aspectlib

Release 1.1.1

June 25, 2014

Contents

`aspectlib` is an aspect-oriented programming, monkey-patch and decorators library. It is useful when changing behavior in existing code is desired.

CHAPTER 1

Introduction

aspectlib provides two core tools to do AOP: Aspects and a weaver.

1.1 The aspect

An *aspect* can be created by decorating a generator with an `Aspect`. The generator yields *advices* - simple behavior changing instructions.

An `Aspect` instance is a simple function decorator. Decorating a function with an *aspect* will change the function's behavior according to the *advices* yielded by the generator.

Example:

```
@aspectlib.Aspect
def strip_return_value():
    result = yield aspectlib.Proceed
    yield aspectlib.Return(result.strip())

@strip_return_value
def read(name):
    return open(name).read()
```

1.1.1 Advices

You can use these *advices*:

- `Proceed` or `None` - Calls the wrapped function with the default arguments. The *yield* returns the function's return value or raises an exception. Can be used multiple times (will call the function multiple times).
- `Proceed (*args, **kwargs)` - Same as above but with different arguments.
- `Return` - Makes the wrapper return `None` instead. If `aspectlib.Proceed` was never used then the wrapped function is not called. After this the generator is closed.
- `Return (value)` - Same as above but returns the given `value` instead of `None`.
- `raise exception` - Makes the wrapper raise an exception.

1.2 The weaver

Patches classes and functions with the given *aspect*. When used with a class it will patch all the methods. In AOP parlance these patched functions and methods are referred to as *cut-points*.

Returns a `Rollback` object that can be used a context manager. It will undo all the changes at the end of the context.

Example:

```
@aspectlib.Aspect
def mock_open():
    yield aspectlib.Return(StringIO("mystuff"))

with aspectlib.weave(open, mock_open):
    assert open("/doesnt/exist.txt").read() == "mystuff"
```

You can use `aspectlib.weave()` on: classes, instances, builtin functions, module level functions, methods, classmethods, staticmethods, instance methods etc.

CHAPTER 2

Installation

```
pip install aspectlib
```

Or, if you live in the stone age:

```
easy_install aspectlib
```

For your convenience there is a python-aspectlib meta-package that will just install aspectlib, in case you run `pip install python-aspectlib` by mistake.

2.1 Requirements

OS Any

Runtime Python 2.6, 2.7, 3.3, 3.4 or PyPy

Python 3.2, 3.1 and 3.0 are *NOT* supported (some objects are too crippled).

CHAPTER 3

Testing with `aspectlib.test`

3.1 Spy & mock toolkit: `record`/`mock` decorators

Lightweight spies and mock responses

Example usage, suppose you want to test this class:

```
>>> class ProductionClass(object):
...     def method(self):
...         return 'stuff'
>>> real = ProductionClass()
```

With `aspectlib.test.mock` and `aspectlib.test.record`:

```
>>> from aspectlib import weave, test
>>> patch = weave(real.method, [test.mock(3), test.record])
>>> real.method(3, 4, 5, key='value')
3
>>> assert real.method.calls == [(real, (3, 4, 5), {'key': 'value'})]
```

As a bonus, you have an easy way to rollback all the mess:

```
>>> patch.rollback()
>>> real.method()
'stuff'
```

With `mock`:

```
>>> from mock import Mock
>>> real = ProductionClass()
>>> real.method = Mock(return_value=3)
>>> real.method(3, 4, 5, key='value')
3
>>> real.method.assert_called_with(3, 4, 5, key='value')
```

3.2 Capture-replay toolkit: `Story` and `Replay`

Elaborate tools for testing difficult code

Writing tests using the `Story` is viable when neither integration tests or unit tests seem adequate:

- Integration tests are too difficult to integrate in your test harness due to automation issues, permissions or plain lack of performance.

- Unit tests are too difficult to write due to design issues (like too many dependencies, dependency injects is too hard etc) or take too much time to write to get good code coverage.

The `Story` is the middle-ground, bringing those two types of testing closer. It allows you to start with *integration tests* and later *mock/stub* with great ease all the dependencies.

Warning: The `Story` is not intended to patch and mock complex libraries that keep state around. E.g.: requests keeps a connection pool around - there are better choices.

Note: Using the `Story` on imperative, stateless interfaces is best.

3.2.1 An example: mocking out an external system

Suppose we implement this simple GNU `tree` clone:

```
>>> import os
>>> def tree(root, prefix=''):
...     if not prefix:
...         print("%s%s" % (prefix, os.path.basename(root)))
...     for pos, name in reversed(list(enumerate(sorted(os.listdir(root), reverse=True)))):
...         print("%s%s%s" % (prefix, "-- " if pos else "-- ", name))
...         absname = os.path.join(root, name)
...         if os.path.isdir(absname):
...             tree(absname, prefix + ("|   " if pos else "    "))
```

Lets suppose we would make up some directories and files for our tests:

```
>>> if not os.path.exists('some/test/dir'):   os.makedirs('some/test/dir')
>>> if not os.path.exists('some/test/empty'): os.makedirs('some/test/empty')
>>> with open('some/test/dir/file.txt', 'w') as fh:
...     pass
```

And we'll assert that `tree` has this output:

```
>>> tree('some')
some
-- test
    -- dir
    |   -- file.txt
    -- empty
```

But now we're left with some garbage and have to clean it up:

```
>>> import shutil
>>> shutil.rmtree('some')
```

This is not very practical - we'll need to create many scenarios, and some are not easy to create automatically (e.g: tests for permissions issues - not easy to change permissions from within a test).

Normally, to handle this we'd have have to manually monkey-patch the `os` module with various mocks or add dependency-injection in the `tree` function and inject mocks. Either approach we'll leave us with very ugly code.

With dependency-injection tree would look like this:

```
def tree(root, prefix='', basename=os.path.basename, listdir=os.listdir, join=os.path.join, isdir=os.
    ...
```

One could argue that this is overly explicit, and the function's design is damaged by testing concerns. What if we need to check for permissions ? We'd have to extend the signature. And what if we forget to do that ? In some situations one cannot afford all this (re-)engineering (e.g: legacy code, simplicity goals etc).

The `aspectlib.test.Story` is designed to solve this problem in a neat way.

We can start with some existing test data in the filesystem:

```
>>> os.makedirs('some/test/dir')
>>> os.makedirs('some/test/empty')
>>> with open('some/test/dir/file.txt', 'w') as fh:
...     pass
```

Write an empty story and examine the output:

```
>>> from aspectlib.test import Story
>>> with Story(os, methods="^(?!error)[a-z]+$") as story:
...     pass
>>> with story.replay(strict=False) as replay:
...     tree('some')
some
-- test
    -- dir
    |   -- file.txt
    -- empty
STORY/REPLAY DIFF:
    --- expected...
    +++ actual...
    @@ ... @@
    +os.listdir('some') == ['test']  # returns
    +os.stat('some/test') == os.stat_result((...))  # returns
    +os.listdir('some/test') == [...'dir'...]  # returns
    +os.stat('some/test/dir') == os.stat_result((...))  # returns
    +os.listdir('some/test/dir') == ['file.txt']  # returns
    +os.stat('some/test/dir/file.txt') == os.stat_result((...))  # returns
    +os.stat('some/test/empty') == os.stat_result((...))  # returns
    +os.listdir('some/test/empty') == []  # returns
ACTUAL:
    os.listdir('some') == ['test']  # returns
    os.stat('some/test') == os.stat_result((...))  # returns
    os.listdir('some/test') == [...'dir'...]  # returns
    os.stat('some/test/dir') == os.stat_result((...))  # returns
    os.listdir('some/test/dir') == ['file.txt']  # returns
    os.stat('some/test/dir/file.txt') == os.stat_result((...))  # returns
    os.stat('some/test/empty') == os.stat_result((...))  # returns
    os.listdir('some/test/empty') == []  # returns
```

We can quickly get whatever we would need to put in the story with `aspectlib.test.Replay.unexpected`:

```
>>> print(replay.unexpected)
os.listdir('some') == ['test']  # returns
os.stat('some/test') == os.stat_result((...))  # returns
os.listdir('some/test') == [...'dir'...]  # returns
os.stat('some/test/dir') == os.stat_result((...))  # returns
os.listdir('some/test/dir') == ['file.txt']  # returns
os.stat('some/test/dir/file.txt') == os.stat_result((...))  # returns
os.stat('some/test/empty') == os.stat_result((...))  # returns
os.listdir('some/test/empty') == []  # returns
```

Now we can remove the test directories and fill the story:

```
>>> import shutil
>>> shutil.rmtree('some')
```

The story:

```
>>> with Story(os, methods="^(?!error)[a-z]+$") as story:
...     os.listdir('some') == ['test']
...     os.stat('some/test') == os.stat_result((16893, 6691875, 2049, 3, 1000, 1000, 4096, 13991
...     os.listdir('some/test') == ['empty', 'dir']  # returns
...     os.stat('some/test/dir') == os.stat_result((16893, 6691876, 2049, 2, 1000, 1000, 4096, 1
...     os.listdir('some/test/dir') == ['file.txt']
...     os.stat('some/test/dir/file.txt') == os.stat_result((33204, 6691877, 2049, 1, 1000, 1000
...     os.stat('some/test/empty') == os.stat_result((16893, 6691877, 2049, 2, 1000, 1000, 4096,
...     os.listdir('some/test/empty') == []  # returns
```

And the *strict* `replay`:

```
>>> with story.replay(proxy=False) as replay:
...     tree('some')
some
-- test
    -- dir
    |   -- file.txt
    -- empty
```

If we diverge a bit from the story (or we'd have some unexpected change in the `tree` function) we'd get something like this:

```
>>> with Story(os, methods="^(?!error)[a-z]+$") as story:
...     os.listdir('some') == ['test']
...     os.listdir('bogus') == ['some bogus directory']
...     os.stat('some/test') == os.stat_result((16893, 6691875, 2049, 3, 1000, 1000, 4096, 139913153
...     os.listdir('some/test') == ['empty', 'dir']  # returns
...     os.stat('some/test/dir') == os.stat_result((16893, 6691876, 2049, 2, 1000, 1000, 4096, 13991
...     os.listdir('some/test/dir') == ['file.txt']
...     os.stat('some/test/dir/file.txt') == os.stat_result((33204, 6691877, 2049, 1, 1000, 1000, 9,
...     os.stat('some/test/empty') == os.stat_result((16893, 6691877, 2049, 2, 1000, 1000, 4096, 139
...     os.listdir('some/test/empty') == []  # returns
>>> with story.replay(proxy=False) as replay:
...     tree('some')
Traceback (most recent call last):
...
AssertionError: --- expected...
+++ actual...
@@ ... @@
 os.listdir('some') == ['test']  # returns
-os.listdir('bogus') == ['some bogus directory']  # returns
 os.stat('some/test') == os.stat_result((16893, 6691875, 2049, 3, 1000, 1000, 4096, 1399131539, 13991
 os.listdir('some/test') == [...'dir'...]  # returns
 os.stat('some/test/dir') == os.stat_result((16893, 6691876, 2049, 2, 1000, 1000, 4096, 1399131539, 1
```

CHAPTER 4

Rationale

There are perfectly sane use cases for monkey-patching (aka *weaving*):

- Instrumenting existing code for debugging, profiling and other measurements.
- Testing less flexible code. In some situations it's infeasible to use dependency injection to make your code more testable.

Then in those situations:

- You would need to handle yourself all different kids of patching (patching a module is different than patching a class, a function or a method for that matter). `aspectlib` will handle all this gross patching mumbo-jumbo for you, consistently, over many Python versions.
- Writing the actual wrappers is repetitive, boring and error-prone. You can't reuse wrappers but *you can reuse function decorators*.

CHAPTER 5

Frequently asked questions

5.1 Why is it called weave and not patch ?

Because it does more things that just patching. Depending on the *target* object it will patch and/or create one or more subclasses and objects.

5.2 Why doesn't aspectlib implement AOP like in framework X and Y ?

Some frameworks don't resort to monkey patching but instead force the user to use ridiculous amounts of abstractions and wrapping in order to make weaving possible. Notable example: spring-python.

For all intents and purposes I think it's wrong to have such high amount of boilerplate in Python.

Also, `aspectlib` is targeting a different stage of development: the maintenance stage - where the code is already written and needs additional behavior, in a hurry :)

Where code is written from scratch and AOP is desired there are better choices than both `aspectlib` and `spring-python`.

5.3 Why was aspectlib written ?

`aspectlib` was initially written because I was tired of littering other people's code with prints and logging statements just to fix one bug or understand how something works. `aspectlib.debug.log` is `aspectlib`'s *crown jewel*. Of course, `aspectlib` has other applications, see the *Rationale*.

CHAPTER 6

Examples

6.1 Retry decorator

TODO: Make a more configurable retry decorator and add it in `aspectlib.contrib`.

```
class Client(object):
    def __init__(self, address):
        self.address = address
        self.connect()
    def connect(self):
        # establish connection
    def action(self, data):
        # do some stuff

def retry(retries=(1, 5, 15, 30, 60), retry_on=(IOError, OSError), prepare=None):
    assert len(retries)

    @aspectlib.Aspect
    def retry_aspect(*args, **kwargs):
        durations = retries
        while True:
            try:
                yield aspectlib.Proceed
                break
            except retry_on as exc:
                if durations:
                    logging.warn(exc)
                    time.sleep(durations[0])
                    durations = durations[1:]
                    if prepare:
                        prepare(*args, **kwargs)
                else:
                    raise

    return retry_aspect
```

Now patch the `Client` class to have the retry functionality on all its methods:

```
aspectlib.weave(Client, retry())
```

or with different retry options (reconnect before retry):

```
aspectlib.weave(Client, retry(prepare=lambda self, *_: self.connect())
```

or just for one method:

```
aspectlib.weave(Client.action, retry())
```

You can see here the advantage of having reusable retry functionality. Also, the retry handling is decoupled from the `Client` class.

6.2 Debugging

... those damn sockets:

```
>>> import aspectlib, socket, sys
>>> with aspectlib.weave(
...     socket.socket,
...     aspectlib.debug.log(
...         print_to=sys.stdout,
...         stacktrace=None,
...     ),
...     lazy=True,
... ):
...     s = socket.socket()
...     s.connect(('example.com', 80))
...     s.send(b'GET / HTTP/1.1\r\nHost: example.com\r\n\r\n')
...     s.recv(8)
...     s.close()
...
{socket...}.connect(('example.com', 80))
{socket...}.connect => None
{socket...}.send(...'GET / HTTP/1.1\r\nHost: example.com\r\n\r\n')
{socket...}.send => 37
37
{socket...}.recv(8)
{socket...}.recv => ...HTTP/1.1...
...'HTTP/1.1'
...
```

The output looks a bit funky because it is written to be run by doctest - so you don't use broken examples :)

6.3 Testing

Mock behavior for tests:

```
class MyTestCase(unittest.TestCase):

    def test_stuff(self):

        @aspectlib.Aspect
        def mock_stuff(self, value):
            if value == 'special':
                yield aspectlib.Return('mocked-result')
            else:
                yield aspectlib.Proceed
```

```
    with aspectlib.weave(foo.Bar.stuff, mock_stuff):
        obj = foo.Bar()
        self.assertEqual(obj.stuff('special'), 'mocked-result')
```

CHAPTER 7

Reference

7.1 aspectlib

Safe toolkit for writing decorators (hereby called **aspects**)

`aspectlib.Aspect`	Container for the advice yielding generator.
`aspectlib.Proceed`	Instruction for calling the decorated function.
`aspectlib.Return`	Instruction for returning a *optional* value.

Power tools for patching functions (hereby glorified as **weaving**)

`aspectlib.ALL_METHODS`	Compiled regular expression objects
`aspectlib.NORMAL_METHODS`	Compiled regular expression objects
`aspectlib.weave`	Send a message to a recipient
`aspectlib.Rollback`	When called, rollbacks all the patches and changes the `weave()` has done.

7.1.1 Reference

class `aspectlib.`**`Proceed`**(**args*, ***kwargs*)
Instruction for calling the decorated function. Can be used multiple times.

If not used as an instance then the default args and kwargs are used.

class `aspectlib.`**`Return`**(*value*)
Instruction for returning a *optional* value.

If not used as an instance then `None` is returned.

class `aspectlib.`**`Aspect`**(*advising_function*, *bind=False*)
Container for the advice yielding generator. Can be used as a decorator on other function to change behavior according to the advices yielded from the generator.

Parameters

- **advising_function** (*generator function*) – A generator function that yields *Advices*.
- **bind** (*bool*) – A convenience flag so you can access the cutpoint function (you'll get it as an argument).

Usage:

```
>>> @Aspect
... def my_decorator(*args, **kwargs):
...     print("Got called with args: %s kwargs: %s" % (args, kwargs))
...     result = yield
...     print(" ... and the result is: %s" % (result,))
>>> @my_decorator
... def foo(a, b, c=1):
...     print((a, b, c))
>>> foo(1, 2, c=3)
Got called with args: (1, 2) kwargs: {'c': 3}
(1, 2, 3)
 ... and the result is: None
```

Normally you don't have access to the cutpoints (the functions you're going to use the aspect/decorator on) because you don't and should not call them directly. There are situations where you'd want to get the name or other data from the function. This is where you use the `bind=True` option:

```
>>> @Aspect(bind=True)
... def my_decorator(cutpoint, *args, **kwargs):
...     print("`%s` got called with args: %s kwargs: %s" % (cutpoint.__name__, args, kwargs))
...     result = yield
...     print(" ... and the result is: %s" % (result,))
>>> @my_decorator
... def foo(a, b, c=1):
...     print((a, b, c))
>>> foo(1, 2, c=3)
`foo` got called with args: (1, 2) kwargs: {'c': 3}
(1, 2, 3)
 ... and the result is: None
```

You can use these *advices*:

- `Proceed` or `None` - Calls the wrapped function with the default arguments. The *yield* returns the function's return value or raises an exception. Can be used multiple times (will call the function multiple times).
- `Proceed (*args, **kwargs)` - Same as above but with different arguments.
- `Return` - Makes the wrapper return `None` instead. If `aspectlib.Proceed` was never used then the wrapped function is not called. After this the generator is closed.
- `Return (value)` - Same as above but returns the given `value` instead of `None`.
- `raise exception` - Makes the wrapper raise an exception.

Note: The Aspect will correctly handle generators and coroutines (consume them, capture result).

Example:

```
>>> from aspectlib import Aspect
>>> @Aspect
... def log_errors(*args, **kwargs):
...     try:
...         yield
...     except Exception as exc:
...         print("Raised %r for %s/%s" % (exc, args, kwargs))
...         raise
```

Will work as expected with generators (and coroutines):

```
>>> @log_errors
... def broken_generator():
...     yield 1
...     raise RuntimeError()
>>> from pytest import raises
>>> raises(RuntimeError, lambda: list(broken_generator()))
Raised RuntimeError() for ()/{}
...

>>> @log_errors
... def broken_function():
...     raise RuntimeError()
>>> raises(RuntimeError, broken_function)
Raised RuntimeError() for ()/{}
...
```

And it will handle results:

```
>>> from aspectlib import Aspect
>>> @Aspect
... def log_results(*args, **kwargs):
...     try:
...         value = yield
...     except Exception as exc:
...         print("Raised %r for %s/%s" % (exc, args, kwargs))
...         raise
...     else:
...         print("Returned %r for %s/%s" % (value, args, kwargs))

>>> @log_results
... def weird_function():
...     yield 1
...     raise StopIteration('foobar')  # in Python 3 it's the same as: return 'foobar'
>>> list(weird_function())
Returned 'foobar' for ()/{}
[1]
```

---

class aspectlib.**Rollback**(*rollback=None*)

When called, rollbacks all the patches and changes the weave() has done.

__enter__()

Returns self.

__exit__(*_)

Performs the rollback.

rollback(*_)

Alias of __exit__.

__call__(*_)

Alias of __exit__.

aspectlib.**ALL_METHODS** **Weave all magic methods. Can be used as the value for methods argument in weave.**

Compiled regular expression objects

aspectlib.**NORMAL_METHODS** **Only weave non-magic methods. Can be used as the value for methods argument in weave.**

Compiled regular expression objects

aspectlib.**weave**(*target*, *aspect*[, *subclasses=True*, *methods=NORMAL_METHODS*, *lazy=False*, *aliases=True*])

Send a message to a recipient

Parameters

- **target** (aspectlib.Aspect, function decorator or list of) – The object to weave.
- **aspects** – The aspects to apply to the object.
- **subclasses** (*bool*) – If True, subclasses of target are weaved. *Only available for classes*
- **aliases** (*bool*) – If True, aliases of target are replaced.
- **lazy** (*bool*) – If True only target's __init__ method is patched, the rest of the methods are patched after __init__ is called. *Only available for classes*.
- **methods** (*list or regex or string*) – Methods from target to patch. *Only available for classes*

Returns aspectlib.Rollback instance

Raises TypeError If target is a unacceptable object, or the specified options are not available for that type of object.

Changed in version 0.4.0: Replaced *only_methods*, *skip_methods*, *skip_magicmethods* options with *methods*. Renamed *on_init* option to *lazy*. Added *aliases* option. Replaced *skip_subclasses* option with *subclasses*.

7.2 aspectlib.debug

aspectlib.debug.log	Decorates *func* to have logging.
aspectlib.debug.format_stack	Returns a one-line string with the current callstack.
aspectlib.debug.frame_iterator	Yields frames till there are no more.
aspectlib.debug.strip_non_ascii	Convert to string (using *str*) and replace non-ascii characters with a dot (.).

aspectlib.debug.**format_stack**(*skip=0*, *length=6*, *_sep='/'*)
Returns a one-line string with the current callstack.

aspectlib.debug.**frame_iterator**(*frame*)
Yields frames till there are no more.

aspectlib.debug.**log**(*func=None*, *stacktrace=10*, *stacktrace_align=60*, *attributes=()*, *module=True*, *call=True*, *call_args=True*, *call_args_repr=<built-in function repr>*, *result=True*, *exception=True*, *exception_repr=<built-in function repr>*, *result_repr=<function strip_non_ascii at 0x7fd10050caa0>*, *use_logging='CRITICAL'*, *print_to=None*)
Decorates *func* to have logging.

Parameters

- **func** (*function*) – Function to decorate. If missing log returns a partial which you can use as a decorator.
- **stacktrace** (*int*) – Number of frames to show.
- **stacktrace_align** (*int*) – Column to align the framelist to.
- **attributes** (*list*) – List of instance attributes to show, in case the function is a instance method.
- **module** (*bool*) – Show the module.
- **call** (*bool*) – If True, then show calls. If False only show the call details on exceptions (if exception is enabled) (default: True)

- **call_args** (*bool*) – If `True`, then show call arguments. (default: `True`)
- **call_args_repr** (*bool*) – Function to convert one argument to a string. (default: `repr`)
- **result** (*bool*) – If `True`, then show result. (default: `True`)
- **exception** (*bool*) – If `True`, then show exceptions. (default: `True`)
- **exception_repr** (*function*) – Function to convert an exception to a string. (default: `repr`)
- **result_repr** (*function*) – Function to convert the result object to a string. (default: `strip_non_ascii` - like `str` but nonascii characters are replaced with dots.)
- **use_logging** (*string*) – Emit log messages with the given loglevel. (default: `"CRITICAL"`)
- **print_to** (*fileobject*) – File object to write to, in case you don't want to use logging module. (default: `None` - printing is disabled)

Returns A decorator or a wrapper.

Example

```
>>> @log(print_to=sys.stdout)
... def a(weird=False):
...     if weird:
...         raise RuntimeError('BOOM!')
>>> a()
a()                                                    <<< ...
a => None
>>> try:
...     a(weird=True)
... except Exception:
...     pass # naughty code !
a(weird=True)                                          <<< ...
a ~ raised RuntimeError('BOOM!',)
```

You can conveniently use this to logs just errors, or just results, example:

```
>>> import aspectlib
>>> with aspectlib.weave(float, log(call=False, result=False, print_to=sys.stdout)):
...     try:
...         float('invalid')
...     except Exception as e:
...         pass # naughty code !
float('invalid')                                       <<< ...
float ~ raised ValueError(...float...invalid...)
```

This makes debugging naughty code easier.

PS: Without the weaving it looks like this:

```
>>> try:
...     log(call=False, result=False, print_to=sys.stdout)(float)('invalid')
... except Exception:
...     pass # naughty code !
float('invalid')                                       <<< ...
float ~ raised ValueError(...float...invalid...)
```

Changed in version 0.5.0: Renamed *arguments* to *call_args*. Renamed *arguments_repr* to *call_args_repr*. Added *call* option.

aspectlib.debug.**strip_non_ascii**(*val*)
Convert to string (using *str*) and replace non-ascii characters with a dot (`.`).

7.3 aspectlib.test

This module aims to be a lightweight and flexible alternative to the popular mock framework and more.

aspectlib.test.record	Factory or decorator (depending if *func* is initially given).
aspectlib.test.mock	Factory for a decorator that makes the function return a given *return_value*.
aspectlib.test.Story	This a simple yet flexible tool that can do "capture-replay mocking" or "test doubles" [1].
aspectlib.test.Replay	Object implementing the *replay transaction*.

aspectlib.test.**record**(*func=None*, *recurse_lock_factory=<function allocate_lock at 0x7fd10040de60>*, ***options*)
Factory or decorator (depending if *func* is initially given).

Parameters

- **callback** (*list*) – An a callable that is to be called with `instance, function, args, kwargs`.
- **calls** (*list*) – An object where the *Call* objects are appended. If not given and `callback` is not specified then a new list object will be created.
- **iscalled** (*bool*) – If `True` the *func* will be called. (default: `False`)
- **extended** (*bool*) – If `True` the *func*'s `__name__` will also be included in the call list. (default: `False`)
- **results** (*bool*) – If `True` the results (and exceptions) will also be included in the call list. (default: `False`)

Returns A wrapper that has a *calls* property.

The decorator returns a wrapper that records all calls made to *func*. The history is available as a `call` property. If access to the function is too hard then you need to specify the history manually.

Example

```
>>> @record
... def a(x, y, a, b):
...     pass
>>> a(1, 2, 3, b='c')
>>> a.calls
[Call(self=None, args=(1, 2, 3), kwargs={'b': 'c'})]
```

Or, with your own history list:

```
>>> calls = []
>>> @record(calls=calls)
... def a(x, y, a, b):
...     pass
>>> a(1, 2, 3, b='c')
>>> a.calls
[Call(self=None, args=(1, 2, 3), kwargs={'b': 'c'})]
```

[1] http://www.martinfowler.com/bliki/TestDouble.html

```
>>> calls is a.calls
True
```

Changed in version 0.9.0: Renamed *history* option to *calls*. Renamed *call* option to *iscalled*. Added *callback* option. Added *extended* option.

`aspectlib.test.`**`mock`**(*return_value*, *call=False*)

Factory for a decorator that makes the function return a given *return_value*.

Parameters

- **return_value** – Value to return from the wrapper.
- **call** (*bool*) – If `True`, call the decorated function. (default: `False`)

Returns A decorator.

class `aspectlib.test.`**`Story`**(**args*, ***kwargs*)

This a simple yet flexible tool that can do "capture-replay mocking" or "test doubles" [1]. It leverages `aspectlib`'s powerful `weaver`.

Parameters

- **target** (Same as for `aspectlib.weave`.) – Targets to weave in the *story*/*replay* transactions.
- **subclasses** (*bool*) – If `True`, subclasses of target are weaved. *Only available for classes*
- **aliases** (*bool*) – If `True`, aliases of target are replaced.
- **lazy** (*bool*) – If `True` only target's `__init__` method is patched, the rest of the methods are patched after `__init__` is called. *Only available for classes*.
- **methods** (*list or regex or string*) – Methods from target to patch. *Only available for classes*

The `Story` allows some testing patterns that are hard to do with other tools:

•**Proxied mocks**: partially mock *objects* and *modules* so they are called normally if the request is unknown.

•**Stubs**: completely mock *objects* and *modules*. Raise errors if the request is unknown.

The `Story` works in two of transactions:

•**The story**: You describe what calls you want to mocked. Initially you don't need to write this. Example:

```
>>> import mymod
>>> with Story(mymod) as story:
...     mymod.func('some arg') == 'some result'
...     mymod.func('bad arg') ** ValueError("can't use this")
```

•**The replay**: You run the code uses the interfaces mocked in the *story*. The `replay` always starts from a *story* instance.

Changed in version 0.9.0: Added in.

`replay`(***options*)

Parameters

- **proxy** (*bool*) – If `True` then unexpected uses are allowed (will use the real functions) but they are collected for later use. Default: `True`.
- **strict** (*bool*) – If `True` then an `AssertionError` is raised when there were *unexpected calls* or there were *missing calls* (specified in the story but not called). Default: `True`.

- **dump** (*bool*) – If `True` then the *unexpected/missing calls* will be printed (to `sys.stdout`). Default: `True`.

Returns A `aspectlib.test.Replay` object.

Example

```
>>> import mymod
>>> with Story(mymod) as story:
...     mymod.func('some arg') == 'some result'
...     mymod.func('other arg') == 'other result'
>>> with story.replay(strict=False):
...     print(mymod.func('some arg'))
...     mymod.func('bogus arg')
some result
Got bogus arg in the real code !
STORY/REPLAY DIFF:
    --- expected...
    +++ actual...
    @@ -1,2 +1,2 @@
     mymod.func('some arg') == 'some result'  # returns
    -mymod.func('other arg') == 'other result'  # returns
    +mymod.func('bogus arg') == None  # returns
ACTUAL:
    mymod.func('some arg') == 'some result'  # returns
    mymod.func('bogus arg') == None  # returns
```

class `aspectlib.test.`**`Replay`**`(?)`

Object implementing the *replay transaction*.

This object should be created by `Story`'s `replay` method.

`diff`

Returns a pretty text representation of the unexpected and missing calls.

Most of the time you don't need to directly use this. This is useful when you run the *replay* in `strict=False` mode and want to do custom assertions.

`missing`

Returns a pretty text representation of just the missing calls.

`unexpected`

Returns a pretty text representation of just the unexpected calls.

The output should be usable directly in the story (just copy-paste it). Example:

```
>>> import mymod
>>> with Story(mymod) as story:
...     pass
>>> with story.replay(strict=False, dump=False) as replay:
...     mymod.func('some arg')
...     try:
...         mymod.badfunc()
...     except ValueError as exc:
...         print(exc)
Got some arg in the real code !
boom!
>>> print(replay.unexpected)
```

```
mymod.func('some arg') == None  # returns
mymod.badfunc() ** ValueError('boom!',)  # raises
```

We can just take the output and paste in the story:

```
>>> import mymod
>>> with Story(mymod) as story:
...     mymod.func('some arg') == None  # returns
...     mymod.badfunc() ** ValueError('boom!')  # raises
>>> with story.replay():
...     mymod.func('some arg')
...     try:
...         mymod.badfunc()
...     except ValueError as exc:
...         print(exc)
boom!
```

CHAPTER 8

Development

Development is happening on Github.

CHAPTER 9

TODO & Ideas

9.1 Validation

```
class BaseProcessor(object):
    def process_foo(self, data):
        # do some work

    def process_bar(self, data):
        # do some work

class ValidationConcern(aspectlib.Concern):
    @aspectlib.Aspect
    def process_foo(self, data):
        # validate data
        if is_valid_foo(data):
          yield aspectlib.Proceed
        else:
          raise ValidationError()

    @aspectlib.Aspect
    def process_bar(self, data):
        # validate data
        if is_valid_bar(data):
          yield aspectlib.Proceed
        else:
          raise ValidationError()

aspectlib.weave(BaseProcesor, ValidationConcern)

class MyProcessor(BaseProcessor):
    def process_foo(self, data):
        # do some work

    def process_bar(self, data):
        # do some work

# MyProcessor automatically inherits BaseProcesor's ValidationConcern
```

CHAPTER 10

Changelog

10.1 Version 1.1.1

- Use `ASPECTLIB_DEBUG` for every logger in `aspectlib`.

10.2 Version 1.1.0

- Add a *bind* option to `aspectlib.Aspect` so you can access the cutpoint from the advisor.
- Replaced automatic importing in `aspectlib.test.Replay` with extraction of context variables (locals and globals from the calling `aspectlib.test.Story`). Works better than the previous inference of module from AST of the result.
- All the methods on the replay are now properties: `aspectlib.test.Story.diff`, `aspectlib.test.Story.unexpected` and `aspectlib.test.Story.missing`.
- Added `aspectlib.test.Story.actual` and `aspectlib.test.Story.expected`.
- Added an `ASPECTLIB_DEBUG` environment variable option to switch on debug logging in `aspectlib`'s internals.

10.3 Version 1.0.0

- Reworked the internals `aspectlib.test.Story` to keep call ordering, to allow dependencies and improved the serialization (used in the diffs and the missing/unexpected lists).

10.4 Version 0.9.0

- Changed `aspectlib.test.record`:
  - Renamed *history* option to *calls*.
  - Renamed *call* option to *iscalled*.
  - Added *callback* option.
  - Added *extended* option.
- Changed `aspectlib.weave`:

  - Allow weaving everything in a module.
  - Allow weaving instances of new-style classes.
- Added `aspectlib.test.Story` class for capture-replay and stub/mock testing.

10.5 Version 0.8.1

- Use simpler import for the py3support.

10.6 Version 0.8.0

- Change `aspectlib.debug.log` to use `Aspect` and work as expected with coroutines or generators.
- Fixed `aspectlib.debug.log` to work on Python 3.4.
- Remove the undocumented `aspectlib.Yield` advice. It was only usable when decorating generators.

10.7 Version 0.7.0

- Add support for decorating generators and coroutines in `Aspect`.
- Made aspectlib raise better exceptions.

10.8 Version 0.6.1

- Fix checks inside `aspectlib.debug.log` that would inadvertently call `__bool__`/`__nonzero`.

10.9 Version 0.6.0

- Don't include __getattribute__ in ALL_METHODS - it's too dangerous dangerous dangerous dangerous dangerous dangerous ... ;)
- Do a more reliable check for old-style classes in debug.log
- When weaving a class don't weave attributes that are callable but are not actually routines (functions, methods etc)

10.10 Version 0.5.0

- Changed `aspectlib.debug.log`:
  - Renamed *arguments* to *call_args*.
  - Renamed *arguments_repr* to *call_args_repr*.
  - Added *call* option.
  - Fixed issue with logging from old-style methods (object name was a generic "instance").

- Fixed issues with weaving some types of builtin methods.
- Allow to apply multiple aspects at the same time.
- Validate string targets before weaving. `aspectlib.weave('mod.invalid name', aspect)` now gives a clear error (`invalid name` is not a valid identifier)
- Various documentation improvements and examples.

10.11 Version 0.4.1

- Remove junk from 0.4.0's source distribution.

10.12 Version 0.4.0

- Changed `aspectlib.weave`:
  - Replaced *only_methods*, *skip_methods*, *skip_magicmethods* options with *methods*.
  - Renamed *on_init* option to *lazy*.
  - Added *aliases* option.
  - Replaced *skip_subclasses* option with *subclasses*.
- Fixed weaving methods from a string target.

CHAPTER 11

Indices and tables

- *genindex*
- *modindex*
- *search*

a