CSC326 Meta Programming

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME
1.0	2011-09		JZ

Contents

1 Agenda

- Python Types and Objects
- Meta Classes
- Decorators
- Aspect-oriented Programming

2 Class Factory

- Go through Shalabh Chaturvedi's tutorial

3 Meta Class

- Class factory
  - Class that acts as template for other classes
- Old fashioned: a function

```
>>> def class_with_method(func):
...     class klass: pass
...     setattr(klass, func.__name__, func)
...     return klass
...
>>> def say_foo(self): print 'foo'
...
>>> Foo = class_with_method(say_foo)
>>> foo = Foo()
>>> foo.say_foo()
foo
```

- Another way

```
#----- Class factory in the [new] module (python) -------#
>>> from new import classobj
>>> Foo2 = classobj('Foo2',(Foo,),{'bar':lambda self:'bar'})
>>> Foo2().bar()
'bar'
>>> Foo2().say_foo()
foo
```

- Use type
  - special class type is just a class factory: same as new.classobj

```
>>> X = type('X',(),{'foo':lambda self:'foo'})
>>> X, X().foo()
(<class '__main__.X'>, 'foo')
```

- Inherit from type

```
>>> class ChattyType(type):
...     def __new__(cls, name, bases, dct):
...         print "Allocating memory for class", name
...         return type.__new__(cls, name, bases, dct)
...     def __init__(cls, name, bases, dct):
...         print "Init'ing (configuring) class", name
...         super(ChattyType, cls).__init__(name, bases, dct)
...
>>> X = ChattyType('X',(),{'foo':lambda self:'foo'})
Allocating memory for class X
Init'ing (configuring) class X
>>> X, X().foo()
(<class '__main__.X'>, 'foo')
```

- Setting metaclass of a class

```
>>> class Printable(type):
...     def whoami(cls): print "I am a", cls.__name__
...
>>> Foo = Printable('Foo',(),{})
>>> Foo.whoami()
I am a Foo
>>> Printable.whoami()
Traceback (most recent call last):
TypeError:  unbound method whoami() [...]

#---- Setting metaclass with class attribute (python) ---#
>>> class Bar:
...     __metaclass__ = Printable
...     def foomethod(self): print 'foo'
...
>>> Bar.whoami()
I am a Bar
>>> Bar().foomethod()
foo
```

- When do you use it?
  - NEVER!
  - You will know when you do

4 Decorator

- You have seen it!

```
@route('/:name')
def index(name='World'):
    return '<b>Hello %s!</b>' % name
```

- Ever wonder what they are?
  - modify the function that is defined immediately after

- Disclaimer: you can do it without them — syntactic sugar
  - but life is a lot easier with them

```
class C:
    def foo(cls, y):
        print "classmethod", cls, y
    foo = classmethod(foo)
```

```
def enhanced(meth):
    def new(self, y):
        print "I am enhanced"
        return meth(self, y)
    return new
class C:
    def bar(self, x):
        print "some method says:", x
    bar = enhanced(bar)
```

- Simple magic
  - avoid repeating the method name,
  - and put the decorator near the first mention of the method
- Work for regular functions too
- Can be chained

```
@synchronized
@logging
def myfunc(arg1, arg2, ...):
    # ...do something
# decorators are equivalent to ending with:
#    myfunc = synchronized(logging(myfunc))
# Nested in that declaration order
```

```
class C:
    @classmethod
    def foo(cls, y):
        print "classmethod", cls, y
    @enhanced
    def bar(self, x):
        print "some method says:", x
```

5 Misuse of Decorators

- Not returning a function

```
>>> def spamdef(fn):
...     print "spam, spam, spam"
...
>>> @spamdef
... def useful(a, b):
...     print a**2 + b**2
```

```
...
spam, spam, spam
>>> useful(3, 4)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: 'NoneType' object is not callable
```

- Not returning a **meaningful** function

```
>>> def spamrun(fn):
...     def sayspam(*args):
...         print "spam, spam, spam"
...     return sayspam
...
>>> @spamrun
... def useful(a, b):
...     print a**2 + b**2
...
>>> useful(3,4)
spam, spam, spam
```

- Right one

```
>>> def addspam(fn):
...     def new(*args):
...         print "spam, spam, spam"
...         return fn(*args)
...     return new
...
>>> @addspam
... def useful(a, b):
...     print a**2 + b**2
...
>>> useful(3,4)
spam, spam, spam
25
```

6 Using Decorators

```
def elementwise(fn):
    def newfn(arg):
        if hasattr(arg,'__getitem__'):  # is a Sequence
            return type(arg)(map(fn, arg))
        else:
            return fn(arg)
    return newfn

@elementwise
def compute(x):
    return x**3 - 1

print compute(5)        # prints: 124
print compute([1,2,3])  # prints: [0, 7, 26]
print compute((1,2,3))  # prints: (0, 7, 26)
```

7 Aspect Oriented Programming

- Seperatating cross-cutting concerns
  - Common across a number of classes, methods and functions
- Category of aspects
  - Debugging: logging function argments, entry and exit
  - Type safty checks
  - Deprecation warnings
  - Database transactions
  - Authorization
  - Profiling

```
def trace( aFunc ):
    """Trace entry, exit and exceptions."""
    def loggedFunc( \*args, \*\*kw ):
        print "enter", aFunc.__name__
        try:
            result= aFunc( \*args, \*\*kw )
        except Exception, e:
            print "exception", aFunc.__name__, e
            raise
        print "exit", aFunc.__name__
        return result
    loggedFunc.__name__= aFunc.__name__
    loggedFunc.__doc__= aFunc.__
    return loggedFunc
```

```
class MyClass( object ):
    @trace
    def __init__( self, someValue ):
        """Create a MyClass instance."""
        self.value= someValue
    @trace
    def doSomething( self, anotherValue ):
        """Update a value."""
        self.value += anotherValue
```

- Class is a factory of objects
  - Call Point() to create instance as if it were a function
  - returns a "reference" to a Point object

8 Decorator Tool

```
>>> from decorator import decorator
>>> @decorator
... def addspam(f, *args, **kws):
...     print "spam, spam, spam"
...     return f(*args, **kws)
>>> @addspam
... def useful(a, b): return a**2 + b**2
>>> useful.__name__
'useful'
```

9 Usage in Bottle

```
from bottle import route, run

@route('/:name')
def index(name='World'):
    return '<b>Hello %s!</b>' % name

run(host='localhost', port=8080)
```

- What was happening?
  - When a callback function like index is defined, corresponding URL registration and binding was done
  - At Runtime, web framework parse URL and route to call back

10 Recap

- Everything is an object
- Objects are different
  - Meta classes
  - classes
  - non-type instances
- Be clear of relations
  - instanceof
  - typeof
- classes as first class citizen
  - class factories
  - decorators