aopy: A program transformation-based aspect oriented framework for Python

Martin Matusiak

May 18, 2009

Contents

1 Introduction 3

1.1 aopy . . . 3

2 Quick start 3

2.1 A barebones module . . . 3

2.2 Aspects . . . 4

2.2.1 A caching aspect . . . 4

2.2.2 A timing aspect . . . 5

2.2.3 A logging aspect . . . 5

2.3 The spec . . . 6

2.4 Compiling . . . 7

2.5 The transformed module . . . 7

2.6 Running the program . . . 8

3 Design 9

3.1 Problem description . . . 9

3.2 Aspects and join points . . . 11

3.3 Point cuts . . . 11

3.4 Programming interface . . . 11

3.5 The aopyc compiler . . . 12

4 Program transformation 12

4.1 Mode of operation . . . 12

4.1.1 Specification file format . . . 13

4.1.2 Limitations of static analysis . . . 13

4.2 Setting metaclass . . . 14

4.3 Setting properties . . . 14

4.3.1 Finding instance methods . . . 14

4.3.2 Finding instance attributes . . . 16

4.4 Adding decorators 16
4.4.1 Parameterized decorators 17
4.5 Resolving references 18
4.5.1 Import chasing 18

1 Introduction

Aspect oriented programming (AOP) is still something of a curiosity in the professional world, its merits perhaps still to be fully examined. Not even the canonical and most cited AOP framework, AspectJ[2], is widely used. Even so, there has been much experimentation with AOP techniques, also in Python. Wikipedia[3] has an extensive list of AOP frameworks, past and present.

The various Python based frameworks have used Python's dynamic features extensively to provide AOP features, but none have taken the route staked out by AspectJ, namely:

- to compile source files to instrumented bytecode, such that
- the source files are not touched, and
- the invocation of the program in the instrumented state is unchanged.

In other words, none of the existing frameworks offer an out-of-band instrumentation, one that can be turned on or off at will through recompilation, and is completely external to the source code. For more on this topic, see the companion paper "Strategies for aspect oriented programming in Python"[1].

1.1 aopy

aopy is an AOP framework in the AspectJ mold, that is an out-of-band instrumentation through program transformation. The high level strategy is as follows.

1. Parse the program into an abstract syntax tree (ast), then
2. transform the program with the aspects,
3. compile the program to bytecode.

The main aims of aopy are:

- to perform an out-of-band transformation,
- to use well known dynamic Python techniques to instrument code in well understood ways (that is, with the use of properties, decorators and metaclasses),
- to provide a static mutation (that is, instrumentation happens before the first client of the code can access it, so the results for all clients are uniform),
- to provide full reach (that is, all clients of the code have the same view of the module, regardless of module import sequence),

2 Quick start

2.1 A barebones module

The subject for this demonstration is a `NetworkIface` (network interface) class, such that you might find in a network manager type of application. For brevity we include only the bare essentials. The class has an instance attribute `ip`, representing the ip address on the interface. `ip` can be set statically by assigning to it directly, but it can also be set to an ip address obtained through a dhcp request, by calling the method `dhcp_request`.

```python
# <./main.py>
class NetworkIface(object):
    def __init__(self):
        self.ip = None

    def dhcp_request(self):
        self.ip = (10,0,0,131) # XXX magic goes here


if __name__ == '__main__':
    iface = NetworkIface()
    iface.ip = (10,0,0,1)
    iface.ip = (10,0,0,2)
    iface.dhcp_request()
```

With that the semantics of the class is complete. But suppose we are interested in additional diagnostic logic that does not directly concern the function of the class.

1. On dhcp managed networks clients are typically handed out arbitrary ip addresses. Suppose we want to keep track of all the ips that have been assigned on this interface, to understand how the dhcp service allocates ip addresses on a particular network. The cleanest and most Pythonic way to do this is to use a property on the `ip` attribute that has the side effect of caching the value being set.
2. Setting the ip via dhcp should ideally be instantaneous. But sometimes it seems to take a long time, so it would be useful to time the execution of the `dhcp_request` method. The Pythonic way is to use a function decorator here.
3. This module is in heavy development at the moment, and is unreliable on some systems. We would like to be able to log as much information as is necessary to understand the various cases that occur on our users' machines. One, admittedly heavy handed, logging philosophy is to log every single function call with arguments and return value, because one cannot predict just what details will be of value once an error occurs.

What all these concerns have in common is their temporal nature. We do not wish any of these features in our final product, but they are helpful in development, thus it would be well advised to keep this logic separate if possible.

2.2 Aspects

The aspect code is written quite separately from the module to be transformed.

2.2.1 A caching aspect

This module defines a pair of property functions, a `get` and a `set`. These will become instance methods in the class they are applied to. The `get` method is trivial, while the `set` method caches non null inputs of the property value. Both methods store the property value in an instance of the `Cache` helper class. The `cache` object may be imported by other code to access values of the property attribute.

```python
# <aspects/cache.py>
class Cache():
    def __init__(self):
        self.values = set()
        self.value = None
cache = Cache()

def get(self):
    return cache.value

def set(self, value):
    if value:
        print "c New value: %s" % str(value)
    if any(cache.values):
        prev = ", ".join([str(val) for val in cache.values])
        print "c  Previous values: %s" % prev
    if value:
        cache.values = cache.values.union([value])
    cache.value = value
```

2.2.2 A timing aspect

This module defines a decorator function which delegates the function call to the function it wraps, while timing the execution of the function.

```python
# <aspects/cache.py>
import functools
import time

def timed(func):
    @functools.wraps(func)    # preserve function name
    def new_func(*args, **kwargs):
        before = time.time()
        res = func(*args, **kwargs)
        after = time.time()
        print "t Timed execution: %s" % func.__name__
        print "t  Arguments: %s" % str((args, kwargs))
        print "t  Time: %s" % str(after - before)
        return res
    return new_func
```

2.2.3 A logging aspect

This module defines a metaclass for logging. When a class is being instantiated from it, every method in the new class is decorated with a special logging decorator (`functionlogger`). The decorator is defined subsequently. It reports every function entry and exit.

```python
# <aspects/logger.py>
import functools
import types

class LoggerMeta(type):
    def __new__(cls, name, bases, dct):
        for (key, value) in dct.items():
            if type(value) == types.FunctionType:
                dct[key] = functionlogger(value)
        return type.__new__(cls, name, bases, dct)

    def __init__(cls, name, bases, dct):
        super(LoggerMeta, cls).__init__(name, bases, dct)

def functionlogger(func):
    @functools.wraps(func)    # preserve function name
    def new_func(*args, **kwargs):
        # XXX rewrite to use syslog instead of stdout
        print "l Entering function: %s" % func.__name__
        print "l   with arguments: %s" % str((args, kwargs))
        res = func(*args, **kwargs)
        print "l Leaving function: %s" % func.__name__
        print "l   with return value: %s" % str(res)
        return res
    return new_func
```

2.3 The spec

The spec module imports the relevant aspect code and constructs Aspect objects that define how transformation will occur.

First there is a `caching_aspect`, on which we set a property using functions from the `cache` module.

Next we have a `timing_aspect`, which sets a decorator

Then we have a `logging_aspect`, which sets a metaclass.

```python
# <./spec.py>
import aopy

import aspects.cache
import aspects.logger
import aspects.timer

caching_aspect = aopy.Aspect()
caching_aspect.add_property('main:NetworkIface/ip',
    fget=aspects.cache.get, fset=aspects.cache.set)

timing_aspect = aopy.Aspect()
timing_aspect.add_decorator('main:NetworkIface/.*dhcp.*',
     aspects.timer.timed)

logging_aspect = aopy.Aspect()
logging_aspect.add_metaclass('main:NetworkIface',
     aspects.logger.LoggerMeta)

__all__ = ['caching_aspect', 'timing_aspect', 'logging_aspect']
```

2.4 Compiling

We have now assembled all the pieces we need to compile the aspects into the program:

1. Program code
2. Aspects
3. A specification

Note that all of these files are separate modules that can be executed and tested one at a time.

The time has come to compile the program, like so:

```
$ aopyc -t spec.py main.py
```

```
Transforming module /home/alex/uu/colloq/aopy/code/main.py
Pattern matched: main:NetworkIface on main:NetworkIface
Pattern matched: main:NetworkIface/ip on main:NetworkIface/ip
Pattern matched: main:NetworkIface/ip on main:NetworkIface/ip/ip
Pattern failed: main:NetworkIface/.*dhcp.* on main:NetworkIface/__init__
Pattern matched: main:NetworkIface/.*dhcp.* on main:NetworkIface/dhcp_request
```

The output from the compiler describes all the possible matches that were made between join points and pointcuts. That is, every function is matched against every decorator, every class against every metaclass and so forth. In this case, all point cuts were matched, except for the `timed` decorator, which was told only to match functions with "dhcp" in their name.

The compilation result is the bytecode module `main.pyc`. The original source module is left untouched.

2.5 The transformed module

Since the compiler produces a bytecode module, there is no actual source version of the transformed module. Nevertheless, the code is equivalent to the following.

First of all, the module has been abridged quite generously. Lines marked with a red plus + are new. Lines 2-5 are to ensure that the module can find the aspect code (in this case both share the same directory). Lines 7-9 import all the aspect modules used in this instrumentation.

Line 12 holds the first useful injection: `NetworkIface` has been given a metaclass. Line 19 shows that the function `dhcp_request` has been decorated with timed execution. Line 23 sets a property for the attribute `ip`.

But what of the purple hashes #? Both methods of `NetworkIface` have been decorated with a `functionlogger`, but this injection is not found in the specification. No, this is the work of `LoggerMeta` – injecting decorators programmatically upon class creation.

```
# <./main.py> transformed
import sys+
for path in ('.'):+
    if path not in sys.path:+
        sys.path.append(path)+

import aspects.cache as cache+
import aspects.logger as logger+
import aspects.timer as timer+

class NetworkIface(object):
    __metaclass__ = logger.LoggerMeta+

    @logger.functionlogger#
    def __init__(self):
        self.ip = None

    @logger.functionlogger#
    @timer.timed+
    def dhcp_request(self):
        self.ip = (10,0,0,131)  # XXX magic goes here

    ip = property(fget=cache.get, fset=cache.set)+


if __name__ == '__main__':
    iface = NetworkIface()
    iface.ip = (10,0,0,1)
    iface.ip = (10,0,0,2)
    iface.dhcp_request()
```

2.6 Running the program

The original program code did not produce any output, but the instrumented version certainly is expected to. We run the program by executing the bytecode module.

```
$ python main.pyc
```

```
l Entering function: __init__
l   with arguments: ((<__main__.NetworkIface object at 0xb7d1ec6c>,), {})
l Leaving function: __init__
l   with return value: None
c New value: (10, 0, 0, 1)
c New value: (10, 0, 0, 2)
c  Previous values: (10, 0, 0, 1)
l Entering function: dhcp_request
l   with arguments: ((<__main__.NetworkIface object at 0xb7d1ec6c>,), {})
c New value: (10, 0, 0, 131)
c  Previous values: (10, 0, 0, 1), (10, 0, 0, 2)
t Timed execution: dhcp_request
t  Arguments: ((<__main__.NetworkIface object at 0xb7d1ec6c>,), {})
t  Time: 5.69820404053e-05
l Leaving function: dhcp_request
l   with return value: None
```

The program produces three kinds of output.

- Lines beginning with l are produced by the logging aspect.

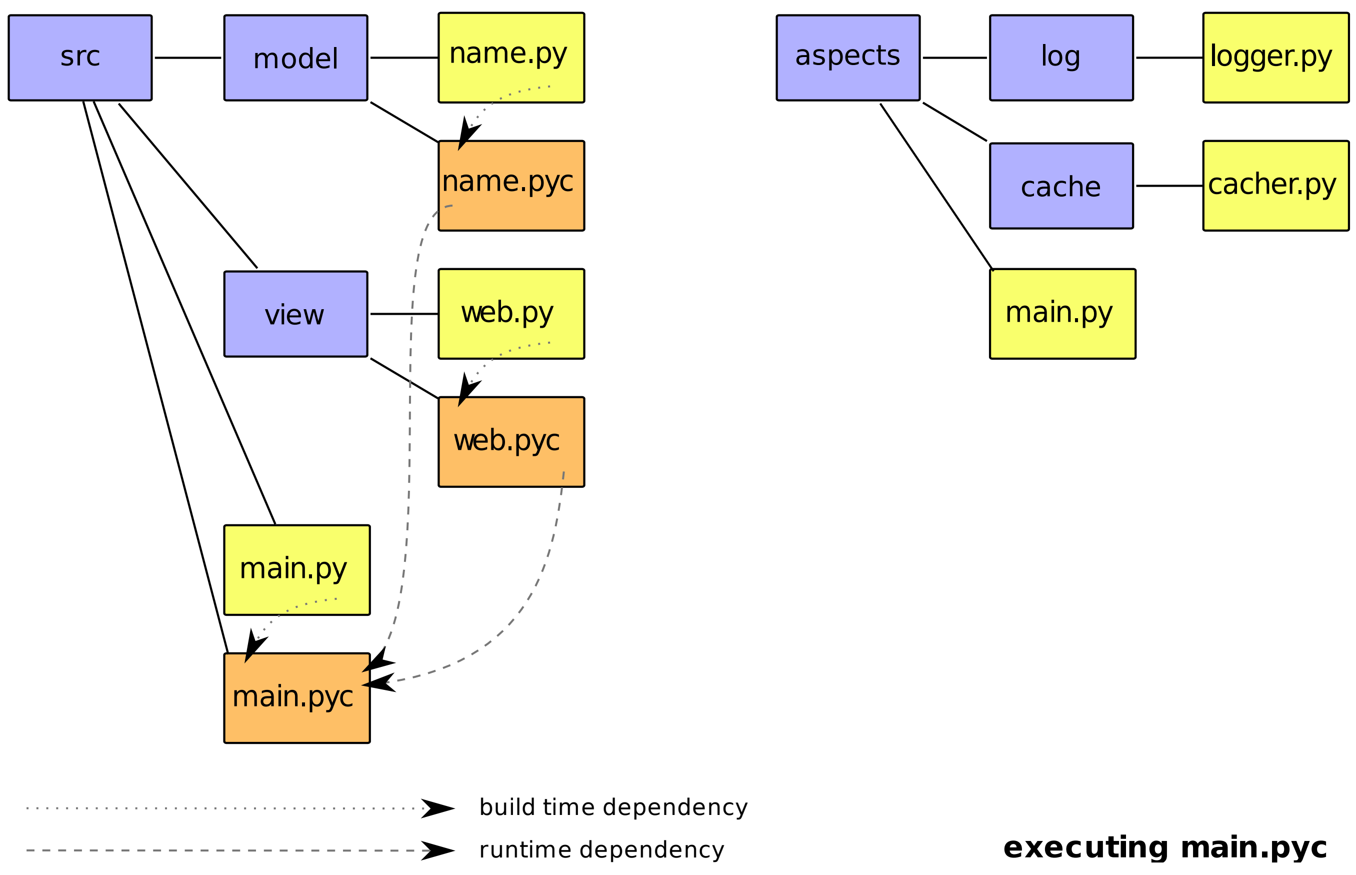

Figure 1: Starting point

- Lines beginning with `c` are produced by the caching aspect.
- Lines beginning with `t` are produced by the timing aspect.

The first thing to happen is the instantiation of the `NetworkIface` class. This triggers the `__init__` function to run, duly recorded by our logger. Next, we assign values to `iface.ip`, which is where the cache steps in.

Then we execute `dhcp_request`, which is recorded by the logger. Inside the function we assign a new value to `self.ip`, which again the cache records. And since the function is timed, the timer reports the running time. Finally, the logger records the exit of the function.

3 Design

3.1 Problem description

The aim of aopy is to perform and out-of-band program transformation on source files, producing bytecode. The problem is sketched in the figure 1. From the outset, we have:

1. a directory structure containing the program source code (`src`),
    - where the functionality of the modules deeper in the tree is used by modules nearer the root,
    - where the program can be executed as is by running `main.py`, which (along with all imported modules) will be compiled to bytecode (`.pyc`) by the Python interpreter.

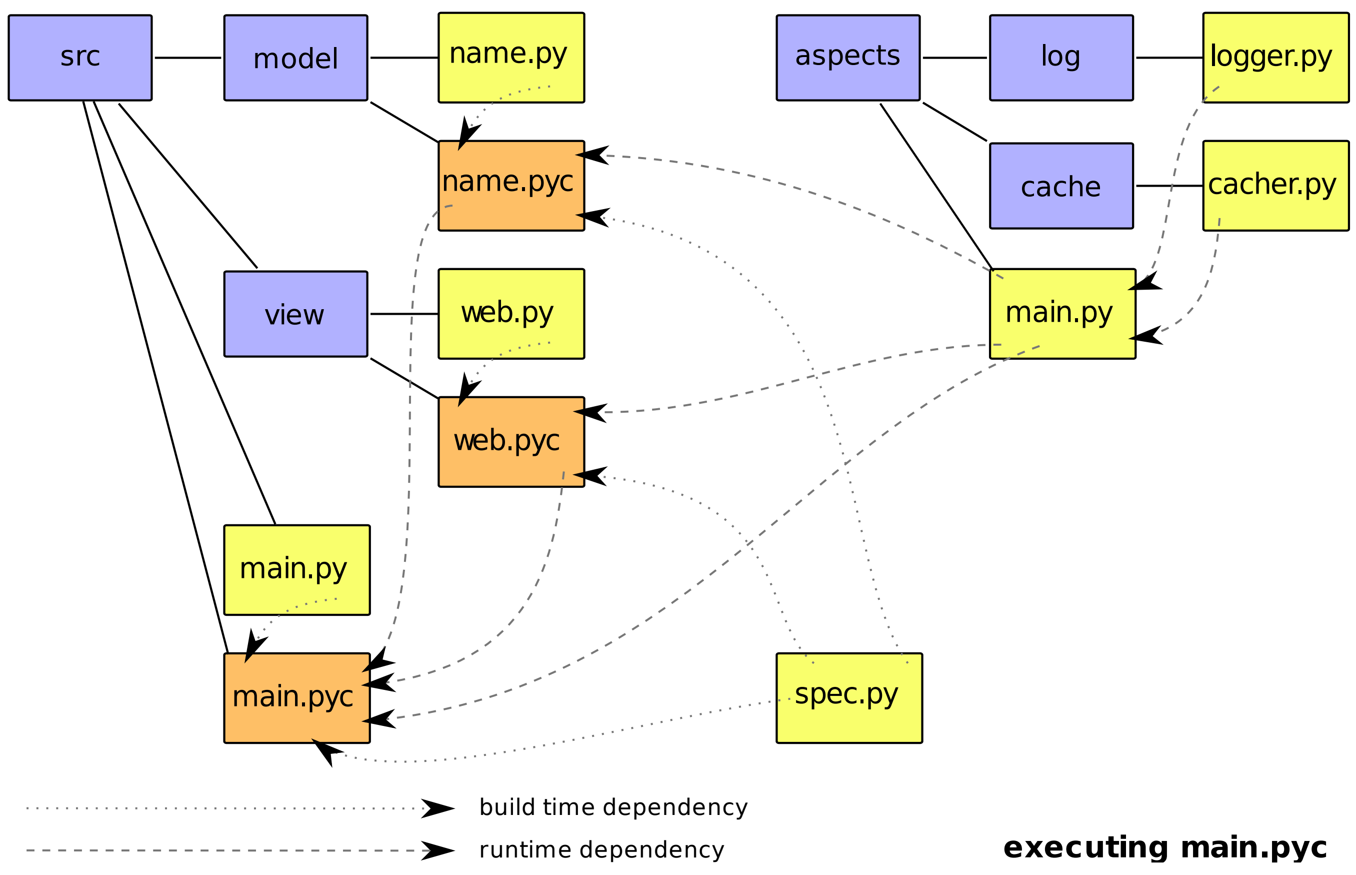

Figure 2: Solution sketch

2. a directory structure of diagnostic code we would like to use to instrument the program (`aspects`),
   - where the modules nearer the root import modules deeper in the tree[1],
   - and we would like to keep this code reusable and avoid binding it to the program in any way.

The two directories are not necessarily co-located, so any kind of instrumentation would have to ensure that the program can find the aspect code.

Clearly, once the program has been instrumented, all it needs to run is the aspect code. But in order to bind the two code trees together, so as to compile the program into bytecode, we need an intermediary. For this we devise a specification file, see figure 2.

The specification describes the binding between the program and the aspect code. A specification is needed to match injection points in the program (join points) with injection code (advices). The binding is specified using matching strings (point cuts).

The main implications of this design are the following.

- We have an out-of-band instrumentation method. Whether we compile the program normally with the Python compiler or we compile it with aopy (using the specification file), we neither touch the source files or change how the program is run.
- Since we compile the program to bytecode, we have a static mutation. Every client of the program modules will always see their instrumented version.

[1]Bytecode files are not shown, because the distinction between source code and bytecode is not important in this case – we can treat them as one and the same.

- We have full reach, ie. there is no way to import a module by bypassing the API and getting the uninstrumented version.

3.2 Aspects and join points

One of the stated aims of aopy is to leverage Python's dynamic features for instrumentation. There are essentially three highly powerful features that make instrumentation convenient:

1. Properties can be used in classes to bind instance attributes to a getter/setter/deleter interface of methods.
2. Function decorators can be used to wrap functions calls, to intercept and examine their arguments, and to execute additional statements.
3. Metaclasses define how classes are instantiated from them. They can be used to mutate the namespace of a class before the class is initialized (and thus before instances of the class come into existence).

Even though program transformation makes it possible to inject or transform any statement in the program, these features of the language already give enough expressive power to transform classes and functions. They have the added benefit of being standard units of Python code, and can therefore be tested separately, and reused elsewhere.

Consequently, aopy has these three types of advice. Advices can then be collected in containers called aspects. The join points are classes and functions.

3.3 Point cuts

Point cuts are represented as "pathspecs", that is regular expressions which match object paths. The path separator / is used to match nested elements. The pathspec:

`directory/subdirectory/main:MyClass/member`

will match a module `main.py` under `directory/subdirectory`, relative to the path given to the compiler. Inside `main.py` it will match any attribute or function called `member` of a class called `MyClass`. Since pathspecs are regular expressions, they allow complex matching. To match anything at all one could use:

`.*:.*`

The only limitation on matching is that the type of advice will only be matched against the corresponding join point (a metaclass to a class, a decorator to a function etc.)

3.4 Programming interface

A specification module has to import the `aopy` module. This exposes a class `Aspect` with methods `add_<advice_type>`, taking a pathspec and the object to use as the advice.[2] The aspect objects to be used by the compiler have to be exported with the `__all__` assignment.

[2]There is no type checking performed here to verify that the object passed in valid. A metaclass passed to `add_decorator` will simply fail to match the pathspec, and this advice will fail silently.

```python
#<./spec.py>

# import aopy framework
import aopy

# import the objects I will use as aspects


# begin a new aspect
some_aspect = aopy.Aspect()

some_aspect.add_property(<pathspec>,
    fget=<getter>, fset=<setter>, fdel=<deleter>)

some_aspect.add_decorator(<pathspec>, <decorator_object>)

some_aspect.add_metaclass(<pathspec>, <metaclass_object>)


# export aspects to the compiler
__all__ = ['some_aspect']
```

3.5 The aopyc compiler

The compiler has three options:

1. parse the program and display the parse tree,
2. compile the module (unchanged) to bytecode,
3. transform the module (or path) given a specification module, compiling to bytecode.

```
Usage: ./aopyc -t spec.py ( module.py | path/ )

Options:
  -h, --help            show this help message and exit
  -s module, --show=module
                        show parse tree
  -c module, --compile=module
                        compile module
  -t specfile path, --transform=specfile path
                        transform files based on spec
  -v, --verbose         show parse tree after transformation
```

A demonstration of compilation is given in section 2.4.

If instead of a module to be transformed a path is given, the compiler will recursively find all modules as candidates for transformation, and match all aspects in all modules. It will only transform (and compile) a module if at least one match was made against it, however.

4 Program transformation

4.1 Mode of operation

The aopyc compiler imports the specification module and thus extracts all the objects that are to be injected into the program. As long as the specification module can resolve the object (which, by definition, it does – the object is live in the specification module), the path can be found with `moduleobj.__file__`.

The Aspect objects exported in the specification module, along with their pathspecs, are then aggregated to form a list of advices. We also have a list of candidate modules to which to apply instrumentation.

With each candidate module separately, we then proceed as follows.

1. Parse the module into abstract syntax.
2. Traverse syntax tree to find all names used in the module (class names, function names, variable names etc.)
3. Mangle module names that belong to advices, to prevent nameclashes.
4. Traverse syntax tree, matching nodes to pathspecs. Where a match is found, transformation is performed.
5. Traverse syntax tree with the [mangled] names of modules from which objects were injected, and add these as imports.
6. Compile module to bytecode.

4.1.1 Specification file format

The format of the specification file could in principle be arbitrarily different. One possibility is to make it more of a recipe file, with pathspecs and aspects written in a string format. However, it was considered that it would be best for specifications to be regular Python modules. That way they can be checked for syntax errors and type errors just by standalone execution, and those kinds of mistakes are caught long before aopy starts to process it.

4.1.2 Limitations of static analysis

The subsequent sections describe actual program transforms, which require some amount of static analysis. As such, one should keep in mind that static analysis cannot clarify all aspects of a program, least of all in a dynamic language like Python.

aopy's approach of transforming one module at a time fails to track operations upon objects that find place outside the module in which they are defined. It might seem prudent, then, to adopt a C compiler sort of approach: resolve all imports by importing all modules and use that as the search space for analysis. But there are two problems with this approach:

1. Imports are inherently dynamic, like everything else in Python. So the only way to know what will be imported is to execute the program.
2. Import chasing cannot track down all imports anyway, see 4.5.1.

In effect, there is only so much that *can* be discovered about a program. If an attribute is being reassigned in a class body, there is no way to know what the new object is, or even if it is of the same type as the old. This means that if an assignment exists for an attribute, this might be a property, but it could also be something else entirely, and there is no way to be sure.

4.2 Setting metaclass

Metaclasses are assigned by setting the class attribute `__metaclass__` in the body of a class. Setting `__metaclass__` more than once is invalid, so in order to inject a metaclass with aopy, all that is necessary is to remove any existing assignment to `__metaclass__`, and add the injected metaclass.

```
class MyMetaclass(type):
    def __new__(cls, name, bases, dct):
        return type.__new__(cls, name, bases, dct)

    def __init__(cls, name, bases, dct):
        super(MyMetaclass, cls).__init__(name, bases, dct)

class MyClass(object):
    #__metaclass__ = MyMetaclass   # removed
    __metaclass__ = myaspects.InjectedMetaclass   # injected
```

4.3 Setting properties

Properties in Python are realized through the mechanism of class instantiation (which, in fact, happens in the metaclass):

1. The class name, along with its base class (here, `object`), and all bindings made in the class body (functions and attributes alike), are passed to the `__new__` method in the metaclass. At this time, only class attributes can be assigned. The metaclass creates the class object.
2. Once the class object exists, the first instance of the class can be created by instantiating the class. Instantiating the class with parameters in effect dispatches the `__init__` method with those same parameters.

Since `__init__` is called on instantiation, it is customary to initialize instance attributes in its body. However, there is a small twist. If a class attribute exists with the same name, and if this attribute is a property object, then assignment will be made to the property, not the instance attribute directly.

```
class MyClass(object):
    def __init__(self):
        self.att = 1   # set via property

    att = property(fget=_get fset=_set fdel=_del)

    def _get(self):
        return self._att

    def _set(self, value):
        self._att = value

    def _del(self):
        pass
```

Naturally, instance attributes can be set in all instance methods, not just `__init__`. Thus, the first step to setting a property is to examine all instance methods to find all instance attributes.

4.3.1 Finding instance methods

Python classes have three types of functions. All of them are methods (this happens by virtue of being declared in the body of a class).

The code example shows three functions with identical parameters lists and function bodies. But in each case the parameter has a different meaning.

1. `ins_method` is an instance method. Unless stated otherwise, all functions in a class are instance methods. An instance method receives an implicit first argument (that is, the sender does not explicitly give it), which is the object in which the method lives. By convention, this first argument is called `self`.
   `self` is used to access both attributes of the object (`self.att`), as well as methods of the object (`self.method()`).

2. `cls_method` is a class method (that is, common to all instances). This is accomplished by using the built-in decorator `classmethod` to wrap the function name. Class methods receive an implicit first argument, which is the class the method lives in. By convention it is called `cls`. `cls` is used to access both attributes of the class (`cls.att`), as well as methods of the class (`cls.method()`).

3. `sta_method` is a static method. It is a function that lives in the class, and hence can be accessed through the class (and instances). This is broght about with the built-in `staticmethod` decorator. Static methods have no implicit arguments, and thus no access either to instances of their class, nor the class object itself.

```python
class MyClass(object):

    def ins_method(arg):
        arg.att = 1

    @classmethod     # decorator syntax
    def cls_method(arg):
        arg.att = 2
#    cls_method = classmethod(cls_method)    # legacy syntax

    @staticmethod    # decorator syntax
    def sta_method(arg):
        arg.att = 3
#    sta_method = staticmethod(sta_method)    # legacy syntax


myobj = MyClass()
myobj.ins_method()
myobj.cls_method()

class Other(object): pass
someobj = Other()
myobj.sta_method(someobj)

print(getattr(myobj, "att"))
print(getattr(MyClass, "att"))
print(getattr(someobj, "att"))

#> 1
#> 2
#> 3
```

In order to detect instance methods therefore, we look for methods annotated with `classmethod` or `staticmethod`, either with a decorator or through the older syntax, (see 4.4). If no such annotation exists, it must be an instance method.[3]

[3]It is, however, possible that an aliasing such as `newname = classmethod` could exist, and then our method of detection

4.3.2 Finding instance attributes

Once all instance methods have been found, we know that the first argument of every instance method is a reference to the instance. Thus we know the name by which `self` is called in this method. We can then find all assignments to attributes of `self` – these will be members of the instance.

```python
class MyClass(object):

    def some_method(arg1, arg2, arg3):
        value = self.compute_estimate(arg2)
        arg1.value = value    # assignment to attribute of instance
        intr = self.get_interest(value)
        arg3.intr = intr
```

Once we have found all assignments to instance attributes, we have found all instance attributes and thus we can determine if any of the point cuts match.[4]

For every matching point cut, we set the attribute's name in the class as a property. It is possible than an assignment to this name already exists (it could be a property or it could be for some other purpose). If so, it is removed.[5]

4.4 Adding decorators

Decorators were introduced to Python in version 2.4[4]. But even before that it was possible to use decorators informally, by rebinding the function name through a callable object. The semantics of the new decorator syntax were thus defined in terms of the existing method of rebinding function names, as shown in the code below.

In terms of ordering, it was thought reasonable that the decorator placed closest to the function definition be the innermost wrapping.

```python
@outer # Python 2.4 syntax
@inner
def func(*args, **kwargs):
    pass

func = outer(inner(func)) # Old syntax
```

If the two methods are mixed, the decorators listed atop the function will produce a function that has already been wrapped by the time the rebinding occurs with the old syntax. This means old-style decorators will wrap around new-style decorators.

The following example serves to illustrate. Suppose there are two decorators, `old` and `new`, respectively using the old syntax and the new, to assign two decorators to the function `compute`. A third decorator, `injected`, is injected via aopy. In syntactic terms, `injected` will be placed at the top of the new-style decorators. This placement is arbitrary, and could be made configurable within the new style decorators. However, reordering old-style decorators presents the challenge described in section 4.1.2.

would fail. But in such a case static analysis could not ascertain whether `newname == classmethod` holds. (Even if said assignment exists, it does not have to be in the current module, it could even be in a binary module.)

[4]It is possible for attributes to be assigned to objects outside of their class, outside the module of the class, in fact. Static analysis is helpless to detect this, however, so such attributes will not be matched by point cuts.

[5]If an assignment to the attribute name exists, it is most likely a property, and thus replacing it seems to be what is intended. We could also give the user the option to preserve such assignments.

```
def new(func):
    def new_func(*args, **kw):
        print("New decorator")
        return func(*args, **kw)
    return new_func

def old(func):
    def new_func(*args, **kw):
        print("Old decorator")
        return func(*args, **kw)
    return new_func

def injected(func):
    def new_func(*args, **kwargs):
        print("Injected decorator")
        return func(*args, **kwargs)
    return new_func

#@injected   # via aopy
@new
def compute(x):
    """Compute fun exponents"""
    print("Function")
    return x**x

compute = old(compute)

if __name__ == "__main__":
    compute(4)

#> Old decorator
#> Injected decorator
#> New decorator
#> Function
```

4.4.1 Parameterized decorators

Decorators can also be written in a parameterized fashion. In this style, there is an outer function holding the decorator, which receives the parameters the decorator expects. This outer function, when called, returns the decorator proper (`wrap`).

```
def logger(verbose=False):
    def wrap(func):
        def new_func(*args, **kwargs):
            if verbose:
                print("Injected decorator")
            return func(*args, **kwargs)
        return new_func
    return wrap

@logger(verbose=True)    # syntax
def func(*args, **kwargs):
    pass

func = logger(verbose=True)(func)    # semantics
```

Owing to the realization of aopy, parameterized decorators cannot be used as advice. The specification is evaluated, not parsed, thus the object aopy receives is, in fact, the `wrap` function.[6] `wrap` is not exposed at the top level of any module (as it is an inner function), and therefore an unresolved reference.

```
#<./myaspects.py>
def logger(verbose=False):
    def wrap(func):
        def new_func(*args, **kwargs):
            if verbose:
                print("Injected decorator")
            return func(*args, **kwargs)
        return new_func
    return wrap

#<./spec.py>
aspect.add_decorator(<pathspec>, myaspects.logger(verbose=True))

#<./main.py>

#@myaspects.wrap   # erroneous
@logger(verbose=True)
def func(*args, **kwargs):
    pass

func = logger(verbose=True)(func)
```

This limitation could likely be resolved by doing static analysis on the specification file.

4.5 Resolving references

Once we have established which objects have been injected into the program, we need to add an import statement to make sure they can be resolved. The following header is inserted at the top of the module.

```
import sys
for path in ("/path/to/aspects"):
    if path not in sys.path:
        sys.path.append(path)

import myaspects.cache.main as main
import myaspects.logger.main as main_
```

`path` ranges over the items in the path tuple. If the modules from which the objects were injected live in the same directory structure, this tuple will have a single element.

Following the path handling come the actual imports, with names mangled so as not to clash with any existing names in the module.

4.5.1 Import chasing

It is also possible to perform program transformation recursively, tracing all the imported modules (ie. libraries) used by the program. This way, the user could just as well instrument modules in the standard library. However, this method could fail to find and instrument all modules.

- If a module can be read, then it can be parsed, and its imports can be discovered.

[6]Strictly speaking, it is not so much the `wrap` *function* as it is the `wrap` *closure*, storing the values of variables bound at the point where `wrap` was declared; in this case the attribute `verbose`.

- If a module's bytecode file is not writable, the instrumented module, understandably, cannot be written.
- If the module is not a source module (it could be a bytecode module, or a shared library), it cannot be parsed, and its imports cannot be discovered.

A further restriction applies. Only static import statements will be detected, the example below will not.

```
mod = "mymodule"
exec("import %s" % mod)
```

References

[1] Martin Matusiak. *Strategies for aspect oriented programming in Python* 2009.

[2] The AspectJ Project *http://www.eclipse.org/aspectj/* 2008.

[3] Aspect oriented implementations *http://en.wikipedia.org/wiki/Aspect-oriented_programming#Implementations* 2008.

[4] Decorators for Functions and Methods *http://www.python.org/dev/peps/pep-0318/* 2008.