

Python – Part 2  
Meta-Classes in Python  
(Formerly known as “The Killer Joke”)

Guy Wiener

24/6/2007

1/1

## Meta-Classes in Python

Meta-classes in Python are responsible for **creating** classes.

- ▶ Meta-classes extend the type “type”
- ▶ The “`__new__`” method of the meta-class creates an instance of the meta-class that is the new class

Type arguments: `__new__(cls, name, bases, dict)`

`cls` The class object  
`name` The name of the class  
`bases` A list of the base classes  
`dict` The inner dictionary of the new class

(Same can apply to `__init__` using `getattr/setattr` on `cls`)

3/1

## Creating a New Class Directly

Creating an instance of a meta-class declares a new class.

Direct-style class creation

```
class Printable(type):
    def whoami(self): print "I am a", self.__name__

>>> Foo = Printable('Foo', (), {})
>>> Foo.whoami()
I am a Foo
>>> f = Foo()
>>> f.__class__
<class '__main__.Foo'>
```

4/1

## The `__metaclass__` field

Class C has a meta-class M if:

1. C has a field `__metaclass__`
2. One of the ancestors of C has a meta-class
3. There is a global variable `__metaclass__`
4. Otherwise, the default meta-class type is used

5/1

## `__metaclass__` Example

Declaring a meta-class with `__metaclass__`

```
class Bar:
    __metaclass__ = Printable
    def foo(self): print 'foo'

>>> Bar.whoami()
I am a Bar
>>> b = Bar()
>>> b.foo()
foo
```

6/1

## Logging with Meta-Classes

A logger decorator

```
def log(name, f):
    def ret(*args, **kw):
        print "enter", name
        f(*args, **kw)
        print "exit", name
    return ret
```

8/1

## Logging with Meta-Classes (cont'd)

A logger meta-class

```
class Logged(type):
    def __new__(cls, name, bases, dict):
        p = re.compile(dict['_logmatch_'])
        for attr, item in dict.items():
            if callable(item) and p.match(attr):
                dict[attr] = log(attr, item)
        return type.__new__(cls, name, bases, dict)
```

9 / 1

## Loggin with Meta-Classes (cont'd)

A logged class

```
class Test:
    __metaclass__ = Logged
    __logmatch__ = '.*'
    def foo(self):
        print 'foo'
```

```
>>> t = Test()
>>> t.foo()
enter foo
foo
exit foo
```

10 / 1

## Automatic Properties

We want to create classes C that has JavaBeans-like properties:

- ▶ `c.x` calls `c._get_x()`
- ▶ `c.x = y` calls `c._set_x(y)`

We can use the Python auxiliary type property:

- ▶ Takes two functions as arguments, setter and getter
- ▶ Maps them to get and assign operations

11 / 1

## Automatic Properties with Meta-Classes

Auto-props meta-class

```
class Autoprop(type):
    def __init__(cls, name, bases, dict):
        type.__init__(name, bases, dict)
        props = {}
        for name in dict.keys():
            if name.startswith("_get_") or \
                name.startswith("_set_"):
                props[name[5:]] = 1
        for name in props.keys():
            fget = getattr(cls, "_get_%s" % name, None)
            fset = getattr(cls, "_set_%s" % name, None)
            setattr(cls, name, property(fget, fset))
```

12 / 1

## Automatic Properties with Meta-Classes (cont'd)

Using auto-props

```
class InvertedX:
    __metaclass__ = Autoprop
    def _get_x(self):
        return self._x
    def _set_x(self, x):
        self._x = -x
```

```
i = InvertedX()
i.x = 12
print i.x
>>> -12
```

13 / 1

## Automatic Delegation

Delegate An object of class A that dispatches all message of class B to an instance of that class, the *target*.

- ▶ Writing a delegate class is a monotonous work
- ▶ But it can be done automatically

14 / 1

## Delegation Using Meta-Classes

A delegation function decorator

```
def dlgt(cls, mthd):
    def ret(self, *args, **kw):
        mthd(self.__tgt__, *args, **kw)
    return instancemethod(ret, None, cls)
```

instancemethod takes a function, an object/None and a class and returns a method of the class

Auxiliary – Print class name

```
def clsname(self):
    return self.__class__.__name__
```

15 / 1

## Delegation Using Meta-Classes (cont'd)

The Delegate meta-class

```
class Delegate(type):
    def __init__(cls, name, bases, dict):
        type.__init__(cls, name, bases, dict)
        src = getattr(cls, '__tgtclass__', None)
        for attr in dir(src):
            val = getattr(src, attr, None)
            if callable(val):
                setattr(cls, attr, dlgt(val))
```

16 / 1

## Delegation Using Meta-Classes (cont'd)

The delegated class

```
class A:
    def bar(self):
        print clsname(self), 'bar'
    def baz(self):
        print clsname(self), 'baz'
```

17 / 1

## Delegation Using Meta-Classes (cont'd)

The delegating class

```
class B:
    __metaclass__ = Delegate
    __tgtclass__ = A
    def __init__(self, tgt):
        self.__tgt__ = tgt
    def boo(self):
        print clsname(self), 'boo'
```

18 / 1

## Delegation Using Meta-Classes (cont'd)

Delegation test

```
b = B(A())
b.bar()
>>> A bar
b.baz()
>>> A baz
b.boo()
>>> B boo
```

19 / 1