MagicMock`, `NonCallableMagicMock`... each for a specific use case. To understand what `StrictMock` brings to the table, let's start by looking at Python's mocks.

Let's pretend we depend on a `Calculator` class and we want to create a mock for it:

```
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)

```

(continues on next page)

(continued from previous page)

```
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'>
```

Wow! The calculator mock is lying to us telling that 2 is odd! And worse: we are able to violate the method signature without issues! How can we trust our tests with mocks like this? **This is precisely the kind of problem `StrictMock` solves!**

Note: Since Python 3.7 we can seal mocks. This helps, but as you will see, `StrictMock` has a lot unpaired functionality.

1.2.2 Thorough API Validations

`StrictMock` does a lot of validation under the hood to ensure you are configuring your mocks in conformity with the given template class interface. This has obvious immediate advantages, but is surprisingly helpful in catching bugs when refactoring happens (eg: the interface of the template class changed).

Safe By Default

`StrictMock` allows you to create **mocks of instances of a given template class**. Its default is **not** to give arbitrary canned responses, but rather be clear that it is missing some configuration:

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

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

In [3]: mock = StrictMock(template=Calculator)

In [4]: mock.is_odd(2)
(...)
UndefinedAttribute: 'is_odd' is not defined.
<StrictMock 0x7F17E06C7310 template=__main__.Calculator> must have a value defined␣
↪for this attribute if it is going to be accessed.
```

So, let's define `is_odd` method:

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

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

Any undefined attribute access will raise `UndefinedAttribute`. As you are in control of what values you assign to your mock, you can trust it to do only what you expect it to do.

Note:

- Refer to *mock_callable()* to learn to tighten what arguments `is_odd()` should accept.
- Refer to *mock_constructor()* to learn how to put `StrictMock` in place of your dependency.

Safe Magic Methods Defaults

Any magic methods defined at the template class will also have the safe by default characteristic:

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

In [2]: class NotGreater:
   ...:     def __gt__(self, other):
   ...:         return False
   ...:

In [3]: mock = StrictMock(template=NotGreater)

In [4]: mock > 0
(...)
UndefinedAttribute: '__gt__' is not set.
<StrictMock 0x7FE849B5DCD0 template=__main__.NotGreater> must have a value set for␣
↪this attribute if it is going to be accessed.
```

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(template=Calculator)

In [4]: mock.invalid
(...)
AttributeError: 'invalid' was not set for <StrictMock 0x7F4C62423F10 template=__main__
↪.Calculator>.

In [4]: mock.invalid = "whatever"
(...)
CanNotSetNonExistentAttribute: 'invalid' can not be set.
<StrictMock 0x7F4C62423F10 template=__main__.Calculator> template class does not have␣
↪this attribute so the mock can not have it as well.
See also: 'runtime_attrs' at StrictMock.__init__.
```

Dynamic Attributes

This validation works even for attributes set by `__init__`, as `StrictMock` introspects the code to learn about them:

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

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

In [3]: mock = StrictMock(template=DynamicAttr)

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

Attribute Type

When type annotation is available for attributes, `StrictMock` won't allow setting it with an invalid type:

```
In [1]: import testslide

In [2]: class Calculator:
   ...:     VERSION: str = "1.0"
   ...:

In [3]: mock = testslide.StrictMock(template=Calculator)

In [4]: mock.VERSION = "1.1"

In [5]: mock.VERSION = 1.2
(...)
TypeError: type of VERSION must be str; got float instead
```

Method Signature

Method signatures must match the signature of the equivalent method at 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(template=Calculator)

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

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

Method Argument Type

Methods with type annotation will have call arguments validated against it and invalid types will raise:

```
In [1]: import testslide

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

In [3]: mock = testslide.StrictMock(template=Calculator)

In [4]: mock.is_odd = lambda x: True

In [5]: mock.is_odd(1)
Out[5]: True

In [6]: mock.is_odd("1")
(...)
TypeError: Call with incompatible argument types:
  'x': type of x must be int; got str instead
```

Method Return Type

Methods with return type annotated will have its return value type validated as well:

```
In [1]: import testslide

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

In [3]: mock = testslide.StrictMock(template=Calculator)

In [4]: mock.is_odd = lambda x: 1
(...)
TypeError: type of return must be bool; got int instead
```

Setting Methods With Callables

If the Template class attribute is a instance/class/static method, `StrictMock` will only allow callable values to be assigned:

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

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

In [3]: mock = StrictMock(template=Calculator)

In [4]: mock.is_odd = "not callable"
```

(continues on next page)

(continued from previous page)

```
(...)
NonCallableValue: 'is_odd' can not be set with a non-callable value.
<StrictMock 0x7F4C62423F10 template=__main__.Calculator> template class requires this␣
↪attribute to be callable.
```

Setting Async Methods With Coroutines

Coroutine functions (`async def`) (whether instance, class or static methods) can only have a callable that returns an awaitable assigned:

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

In [2]: class AsyncMethod:
   ...:     async def async_instance_method(self):
   ...:         pass
   ...:

In [3]: mock = StrictMock(template=AsyncMethod)

In [4]: def sync():
   ...:     pass
   ...:

In [5]: mock.async_instance_method = sync

In [6]: import asyncio

In [7]: asyncio.run(mock.async_instance_method())
(...)
NonAwaitableReturn: 'async_instance_method' can not be set with a callable that does␣
↪not return an awaitable.
<StrictMock 0x7FACF5A974D0 template=__main__.AsyncMethod> template class requires␣
↪this attribute to be a callable that returns an awaitable (eg: a 'async def'␣
↪function).
```

1.2.3 Configuration

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'>"
```

Template Class

By giving a template class, we can leverage all interface validation goodies:

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

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

In [3]: mock = StrictMock(template=Calculator)

In [4]: mock.is_odd(2)
(...)
UndefinedAttribute: 'is_odd' is not defined.
<StrictMock 0x7F17E06C7310 template=__main__.Calculator> must have a value defined␣
↪for this attribute if it is going to be accessed.
```

Generic Mocks

It is higly recommended to use `StrictMock` giving it a template class, so you can leverage its interface 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
(...)
UndefinedAttribute: 'whatever' is not defined.
<StrictMock 0x7FED1C724C18> must have a value defined for this attribute if it is␣
↪going to be accessed.

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

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

It will accept setting any attributes, with any values.

Setting Regular Attributes

They can be set as usual:

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

In [2]: mock = StrictMock()

In [3]: mock.whatever
(...)
UndefinedAttribute: 'whatever' is not defined.
<StrictMock 0x7FED1C724C18> must have a value defined for this attribute if it is␣
↪going to be accessed.

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

(continues on next page)

(continued from previous page)

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

Other than if the attribute is allowed to be set (based on the optional template class), no validation is performed on the value assigned.

Setting Methods

You can assign callables to instance, class and static methods as usual. There's special mechanics under the hood to ensure the mock will receive the correct arguments:

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

In [2]: class Echo:
   ...:   def instance_echo(self, message):
   ...:     return message
   ...:
   ...:   @classmethod
   ...:   def class_echo(cls, message):
   ...:     return message
   ...:
   ...:   @staticmethod
   ...:   def static_echo(message):
   ...:     return message
   ...:

In [3]: mock = StrictMock(template=Echo)
   ...:

In [4]: mock.instance_echo = lambda message: f"mock: {message}"
   ...:

In [5]: mock.instance_echo("hello")
   ...:
Out[5]: 'mock: hello'

In [6]: mock.class_echo = lambda message: f"mock: {message}"
   ...:

In [7]: mock.class_echo("hello")
   ...:
Out[7]: 'mock: hello'

In [8]: mock.static_echo = lambda message: f"mock: {message}"
   ...:

In [9]: mock.static_echo("hello")
   ...:
Out[9]: 'mock: hello'
```

You can also use regular methods:

```
In [11]: def new(message):
    ...:     return f"new {message}"
```

(continues on next page)

(continued from previous page)

```
    ...:

In [12]: mock.instance_echo = new

In [13]: mock.instance_echo("Hi")
Out[13]: 'new Hi'
```

Or even methods from any instances:

```
In [14]: class MockEcho:
    ...:     def echo(self, message):
    ...:         return f"MockEcho {message}"
    ...:

In [15]: mock.class_echo = MockEcho().echo

In [16]: mock.class_echo("Wow!")
Out[16]: 'MockEcho Wow!'
```

Setting Magic Methods

Magic Methods must be defined at the instance's class and not the instance. `StrictMock` has special mechanics that allow you to set them **per instance** trivially:

```
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'
```

Runtime Attributes

`StrictMock` introspects the template's `__init__` code using some heuristics to find attributes that are dynamically set during runtime. If this mechanism fails to detect a legit attribute, you should inform `StrictMock` about them:

```
StrictMock(template=TemplateClass, runtime_attrs=['attr_set_at_runtime'])
```

Default Context Manager

If the template class is a context manager, `default_context_manager` can be used to automatically setup `__enter__` and `__exit__` mocks for you:

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

In [2]: class CM:
   ...:   def __enter__(self):
   ...:     return self
   ...:
   ...:   def __exit__(self, exc_type, exc_value, traceback):
```

(continues on next page)

(continued from previous page)

```
   ...:     pass
   ...:

In [3]: mock = StrictMock(template=CM, default_context_manager=True)

In [4]: with mock as m:
   ...:   assert id(mock) == id(m)
   ...:
```

The mock itself is yielded.

Note: This also works for asynchronous context managers.

Signature Validation (and Attribute Types)

By default, `StrictMock` will validate arguments passed to callable attributes and the return value when called. This is done by inserting a proxy object in between the attribute and the value. In some rare situations, this proxy object can cause issues (eg if you `assert type(self.attr) == Foo`). If having `type()` return the correct value is more important than having API validation, you can disable them:

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

In [2]: class CallableObject(object):
   ...:   def __call__(self):
   ...:     pass
   ...:

In [3]: s = StrictMock()

In [4]: s.attr = CallableObject()

In [5]: type(s.attr)
Out[5]: testslide.strict_mock._MethodProxy

In [6]: s = StrictMock(signature_validation=False, type_validation=False)

In [7]: s.attr = CallableObject()

In [8]: type(s.attr)
Out[8]: __main__.CallableObject
```

Type Validation

By default, `StrictMock` will validate types of set attributes, method call arguments and method return values, against available type hinting information.

If this type validation yields bad results (likely a bug, please report it), you can disable it with:

```
StrictMock(template=SomeClass, type_validation=False)
```

1.2.4 Misc Functionality

- `copy.copy()` and `copy.deepcopy()` works, and give back another StrictMock, with the same behavior.
- Template classes that use `__slots__` are supported.

1.3 Patching

StrictMock solves the problem of having mocks that behave like the real thing. To really accomplish that we need a way of defining what a mocked method call will return. We also need a way of putting the mock in place of real objects. These are problems solved with **patching**.

TestSlide provides patching tools specialized in different problems. They are not only useful to configure `StrictMock`, but any Python object, including "real" ones, like a database connection. You can configure canned responses for specific calls, simulate network timeouts or anything you may need for your test.

Here's a summary of the patching tools available either via `testslide.TestCase` or *TestSlide's DSL*. Also check the comprehensive *Cheat Sheet*.

patch_attribute() Changes the value of an attribute. Eg:

```
self.patch_attribute(math, "pi", 3)
math.pi  # => 3
```

mock_callable()* / *mock_async_callable() Similar to `patch_attribute()` but designed to work with sync/async **functions/methods**. It creates and patches mocks for callables, which implements call **arguments constraints**, different call **behaviors** (return value, raise exception etc) and optional **call assertions**. Eg:

```
self.mock_callable("os.path", "exists")\
  .for_call("/bin")\
  .to_return_value(False)
os.path.exists("/bin")  # => False
```

mock_constructor() Allows classes to return mocks when new instances are created instead of real instances. It has the same fluid interface as `mock_callable()`/`mock_async_callable()`. Eg:

```
popen_mock = StrictMock(template=subprocess.Popen)
self.mock_constructor(subprocess, "Popen")\
  .for_call(["/bin/true"])\
  .to_return_value(popen_mock)
subprocess.Popen(["/bin/true"])  # => popen_mock
```

1.3.1 patch_attribute()

`patch_attribute()` will, for the duration of the test, change the value of a given attribute:

```
import math

class ChangePi(TestCase):
  def test_pi(self):
    self.patch_attribute(math, "pi", 4)
    self.assertEqual(math.pi, 4)
```

`patch_attribute()` works exclusively with **non-callable** attributes.

Note: TestSlide provides *mock_callable()*, *mock_async_callable()* and *mock_constructor()* for callables and classes because those require specific functionalities.

You can use `patch_attribute()` with:

- Modules.
- Classes.
- Instances of classes.
- Class attributes at instances of classes.
- Properties at instances of classes.

Properties are tricky to patch because of the quirky mechanics that Python's Descriptor Protocol requires. `patch_attribute()` has support for that so things "just work":

```
class WithProperty:
  @property
  def prop(self):
    return "prop"

class PatchingProperties(TestCase):
  def test_property(self):
    with_property = WithProperty()
    self.patch_attribute(with_property, "prop", "mock")
    self.assertEqual(with_property.prop, "mock")
```

1.3.2 mock_callable()

patch_attribute() deals with non-callable attributes. `mock_callable()` specializes on patching and 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)\
```

(continues on next page)

(continued from previous page)

```
      .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.

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?

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.

Note: Also check *Argument Matchers*: they allow more relaxed argument matching like "any string matching this regexp" or "any positive number".

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
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.

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_call('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_call('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
```

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 the easy assertion on such calls, as many times as needed within the same test.

Number of Calls

This will assert that the call was made exactly one time:

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

Alternatively you may define an arbitrary exact number of calls, minimum, maximum or that no call should happen:

```
.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()
```

Call Order

Frequently the order in which calls happen does not matter, but there are cases where this is desirable.

For example, let's say we want to ensure that some asset is first deleted from a storage index and then removed from the backend, thus avoiding the window of it being indexed, but unavailable at the backend. Here's how to do it:

```
self.mock_callable(storage_index, "delete")\
  .for_call_(asset_id)\
  .and_assert_called_ordered()
self.mock_callable(storage_backend, "delete")\
  .for_call_(asset_id)\
  .and_assert_called_ordered()
```

For this test to pass, these calls must happen exactly in this order:

```
storage_index.delete(asset_id)
storage_backend.delete(asset_id)
```

The test will fail if these calls are made in a different order or if they don't happen at all.

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_called_ordered()
  .and_assert_not_called()
```

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

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.

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.

Type Validation

If typing annotation information is available, `mock_callable()` validates types of objects passing through the mock. If an invalid type is detected, it will raise `TypeError`.

This feature is enabled by default. If you need to disable it (potentially due to a bug, please report!), you can do so by `mock_callable(target, name, type_validation=False)`.

Call Argument Types

```
import testslide

class SomeClass:
    def some_method(self, message: str):
        return "world"

class TestArgumentTypeValidation(testslide.TestCase):
    def test_argument_type_validation(self):
        some_class_instance = SomeClass()
        self.mock_callable(some_class_instance, "some_method").to_return_value(
            "mocked world"
        )
        self.assertEqual(some_class_instance.some_method("hello"), "mocked world")
        with self.assertRaises(TypeError):
            # TypeError: Call with incompatible argument types:
            # 'message': type of message must be str; got int instead
            some_class_instance.some_method(1)
```

Return Value Type

```
import testslide

class SomeClass:
```

(continues on next page)

(continued from previous page)

```
    def one(self) -> int:
        return 1

class TestReturnTypeValidation(testslide.TestCase):
    def test_return_type_validation(self):
        some_class_instance = SomeClass()
        self.mock_callable(some_class_instance, "one").to_return_value(
            "one"
        )
        with self.assertRaises(TypeError):
            # TypeError: type of return must be int; got str instead
            some_class_instance.one()
```

Limitations

Currently *TypeVar* annotations are not being checked for.

Test Framework Integration

TestSlide's DSL

Integration comes out of the box for *TestSlide's DSL*: you can simply do `self.mock_callable()` from inside examples or hooks.

Python Unittest

`testslide.TestCase` is provided with off the shelf integration ready:

- Inherit your `unittest.TestCase` from it.
- If you overload `unittest.TestCase.setUp`, make **sure** to call `super().setUp()` before using `mock_callable()`.

Any Test Framework

You must follow these steps for **each** test executed that uses `mock_callable()`:

- 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.

1.3.3 mock_async_callable()

Just like *mock_callable()* works with regular callables, `mock_async_callable()` works with coroutine functions. It implements virtually the same interface (including with all its goodies), with only the following minor differences.

`.with_implementation()`

It requires an async function:

```
async def async_func():
  return 33

self.mock_async_callable(some_object, 'some_method_name')\
  .with_implementation(async_func)
await some_object.some_method_name()  # => 33
```

`.with_wrapper()`

It requires an async function:

```
async def async_trim_query(original_async_callable):
  return await original_async_callable()[0:5]

self.mock_async_callable(some_service, 'big_query')\
  .with_wrapper(async_trim_query)
await some_service.big_query()  # => returns trimmed list
```

Implicit Coroutine Return

`mock_async_callable()` checks if what it is mocking is a coroutine function and refuses to mock if it is not. This is usually a good thing, as it prevents mistakes. In some cases, such as the ones related to this cython issue, this check can fail.

If you are trying to mock some callable with it, that is not a coroutine function, but you are **sure** that it returns a coroutine when called, you can still mock it like this:

```
self.mock_async_callable(
  target,
  "sync_callable_that_returns_a_coroutine",
  callable_returns_coroutine=True,
)
```

Test Framework Integration

Follows the exact same model as `mock_callable()`, but it should be invoked as `testslide.mock_callable.mock_async_callable`.

1.3.4 mock_constructor()

Let's say we want 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.

Note: Also check *Argument Matchers*: they allow more relaxed argument matching like "any string matching this regexp" or "any positive number".

Type Validation

`mock_constructor()` uses type annotation information from constructors to validate that mocks are respecting the interface:

```
import sys
import testslide

class Messenger:
    def __init__(self, message: str):
      self.message = message

class TestArgumentTypeValidation(testslide.TestCase):
    def test_argument_type_validation(self):
        messenger_mock = testslide.StrictMock(template=Messenger)
        self.mock_constructor(sys.modules[__name__], "Messenger").to_return_
↪value(messenger_mock)
        with self.assertRaises(TypeError):
            # TypeError: Call with incompatible argument types:
            # 'message': type of message must be str; got int instead
            Messenger(message=1)
```

If you need to disable it (potentially due to a bug, please report!) you can do so with: `mock_constructor(module, class_name, type_validation=False)`.

Caveats

Because of the way `mock_constructor()` must be implemented (see next section), its usage must respect these rules:

- References to the mocked class saved prior to `mock_constructor()` invocation **can not be used**, including previously created instances.
- Access to the class must happen exclusively via attribute access (eg: `getattr(some_module, "SomeClass")`).

A simple easy way to ensure this is to always:

```
# Do this:
import some_module
some_module.SomeClass
# Never do:
from some_module import SomeClass
```

Note: Not respecting these rules will break `mock_constructor()` and can lead to unpredicted behavior!

Implementation Details

`mock_callable()` should be all you need:

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

However, as of July 2019, Python 3 has an open bug https://bugs.python.org/issue25731 that prevents `__new__` from being patched. `mock_constructor()` is a way around this bug.

Because `__new__` can not be patched, we need to handle things elsewhere. The trick is to dynamically create a subclass of the target class, make the changes to `__new__` there (so we don't touch `__new__` at the target class), and patch it at the module in place of the original class.

This works when `__new__` simply returns a mocked value, but creates issues when used with `.with_wrapper()` or `.to_call_original()` as both requires calling the original `__new__`. This will return an instance of the original class, but the new subclass is already patched at the module, thus `super()` / `super(Class, self)` breaks. If we make them call `__new__` from the subclass, the call comes from... `__new__` and we get an infinite loop. Also, `__new__` calls `__init__` unconditionally, not allowing `.with_wrapper()` to mangle with the arguments.

The way around this, is to keep the original class where it is and move all its attributes to the child class:

- Dynamically create the subclass of the target class, with the same name.
- Move all `__dict__` values from the target class to the subclass (with a few exceptions, such as `__new__` and `__module__`).
- At the subclass, add a `__new__` that works as a factory, that allows `mock_callable()` interface to work.
- Do some trickery to fix the arguments passed to `__init__` to allow `.with_wrapper()` mangle with them.
- Patch the subclass in place of the original target class at its module.
- Undo all of this when the test finishes.

This essentially creates a "copy" of the class, at the subclass, but with `__new__` implementing the behavior required. All things such as class attributes/methods and `isinstance()` are not affected. The only noticeable difference, is that `mro()` will show the extra subclass.

Test Framework Integration

TestSlide's DSL

Integration comes out of the box for *TestSlide's DSL*: you can simply do `self.mock_constructor()` from inside examples or hooks.

Python Unittest

`testslide.TestCase` is provided with off the shelf integration ready:

- Inherit your `unittest.TestCase` from it.
- If you overload `unittest.TestCase.setUp`, make **sure** to call `super().setUp()` before using `mock_constructor()`.

Any Test Framework

You must follow these steps for **each** test executed that uses `mock_constructor()`:

- Integrate *mock_callable()* (used by mock_constructor under the hood).
- 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.

1.3.5 Argument Matchers

mock_callable(), *mock_async_callable()* and *mock_constructor()* allow the definitions of what call arguments to accept by using `.for_call()`. Eg:

```
self.mock_constructor(storage, 'Client')\
  for_call(timeout=60)\
  to_return_value(self.storage_mock)
```

This validation is strict: tests will work with `Client(timeout=60)` but fail with `Client(timeout=61)`. Perhaps letting tests pass with "any positive integer" would be enough. This is precisely what **argument matchers** allow us to do:

```
from testslide.matchers import IntGreaterThan
(...)
self.mock_constructor(storage, 'Client')\
  for_call(timeout=IntGreaterThan(0))\
  to_return_value(self.storage_mock)
```

This matches for `Client(timeout=5)`, `Client(timeout=60)` but not for `Client(timeout=0)` or `Client(timeout=-1)`.

Logic Operations

Argument matchers can be combined using bitwise operators:

```
# String containing "this" AND ending with "that"
StrContaining("this") & StrEndingWith("that")
# String containing "this" OR ending with "that"
StrContaining("this") | StrEndingWith("that")
# String containing "this" EXCLUSIVE OR ending with "that"
StrContaining("this") ^ StrEndingWith("that")
# String NOT containing "this"
~StrContaining("this")
```

Integers

Matcher	Description
`AnyInt()`	Any `int`
`NotThisInt(value)`	Any integer but the given value
`IntBetween(min_value, max_valu)`	Integer `>= min_value` and `<= max_value`
`IntGreaterThan(value)`	Integer `> value`
`IntGreaterOrEquals(value)`	Integer `>= value`
`IntLessThan(value)`	Integer `< value`
`IntLessOrEquals(value)`	Integer `<= value`

Floats

Matcher	Description
`AnyFloat()`	Any `float`
`NotThisFloat(value)`	Any float but the given value
`FloatBetween(min_value, max_valu)`	Float `>= min_value` and `<= max_value`
`FloatGreaterThan(value)`	Float `> value`
`FloatGreaterOrEquals(value)`	Float `>= value`
`FloatLessThan(value)`	Float `< value`
`FloatLessOrEquals(value)`	Float `<= value`

Strings

Matcher	Description
`AnyStr()`	Any `str`
`RegexMatches(pattern, flags=0)`	Any string that matches the regular expression compiled by `re.compile(pattern, flags)`
`StrContaining(text)`	A string which contains `text` in it
`StrStartingWith()`	A string that starts with `text`
`StrEndingWith(text)`	A string that ends with `text`

Lists

Matcher	Description
`AnyList()`	Any `list`
`ListContaining(element)`	Any list containing `element`
`ListContainingAll(element_list)`	Any list which contains every element of `element_list`
`NotEmptyList()`	A list which has at least one element
`EmptyList()`	An empty list: `[]`

Dictionaries

Matcher	Description
`AnyDict()`	Any `dict`
`NotEmptyDict()`	A dictionary with any at least one key
`EmptyDict()`	An empty dictionary: `{}`
`DictContainingKeys(keys_list)`	A dictionary containing all keys from `keys_list`
`DictSupersetOf(this_dict)`	A dictionary containing all key / value pairs from `this_dict`

Generic

Matcher	Description
`Any()`	Any object
`AnyTruthy()`	Any object where `bool(obj) == True`
`AnyFalsey()`	Any object where `bool(obj) == False`
`AnyInstanceOf()`	Any object where `isinstance(obj) == True`
`AnyWithCall(call)`	Any object where `call(obj) == True`

```
self.mock_callable(os, 'remove')\
  .for_call(AnyWithCall(lambda path: path.endswith("py"))\
  .to_return_value(None)\
  .and_assert_called_once()
```

1.3.6 Cheat Sheet

Here is a comprehensive list of use cases for all patching tools TestSlide offers and when to use each of them.

```
# module.py

# self.patch_attribute(module, "MODULE_ATTRIBUTE", "mock")
# module.MODULE_ATTRIBUTE  # => "mock"
MODULE_ATTRIBUTE = "..."

# self.mock_callable(module, "function_at_module")\
#  .for_call()\
#  .to_return_value(None)
# module.function_at_module()  # => "mock"
def function_at_module():
  pass

# self.mock_callable(module, "async_function_at_module")\
#  .for_call()\
#  .to_return_value("mock")
# await module.async_function_at_module()  # => "mock"
async def async_function_at_module():
  pass

# some_class_mock = testslide.StrictMock(template=module.SomeClass)
class SomeClass:
  # Patching here affects all instances of the class as well
  # self.patch_attribute(SomeClass, "CLASS_ATTRIBUTE", "mock")
  # module.SomeClass.CLASS_ATTRIBUTE  # => "mock"
  CLASS_ATTRIBUTE = "..."

  # self.mock_constructor(module, "SomeClass")\
  #   .for_call()\
  #   .to_return_value(some_class_mock)
  # module.SomeClass()  # => some_class_mock
  def __init__(self):
    # Must be patched at instances
    self.init_attribute = "..."

  # Must be patched at instances
```

(continues on next page)

(continued from previous page)

```
  @property
  def property(self):
    return "..."

  # Must be patched at instances
  def instance_method(self):
    pass

  # Must be patched at instances
  async def ainstance_method(self):
    pass

  # self.mock_callable(SomeClass, "class_method")\
  #   .for_call()\
  #   .to_return_value("mock")
  # module.SomeClass.class_method()  # => "mock"
  @classmethod
  def class_method(cls):
    pass

  # self.mock_async_callable(SomeClass, "async_class_method")\
  #   .for_call()\
  #   .to_return_value("mock")
  # await module.SomeClass.async_class_method()  # => "mock"
  @classmethod
  async def async_class_method(cls):
    pass

  # self.mock_callable(SomeClass, "static_method")\
  #   .for_call()
  #   .to_return_value("mock")
  # module.SomeClass.static_method()  # => "mock"
  @staticmethod
  def static_method(cls):
    pass

  # self.mock_async_callable(SomeClass, "async_static_method")\
  #   .for_call()
  #   .to_return_value("mock")
  # await module.SomeClass.async_static_method()  # => "mock"
  @staticmethod
  async def async_static_method(cls):
    pass

  # Must be patched at instances
  def __str__(self):
    return "SomeClass"

some_class_instance = SomeClass()

# self.patch_attribute(some_class_instance, "init_attribute", "mock")
some_class_instance.init_attribute  # => "mock"

# Patching at the instance does not affect other instances or the class
# self.patch_attribute(some_class_instance, "CLASS_ATTRIBUTE", "mock")
some_class_instance.CLASS_ATTRIBUTE  # => "mock"
```

(continues on next page)

(continued from previous page)

```
# self.patch_attribute(some_class_instance, "property", "mock")
some_class_instance.property  # => "mock"

# self.mock_callable(some_class_instance, "instance_method")\
#   .for_call()\
#   .to_return_value("mock")
some_class_instance.instance_method()  # => "mock"

# self.mock_async_callable(some_class_instance, "async_instance_method")\
#   .for_call()\
#   .to_return_value("mock")
some_class_instance.async_instance_method()  # => "mock"

# self.mock_callable(some_class_instance, "class_method")\
#    .for_call()\
#    .to_return_value("mock")
some_class_instance.class_method()  # => "mock"

# self.mock_async_callable(some_class_instance, "async_class_method")
#    .for_call()\
#    .to_return_value("mock")
some_class_instance.async_class_method()  # => "mock"

# self.mock_callable(some_class_instance, "static_method")\
#   .for_call()\
#   .to_return_value("mock")
some_class_instance.static_method()  # => "mock"

# self.mock_async_callable(some_class_instance, "async_static_method")\
#   .for_call()\
#   .to_return_value("mock")
some_class_instance.async_static_method()  # => "mock"

# self.mock_callable(some_class_instance, "__str__")\
#    .for_call()\
#    .to_return_value("mock")
str(some_class_instance)  # => "mock"
```

1.4 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

@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.

1.4.1 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 version 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.

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.

1.4.2 Sharing Contexts

Shared contexts allows sharing of common logic across different contexts. When you declare a shared context, its contents won't be evaluated, unless you either merge or nest it elsewhere. Let's see it in action.

Merging

When you merge a shared context, its hooks and examples will be added to the existing context, alongside existing hooks and examples:

```
from testslide.dsl import context

@context
def Nesting_Shared_Contexts(context):

```

(continues on next page)

(continued from previous page)

```
    @context.shared_context
    def some_shared_things(context):

        @context.before
        def do_common_thing_before(self):
            pass

        @context.example
        def common_example(self):
            pass

    @context.sub_context
    def when_one_thing(context):
        context.merge_context('some shared things')

        @context.before
        def do_one_thing_before(self):
            pass

        @context.example
        def one_thing_example(self):
            pass

    @context.sub_context
    def when_another_thing(context):
        context.merge_context('some shared things')

        @context.before
        def do_another_thing_before(self):
            pass

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

Will result in:

```
Nesting Shared Contexts
  when one thing
    common example
    one thing example
  when another thing
    common example
    another thing example

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

Nesting

If you nest a shared context, another sub-context will be created, with the same name as the shared context, containing all the hooks and examples from the shared context:

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

(continues on next page)

(continued from previous page)

```
@context
def Nesting_Shared_Contexts(context):

    @context.shared_context
    def some_shared_things(context):

        @context.before
        def do_common_thing_before(self):
            pass

        @context.example
        def common_example(self):
            pass

    @context.sub_context
    def when_one_thing(context):
        context.nest_context('some shared things')

        @context.before
        def do_one_thing_before(self):
            pass

        @context.example
        def one_thing_example(self):
            pass

    @context.sub_context
    def when_another_thing(context):
        context.nest_context('some shared things')

        @context.before
        def do_another_thing_before(self):
            pass

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

Will result in:

```
Nesting Shared Contexts
  when one thing
    one thing example
    some shared things
      common example
  when another thing
    another thing example
    some shared things
      common example

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

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.

1.4.3 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)

  @context.before
```

(continues on next page)

(continued from previous page)

```
  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(self):
    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 wrap around all **before hooks**, **example code** and **after hooks**:

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

@context
def Around_hooks(context):

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

1.4.4 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)
    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):
```

(continues on next page)

(continued from previous page)

```
  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
```

1.4.5 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.example(skip=True)
```

(continues on next page)

(continued from previous page)

```
  def skipped_example_from_arg(self):
    pass

  @context.example(skip_unless=False)
  def skipped_example_from_unless_arg(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
  skipped example from arg: SKIP
  skipped example from unless arg: 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
```

1.4.6 `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!

1.4.7 Async Support

TestSlide's DSL supports asynchronous I/O testing.

For that, you must declare all of these as async:

- Hooks: around, before and after.
- Examples.
- Memoize before.
- Functions.

like this:

```
from testslide.dsl import context

@context
def testing_async_code(context):
  @context.around
  async def around(self, example):
    await example()  # Note that this must be awaited!

  @context.before
  async def before(self):
    pass

  @context.after
  async def after(self):
    pass

  @context.memoize_before
  async def memoize_before(self):
    return "memoize_before"
```

(continues on next page)

(continued from previous page)

```
  @context.function
  async def function(self):
    return "function"

  @context.example
  async def example(self):
    assert self.memoize_before == "memoize_before"
    assert self.function == "function"
```

The test runner will create a new event look to execute each example.

Note: You can **not** mix async and sync stuff for the same example. If your example is async, then all its hooks and memoize before must also be async.

Note: It is not possible to support async `@context.memoize`. They depend on __getattr__ to work, which has no async support. Use `@context.memoize_before` instead.

Event Loop Health

Event loops are the engine that runs Python async code. It works by alternating the execution between different bits of async code. Eg: when `await` is used, it allows the event loop to switch to another task. A requirement for this model to work is that async code must be "well behaved", so that it does what it needs to do without impacting other tasks.

TestSlide DSL has specific checks that detect if tested async code is doing something it should not.

Not Awaited Coroutine

Every called coroutine must be awaited. If they are not, it means their code never got to be executed, which indicates a bug in the code. In this example, a forgotten to be awaited coroutine triggers a test failure, despite the fact that no direct failure was reported by the test:

```
import asyncio
from testslide.dsl import context

@context
def Not_awaited_coroutine(context):
  @context.example
  async def awaited_sleep(self):
    await asyncio.sleep(1)

  @context.example
  async def forgotten_sleep(self):
    asyncio.sleep(1)
```

```
$ testslide not_awaited_coroutine.py
Not awaited coroutine
  awaited sleep
  forgotten sleep: RuntimeWarning: coroutine 'sleep' was never awaited

Failures:
```

(continues on next page)

(continued from previous page)

```
  1) Not awaited coroutine: forgotten sleep
    1) RuntimeWarning: coroutine 'sleep' was never awaited
    Coroutine created at (most recent call last)
      File "/opt/python/lib/python3.7/site-packages/testslide/__init__.py", line 394,␣
↪in run
        self._async_run_all_hooks_and_example(context_data)
      File "/opt/python/lib/python3.7/site-packages/testslide/__init__.py", line 334,␣
↪in _async_run_all_hooks_and_example
        asyncio.run(coro, debug=True)
      File "/opt/python/lib/python3.7/asyncio/runners.py", line 43, in run
        return loop.run_until_complete(main)
      File "/opt/python/lib/python3.7/asyncio/base_events.py", line 566, in run_until_
↪complete
        self.run_forever()
      File "/opt/python/lib/python3.7/asyncio/base_events.py", line 534, in run_
↪forever
        self._run_once()
      File "/opt/python/lib/python3.7/asyncio/base_events.py", line 1763, in _run_once
        handle._run()
      File "/opt/python/lib/python3.7/asyncio/events.py", line 88, in _run
        self._context.run(self._callback, *self._args)
      File "/opt/python/lib/python3.7/site-packages/testslide/__init__.py", line 244,␣
↪in _real_async_run_all_hooks_and_example
        self.example.code, context_data
      File "/opt/python/lib/python3.7/site-packages/testslide/__init__.py", line 218,␣
↪in _fail_if_not_coroutine_function
        return await func(*args, **kwargs)
      File "/home/fornellas/tmp/not_awaited_coroutine.py", line 12, in forgotten_sleep
        asyncio.sleep(1)
      File "/opt/python/lib/python3.7/contextlib.py", line 119, in __exit__
        next(self.gen)

Finished 2 example(s) in 1.0s:
  Successful: 1
  Failed: 1
```

Slow Callback

Async code must do their work in small chunks, properly awaiting other functions when needed. If an async function does some CPU intensive task that takes a long time to compute, or if it calls a sync function that takes a long time to return, the entirety of the event loop will be locked up. This means that no other code can be executed until this bad async function returns.

If during the test execution a task blocks the event loop, it will trigger a test failure, despite the fact that no direct failure was reported by the test:

```
import time
from testslide.dsl import context

@context
def Blocked_event_loop(context):
  @context.example
  async def blocking_sleep(self):
    time.sleep(1)
```

```
$ testslide blocked_event_loop.py
Blocked event loop
  blocking sleep: SlowCallback: Executing <Task finished coro=<_ExampleRunner._real_
↪async_run_all_hooks_and_example() done, defined at /opt/python/lib/python3.7/site-
↪packages/testslide/__init__.py:220> result=None created at /opt/python/lib/python3.
↪7/asyncio/base_events.py:558> took 1.002 seconds

Failures:

  1) Blocked event loop: blocking sleep
    1) SlowCallback: Executing <Task finished coro=<_ExampleRunner._real_async_run_
↪all_hooks_and_example() done, defined at /opt/python/lib/python3.7/site-packages/
↪testslide/__init__.py:220> result=None created at /opt/python/lib/python3.7/asyncio/
↪base_events.py:558> took 1.002 seconds
      During the execution of the async test a slow callback that blocked the event␣
↪loop was detected.
      Tip: you can customize the detection threshold with:
        asyncio.get_running_loop().slow_callback_duration = seconds
      File "/opt/python/lib/python3.7/contextlib.py", line 119, in __exit__
        next(self.gen)

Finished 1 example(s) in 1.0s:
  Failed: 1
```

Python's default threshold for triggering this event loop lock up failure is **100ms**. If your problem domain requires something smaller or bigger, you can easily customize it:

```
import asyncio
import time
from testslide.dsl import context

@context
def Custom_slow_callback_duration(context):
  @context.before
  async def increase_slow_callback_duration(self):
    loop = asyncio.get_running_loop()
    loop.slow_callback_duration = 2

  @context.example
  async def blocking_sleep(self):
    time.sleep(1)
```

```
$ testslide custom_slow_callback_duration.py
Custom slow callback duration
  blocking sleep

Finished 1 example(s) in 1.0s:
  Successful: 1
```

1.5 Code Snippets

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

1.5.1 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':
    '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}'
```

(continues on next page)

(continued from previous page)

```
  # 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}'
```