

# Class Decorators Radically Simple

Jack Diederich  
[jackdied.blogspot.com](http://jackdied.blogspot.com)  
PyCon 2009

# Speaker's Afterwards 1/2

[stuff that came up in Qs or in the podium scrum just after]

Q: How do I write a class decorator that curries `__init__` args?

A: This is a curry operation so we want to pass in args. Decorators only accept one argument (the class to be decorated) so you need to make a function that accepts the curry args and returns a decorator. You don't have to replace the original class [I said you did during the Q/A], you just have to replace the `__init__`

```
def curry_args(*args):
    def curry_decorator(cls):
        old_init = cls.__init__
        def new_init__(self, *extra_args):
            combined_args = list(args) + list(extra_args)
            old_init__(self, *combined_args)
        cls.__init__ = new_init
    return curry_decorator
```

```
@curry_args(1, 2, 3)
class C():
    def __init__(a, b, c, d):
        self.args = a, b, c, d
```

```
>>>c = C(4)
>>>c.args
(1, 2, 3, 4)
>>>
```

# Speaker's Afterwards 2/2

[stuff that came up in Qs or in the podium scrum just after]

\* Alex Martelli would like to correct an attribution I made. I attributed the metaclass multiple inheritance recipe to him (I saw it in a talk he did a few years back) but it actually comes from Michele Simionato (w/ some contributions from Hettinger). You can see the ASPN recipe here:

<http://code.activestate.com/recipes/204197/>

\* David Mertz stopped by to say hi. He's one of those guys I've "seen" on python-dev for ages but somehow never met. It wasn't a strictly social visit. He pointed out that metaclasses can move arguments out of the class body starting in 3.x. Here is a rewrite of my cron.scheduler that sticks the metaclass args in the classdef instead of the class body.

```
class SalesReport(metaclass=cron.schedule, when=cron.NIGHTLY): pass
```

This isn't a huge improvement. You have to know that the 'when' argument belongs to the cron.schedule metaclass, and avoid using the same keywords between metaclasses. For reasons like that I was against allowing arbitrary arguments in the class declaration. The good news is that no one uses it.

Finally, Thanks to everyone who came to my talk. I enjoyed giving it. This was V3.0 of the talk (it started as a lightning talk, v1 was at EuroPython, v2 at UKPython). It's about as good as it going to get so I'm retiring it.

I'm looking for a new gig so if you have an interesting product and need a guy (full time or consulting) please drop me a line.

# Function Decorators

```
class A():  
    @staticmethod  
    def function(): pass
```

```
class A():  
    def function(): pass  
    function = staticmethod(function)
```

# Class Decorators

```
@my_decorator  
class A():  
    pass
```

```
class A():  
    pass  
A = my_decorator(A)
```

# Practical Definition

- Function that takes one argument
- Returns something *useful*

# Formal Definition

- A Callable
- That accepts at least one argument  
(but doesn't require more than one or keywords)
- Returns something

# Valid Decorators

```
def identity(ob):  
    return ob
```

```
@identity  
class C():  
    pass
```

```
>>>C  
<class '__main__.C'>  
>>>
```

# Valid Decorators

```
def hello_world(ob):  
    return 'Hello World'
```

```
@hello_world  
class C():  
    pass
```

```
>>>C  
'Hello World'  
>>>
```

# Valid Decorators

```
def replace_with_X(ob):  
    class X():  
        pass  
    return X
```

```
@replace_with_X  
class C():  
    pass
```

```
>>>C  
<class __main__.X>  
>>>
```

# Valid Decorators

```
def instantiate(ob):  
    return ob()
```

```
@instantiate  
class C():  
    pass
```

```
>>>C  
<__main__.C object at 0xb7c1176c>  
>>>
```

# Valid Decorators

```
def decorator_maker(*args):  
    def instantiate(ob):  
        return ob(*args)  
    return instantiate
```

```
@decorator_maker(1, 2, 3)  
class C():  
    def __init__(self, *args):  
        self.args = args
```

```
>>>C  
<__main__.C object at 0xb7c1170a>  
>>>C.args  
(1, 2, 3)  
>>>
```

# Valid Decorators

```
def hello_world(*args): return 'Hello World'
```

```
def bofh(ob):
```

```
    class X(metaclass=hello_world):
```

```
        __slots__ = 'Hello World'
```

```
        def __getattr__(self, name):
```

```
            return random.choice(self.__dict__.items())
```

```
    return X
```

```
@bofh
```

```
class C(): pass
```

# Valid Decorators

```
def hello_world(*args): return 'Hello World'
```

```
def bofh(ob):
```

```
    class X(metaclass=hello_world):
```

```
        __slots__ = 'Hello World'
```

```
        def __getattr__(self, name):
```

```
            return random.choice(self.__dict__.items())
```

```
    return X
```

```
@bofh
```

```
class C(): pass
```

```
>>>C
```

```
'Hello World'
```

```
>>>
```

# Decorator History

- Function decorators in Python 2.4
- Class decorators in Python 2.6 and 3.0

decorated: decorators (classdef | funcdef)

@deco\_c

@deco\_b

@deco\_a

(def|func) NAME(arglist):

# Versus Metaclasses

- Interface

```
def identity(ob):  
    return ob
```

```
class Identity(type):  
    def __new__(meta, name, bases, dict):  
        return type.__new__(meta, name, bases, dict)  
    def __init__(cls, name, bases, dict): pass
```

# Versus Metaclasses

```
import cron
```

```
@cron.schedule(cron.NIGHTLY)
```

```
class SalesReport(Report):
```

```
    def run(self):
```

```
        # do stuff
```

```
class SalesReport(Report, metaclass=cron.meta):
```

```
    cron_when = NIGHTLY
```

# Versus Metaclasses

@deco

```
class A(X): pass
```

@deco

```
class B(X): pass
```

```
class C(Z): pass
```

```
class D(Z): pass
```

# Versus Metaclasses

- Decorators are stackable

```
@cron.schedule(cron.NIGHTLY)
```

```
@document(level='top')
```

```
class SalesReport(Report):
```

```
    pass
```

# Versus Mixins

```
@dict_methods  
class DictLike():  
    def __getitem__(self, key):  
        return self.data[key]
```

```
class DictLike(UserDict.DictMixin):  
    def __getitem__(self, key):  
        return self.data[key]
```

# Popular Patterns

- Register
- Augment
- Fixup
- Verify

# Verify Non-Pattern

```
def assert_candy(cls):  
    assert cls.sweet == True  
    assert cls.calories > 100  
    return cls
```

```
@assert_candy  
class ChocolateBar():  
    sweet = True  
    calories = 200
```

# Registration Pattern

```
import cron
```

```
@cron.schedule(cron.NIGHTLY)
```

```
class SalesReport(Report):
```

```
    def run(self):
```

```
        # do stuff
```

```
class SalesReport(Report, metaclass=cron.meta):
```

```
    cron_when = NIGHTLY
```

# Augment Pattern

```
@total_ordering
class NumberLike():
    def __lt__(self, other):
        return self.data < other.data
```

# Registration Decorator

```
class Factory():  
    def __init__(self):  
        self.all = []  
    def register(self, cls):  
        self.all.append(cls)  
        return cls
```

```
animals = Factory()
```

```
@animals.register  
class Horse(): pass
```

# Registration Metaclass

```
def new_factory_type():  
    class FactoryMeta(type):  
        all = []  
        def __init__(cls, name, bases, dict):  
            FactoryMeta.all.append(cls)  
            return type.__init__(cls, name, bases, dict)  
    return FactoryMeta
```

```
animals = new_factory_type()
```

```
class Horse(metaclass=animals): pass
```

# Registration Metaclass

```
def new_factory_type():
    class FactoryMeta(type):
        all = []
        def __init__(cls, name, bases, dict):
            if getattr(cls, 'DO_NOT_REGISTER'):
                delattr(cls, 'DO_NOT_REGISTER')
            else:
                FactoryMeta.all.append(cls)
            return type.__init__(cls, name, bases, dict)
    return FactoryMeta
```

```
animals = new_factory_type()
```

```
class Horse(metaclass=animals): pass
```

# Augment Pattern

```
def total_ordering(cls):  
    """ implement all rich comparison operators """  
    setattr(cls, '__ne__', lambda self, oth: self < oth or oth < self)  
    setattr(cls, '__eq__', lambda self, oth: not self != oth)  
    setattr(cls, '__gt__', lambda self, oth: oth < self)  
    setattr(cls, '__ge__', lambda self, oth: not (self < oth))  
    setattr(cls, '__le__', lambda self, oth: not (self > oth))  
    return cls
```

# Fixup Pattern

```
def stop_writing_java(cls):
    """ de-privatize access to self.__bar attributes """
    private = '_' + cls.__name__ + '_'

    def new_getattr(self, key):
        if key.startswith(private):
            key = key[len(private):]
        return self.__dict__[key]

    def new_setattr(self, key, value):
        if key.startswith(private):
            key = key[len(private):]
        self.__dict__[key] = value

    cls.__getattr__ = new_getattr
    cls.__setattr__ = new_setattr
    return cls
```

# Fixup Pattern

```
@stop_writing_java  
class Java():  
    def __init__(self):  
        self.__sekret = True
```

```
>>> ob = Java  
>>> print(ob.__sekret)  
True
```

```
# monkey patch a library  
import ugly_lib  
ugly_lib.Java = stop_writing_java(ugly_lib.Java)
```

# Best Practices

- Return the original class
- Don't assume you are the only decorator
- Maybe you want a metaclass
- Don't add `__slots__`

# Links

jackdied.blogspot.com # slides, python blog

<http://www.ibm.com/developerworks/linux/library/l-cpdecor.html>

“Decorators Make Magic Easy” by David Mertz

[http://www.voidspace.org.uk/python/weblog/arch\\_d7\\_2008\\_10\\_04.shtml](http://www.voidspace.org.uk/python/weblog/arch_d7_2008_10_04.shtml)

Total ordering decorator.

<http://docs.python.org/library/functools.html>

Functools module.

Jack Diederich

jackdied@gmail.com

PyCon on the Charlels 2009

# Tidy Decorators

```
import functools
```

```
# use update_wrapper to wrap our decorator
```

```
@functools.update_wrapper
```

```
def trace(func):
```

```
    def func_wrapper(*args):
```

```
        """ log the function call """
```

```
        logging.debug(repr(func, args))
```

```
        return func(*args)
```

```
    return func_wrapper
```

```
@trace # now preserves signature and docstring
```

```
def func(a, b):
```

```
    """ return a + b """
```

```
    return a + b
```