Descriptors

From Functional Wart
to
Decorator Madness
via
Properties

Long, long ago

In the deep darkness of the pre-2.2 era...

There was a wart

A wart of functions

A functional wart

A wart only metaprogrammers really cared about...

'I just want to be loved' said the metaprogrammers!

```
def hug():
    print 'hug'

class TeddyBear:
    DEFAULT_FUNC = hug
    def __init__( self, action=None ):
        self.love = action or self.DEFAULT_FUNC

TeddyBear(hug).love()

TeddyBear().love()
```

A pox on all methods! they cried

What evil magic is this?

(Forgetting all the good this magic did for them all the other days)

That old black magic

```
def __getattribute__( object, name ):
    if object.__dict__.has_key( name ):
        return object.__dict__[name]
    if class_lookup(object.__class__, name ) is not NULL:
        value = class_lookup( object.__class__, name )
        if isinstance( value, types.FunctionType ):
            return types.MethodType(
                value, object,
                object.__class__
            )
        else:
            return value
    if hasattr(object.__class__, '__getattr__' ):
        return object.__class__.__getattr__( object, name )
    raise AttributeError( name )
```

Geeky note

One attribute intercepted
One attribute overridden
One place the pattern seen
In the darkness of classic Python
Where the metaprogrammers cried

The age of logic begins...

Python 2.2 rationalised patterns

Prometheus retold

Made metaprogramming in Python practical

Ducks mate with Python 2.2

Duck-typing
and
protocols

Objects playing roles
regardless of their identity

Escape from the dungeon of C

2.2 introduced new hooks
to let Python programmers
metaprogram
without C

Now any fool with a tab key can create a new descriptor type

Or a new metaclass

Or (horrors) a metadata-driven web-framework

For those who missed the foreshadowing...

Obviously society is going to
crash and burn
in a few minutes

Attribute access becomes a tool of the metaprogrammer...

What rules go there!?

To lookup an attribute:

instance.__getattribute__(attrname)

The object does whatever it wants

Anarchy!

Rather too polite an anarchy...

Default (old-style) classes didn't fix the wart at all

But...

Another metaclass
could do something different

And become a hero to us all...

Thus

type and object

were born

How did they vanquish the wart?

What is a cleaner version
of the functional wart?

What was the general pattern
that underpinned it?

Know the nature of the wart

Objects in the class namespace
can intercept attribute-access
for instances

Tame the wart

Create hooks for two new points in the attribute-access mechanism

One matches the old functional wart

One answers the metaprogrammers' request to intercept a single attribute

New-style instance attributes

```
def __getattribute__( self, name ):
    cls = type(self)
    if class_lookup(cls, name ) is not NULL:
        desc = class_lookup( cls, name )
        if hasattr( desc, '__get__'):
            # is a descriptor...
            if not(
                hasattr( desc, '__set__') or
                hasattr( desc, '__delete__' )
            ):
                # non-data-descriptor, can be overridden
                if self.__dict__.has_key( name ):
                    return self.__dict__[name]
            return desc.__get__( self, cls )
        ...
```

New-style instance attributes (cont)

```
    elif (
        hasattr( desc, '__set__') or
        hasattr( desc, '__delete__') ):
        raise AttrbuteError( '__get__', desc )
    else:
        if self.__dict__.has_key( name ):
            return self.__dict__[name]
        else:
            return desc
elif self.__dict__.has_key(name):
    return self.__dict__[name]
elif name != '__getattr__' and hasattr(cls, '__getattr__'):
    return cls.__getattr__( self, name )
else:
    raise AttributeError( name )
```

Make the beast part of society...

Teach functions to use those hooks instead of relying on their special identity as functions

Allow non-function objects to play the same (or similar) roles

The first act of taming is naming

Needed way to describe those descriptors which hooked one point versus the other

“Non-data descriptors”

Only intercept attribute lookup on the class

Overridden by instance-attributes

Function-like Descriptors

```
class Function( types.FunctionType ):
    """What a function descriptor looks like"""
    def __get__( self, client, cls ):
        """Retrieve/calculate the value for client instance"""
        if client is not None:
            return types.MethodType( self, client, type(client) )
        else:
            return types.UnboundMethodType(self, None, cls)
```

“Data descriptors”

Have __set__ or __delete__

Intercept lookup from both instance and class (oh, and allow for setting values on the instance)

Attribute-like Descriptors

```
class Descriptor( object ):
    """A simple descriptor"""
    def __get__( self, client, cls ):
        """Retrieve/calculate the value for client instance"""
        if client is not None:
            return client.__dict__[ 'hello' ]
        else:
            return self
    def __set__( self, client, value ):
        """Set the value on the client instance"""
        client.__dict__[ 'hello' ] = value
    def __delete__( self, client ):
        """Delete the value from the client instance"""
        del client.__dict__[ 'hello' ]
```

The canon of 2.2 descriptors...

classmethod
Method takes first argument as class

staticmethod
Method ignores first argument

property
Attribute created from accessor/mutator functions/methods

Society is bemused

Most programmers look at the core descriptors and yawn

What's the big deal?

Methods that ignore an argument...
Attributes made of 3 functions...

Not exactly what they asked
Santa to bring

But metaprogrammers quietly start to play with the new features...

Oh, how ominous!

The “Elven” descriptor packages

“Attributes that”

Typed-oo heritage, fields/properties

“Building castles in the air”

One or two types per system

OpenGLContext (VRML97 fields)

- Typing (mostly Numpy arrays)
- Defaults
- Observability (cache operations)
- Introspection
- Domain-specific

BasicProperty, PyTable, wxoo

- Typing and validation
- Defaults
- Introspection (wxoo editor, web editor)
- General domain modelling framework

Zope

- FieldProperty, DublinCore
- Data validation, error messages
- Defaults
- Introspection
- Observability

PEAK

- Automatic hierarchic maintenance
- Value acquisition (defaults, delegation)
- Wrap loaded features to look like attrs

Traits (almost descriptors)

- Delegation to other objects
- Typing and data validation
- Defaults
- Observability
- Introspection (w/GUI library editors)
- Descriptor-like, not actual descriptors

The “Dwarven” descriptors

“Functions that”

Non-data descriptors

“Hammering on the metal”

+ lots of different low-level operations

FFI/C-code Wrapping

PyObjC, ctypes, JythonC, IronPython

All declare lots of metadata about functions (parameter and return types, calling convention, DLL sources)

Decorating masses...

Lock-protected methods
Type-dispatched compound methods
Result-caching/memoizing methods
Database-aware methods
Currying methods
Pre/post-conditioned methods
Constant-binding methods
Docstring mutating methods
Error-catching methods
Type-checking methods

Even more massing hordes...

Type converting methods
Generator wrapping methods
Deprecated/warning/abstract methods
Logged/call-counted methods

Metamasses cry for decorators!

Syntax for classmethod and staticmethod was always planned

But it was ctypes & co that seemed to carry the day for getting decorator syntax into 2.4

Here's why

```
def doSomething( a,b,c ):
    """Do something via FFI"""
doSomething = protected( someLock )( doSomething )
doSomething = typed( str, int, str )( doSomething )
doSomething = calltype( WINDLL )( doSomething )
doSomething = fromDLL( myDLL )( doSomething )
```

Let 10,000 messages deluge c.l.p

Much heat and noise deciding the syntax

We did (eventually) get a syntax

(For those who stopped reading c.l.p during the debate)

Decorators help with the pain...

```
@fromDLL( myDLL )
@calltype( WINDLL )
@typed( str, int, str )
@protected( someLock )
def doSomething( a,b,c ):
    """Do something via FFI"""
```

The gathering storm...

Throughout the debate on decorators
(and to this day)
there is an assumption that these
Dwarven descriptors, and particularly
“decorated” functions will become more
common, that they will multiply
exponentially

The ravening hordes

And now we have decorators breeding in the blogs and wikis of the metaverse

Thirsting for our functions

Waiting to make every method an essay in magic, a surprise, a wonder

This decorator magic is a powerful force

We must use it wisely

Or risk falling to the dark side

And losing the simplicity that made Python great

Forcing every programmer to become a meta-magician just to debug their 5 line script

When magic rules

There are no rules.