python-aspectlib 0.4.1

Release 0.4.1

May 03, 2014

`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 `aspectlib.Aspect`. The generator yields *advices* - simple behavior changing instructions.

The *aspect* is 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()
```

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

CHAPTER 3

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 4

Frequently asked questions

4.1 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.

4.2 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`.

4.3 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 5

Examples

5.1 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.

5.2 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(('google.com', 80))
...     s.send(b'GET / HTTP/1.0\r\n\r\n')
...     s.recv(8)
...     s.close()
...
{socket...}.connect(('google.com', 80))
{socket...}.connect => None
{socket...}.send(...'GET / HTTP/1.0\r\n\r\n')
{socket...}.send => 18
18
{socket...}.recv(8)
{socket...}.recv => ...HTTP/1.0...
...'HTTP/1.0'
...
```

The output looks a bit funky because it is written to be run by doctest.

5.3 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 6

Reference

6.1 aspectlib

`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.
`aspectlib.Aspect`	Container for the advice yielding generator.
`aspectlib.Proceed`	Instructs the Aspect Calls to call the decorated function.
`aspectlib.Return`	Instructs the Aspect to return a value.

class `aspectlib.`**`Aspect`**(*advise_function*)
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.

class `aspectlib.`**`Proceed`**(**args*, ***kwargs*)
Instructs the Aspect Calls to call 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*)
Instructs the Aspect to return a value.

class `aspectlib.`**`Rollback`**(*rollback=None*)
When called, rollbacks all the patches and changes the `weave()` has done.

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` or function decorator) – The object to weave
- **aspect** – The aspect 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 patch target's `__init__`, 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*.

6.2 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*, *arguments=True*, *arguments_repr=<built-in function repr>*, *result=True*, *exception=True*, *exception_repr=<built-in function repr>*, *result_repr=<function strip_non_ascii at 0x3d87230>*, *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.
- **arguments** (*bool*) – If `True`, then show arguments.
- **arguments_repr** (*bool*) – Function to convert one argument to a string.
- **result** (*bool*) – If `True`, then show result.
- **exception** (*bool*) – If `True`, then show exceptions.
- **exception_repr** (*function*) – Function to convert an exception to a string.
- **result_repr** (*function*) – Function to convert the result object to a string.
- **use_logging** (*string*) – Emit log messages with the given loglevel.
- **print_to** (*fileobject*) – File object to write to, in case you don't want to use logging module.

Returns A decorator or a wrapper.

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

6.3 aspectlib.test

`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*.

class `aspectlib.test.`**`Call`**
Call(self, args, kwargs)

`args`
Alias for field number 1

`kwargs`
Alias for field number 2

`self`
Alias for field number 0

class `aspectlib.test.`**`RecordingWrapper`**(*wrapped*, *wrapper*, *calls*)
Function wrapper that has a *calls* attribute.

Parameters

- **wrapped** (*function*) – Function to be wrapped
- **wrapped** – Wrapper function
- **calls** (*list*) – Instance to put in the *.calls* attribute.

`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.

Returns A decorator.

aspectlib.test.**record**(*func=None*, *call=False*, *history=None*)
Factory or decorator (depending if *func* is initially given).

The decorator returns a wrapper that records all calls made to *func*.

Parameters

- **history** (*list*) – An object where the *Call* objects are appended. If not given a new list object will be created.
- **call** (*bool*) – If `True` the *func* will be called.

Returns A wrapper that has a *calls* property.

CHAPTER 7

Development

Development is happening on Github.

CHAPTER 8

TODO & Ideas

8.1 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 9

Changelog

9.1 Version 0.4.1

- Remove junk from 0.4.0's source distribution.

9.2 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 10

Indices and tables

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

a