Magia Negra en Python

Lucio Torre (lucio@movilogic.com)

Temario

- Descriptores
- Decoradores
- Metaclases

Un problema

```
class Clase(object):
    def __init__(self):
        self.propiedad = 1

instancia = Clase()
print instancia.propiedad
```

Solucion #1

```
class Clase(object):
      def __init__(self):
            self.propiedad = 1
      def __getattr__(self, name):
         if name == “propiedad”:
            print “Acceso a propiedad”
         return self.__dict__[name]

instancia = Clase()
print instancia.propiedad
```

Problemas

- Atomicidad
- Localidad

Descriptores

Learning about descriptors not only provides access to a larger toolset, it creates a deeper understanding of how Python works and an appreciation for the elegance of its design.

Raymond Hettinger

"How-To Guide for Descriptors"

http://users.rcn.com/python/download/Descriptor.htm

Que son descriptors

- Un protocolo que modifica el mecanismo de acceso a atributos

Descriptor protocol

```
descr.__get__(self, obj, type=None) --> value

descr.__set__(self, obj, value) --> None

descr.__delete__(self, obj) --> None
```

Como funcionan?

- __getattribute__ de new style classes implementa el lookup por descriptors (object.c, linea 1258: PyObject_GenericGetAttr)

Busca el nombre en las clases base

```
def __getattr__(self, name):
            getter = None
            desc = None

        for base in type(self).mro():
            desc = base.__dict__.get(name, None)
            if desc: break
```

Si encontro el objeto, se fija si es un data descriptor y devuelve su valor

```
if desc:
  if isinstance(desc,object):
    if hasattr(desc, __get__):
      getter = desc.__get__
      if hasattr(desc, __set__):
        return getter(o, type(o))
```

Si es un atributo de instancia, devuelve ese valor

```
if name in self.__dict__:
    return self.__dict__[name]
```

Si es un non-data descriptor, lo ejecutamos.
Sino, seguimos el flujo normal.

```
if getter:
   return getter(o, type(o))

# sino, devolvemos el objeto que encontramos en las bases
 if desc:
    return desc

raise AttributeError("'%.50s' object has no attribute '%.
400s'"%(type(o).__name__, name))
```

Detalles

- Los descriptores se crean al momento de creacion de la clase
- No es posible editar una instancia para agregarle descriptores

Solucion #1

```
class Clase(object):
      def __init__(self):
            self.propiedad = 1
      def __getattr__(self, name):
         if name == “propiedad”:
            print “Acceso a propiedad”
         return self.__dict__[name]

instancia = Clase()
print instancia.propiedad
```

Solucion #2

```
class getter(object):
    def __init__(self, initval=None):
        self.val = initval

    def __get__(self, obj, objtype):
        print “Acceso a propiedad”
        return self.val

class Clase(object):
    propiedad = getter(1)

instancia = Clase()
print instancia.propiedad
```

property([fget[, fset[, fdel[, doc]]]])
    Return a property attribute for new-style classes (classes that derive from object).

    fget is a function for getting an attribute value, likewise fset is a function for setting, and fdel a function for del'ing, an attribute.

```
class C(object):
    def __init__(self): self.__x = 1
    def x(self): return self.__x
    x = property(getx)
```

New in version 2.2.

Solucion #3

```
class Clase(object):
    def __init__(self):
        self._propiedad = 1

    def propiedad(self):
        print “Acceso a propiedad”
        return self._propiedad
    propiedad = property(propiedad)

instancia = Clase()
print instancia.propiedad
```

Decoradores

```
@deco
def func(arg1, arg2, ...):
    pass
```

```
def func(arg1, arg2, ...):
    pass
func = deco(func)
```

```
@dec2
@dec1
def func(arg1, arg2, ...):
    pass
```

```
def func(arg1, arg2, ...):
    pass
func = dec2(dec1(func))
```

```
@decomaker(argA, argB, ...)
def func(arg1, arg2, ...):
    pass
```

==

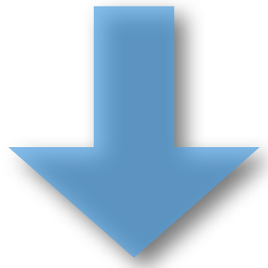
```
deco = decomaker(
        argA, argB, ...)
@deco
def func(arg1, arg2, ...):
    pass
```

```
@decomaker(argA, argB, ...)
def func(arg1, arg2, ...):
    pass
```

```
def func(arg1, arg2, ...):
    pass
func = decomaker(
    argA, argB, ...
    )(func)
```

```
class C(object):
        def __init__(self): self.__x = 1

        def getx(self): return self.__x
        x = property(getx)
```



```
class C(object):
        def __init__(self): self.__x = 1

        @property
        def x(self): return self.__x
```

Solucion #3

```
class Clase(object):
    def __init__(self):
        self._propiedad = 1

    def propiedad(self):
        print “Acceso a propiedad”
        return self._propiedad
    propiedad = property(propiedad)

instancia = Clase()
print instancia.propiedad
```

Solucion #4

```
class Clase(object):
    def __init__(self):
        self._propiedad = 1

    @property
    def propiedad(self):
        print “Acceso a propiedad”
        return self._propiedad

instancia = Clase()
print instancia.propiedad
```

Decoradores

Syntax Sugar

Porque Decoradores?

- Soportar metaprogramming de manera:
  - Legible: Al comienzo, no al final, de la funcion
  - Menos repetitiva: No se repite el nombre de la funcion
  - Simple: No tener que recurrir a metaclases para cosas mas simples

Otros Usos

- Method Signature (control de tipos, integracion con otros lenguajes, etc)
- Tracing
- Sincronizacion
- Memoizacion
- Control de acceso
- Etc

Clases

Por ahora no se pueden decorar clases

I propose that someone start writing a Py3k PEP for class decorators. I don't think it's fair to the 2.5 release team to want to push this into 2.5 though; how about 2.6?

GvR

http://mail.python.org/pipermail/python-dev/2006-March/062942.html

Metaclases

[Metaclasses] are deeper magic than 99% of users should ever worry about. If you wonder whether you need them, you don't (the people who actually need them know with certainty that they need them, and don't need an explanation about why).

Tim Peters (c.l.p post 2002-12-22)

```
class Name(Ba,Ses):
   <<body>>
```

```
Name = type('Name', (Ba,Ses),
    <<dict-built-by-body>>)
```

```
class Name(Ba,Ses):
    __metaclass__ = my_metaclass

    <<body>>
```

```
Name = my_metaclass('Name', (Ba,Ses),
    <<dict-built-by-body>>)
```

Algunos usos

- Registracion de clases
- Verificacion de clases
- Construccion de clases
- Mucho mas.

```
clases = {}

def registrar(nombre, bases, parametros):
        cls = type(nombre, bases, parametros)
        clases[nombre]=cls
        return cls

class Foo:
        __metaclass__ = registrar

assert 'Foo' in clases
```

Cuidado

```
clases = {}

def registrar(nombre, bases, parametros):
        cls = type(nombre, bases, parametros)
        clases[nombre]=cls
        return cls

class Foo:
        __metaclass__ = registrar
class Bar(Foo): pass

assert 'Bar' in clases -> AssertionError
```

Que paso?

```
class Foo:
        __metaclass__ = registrar
class Bar(Foo): pass
```

==

```
class Foo:
        __metaclass__ = registrar
class Bar(Foo):
        __metaclass__ = Foo.__class__ #Foo.__class__ == type
```

Solucion

```
class registrar(type):
    def __init__(cls, nombre, bases, parametros):
        clases[nombre]=cls
```

Preguntas?

python.org/ar

pyar-subscribe@decode.com.ar

- http://www.python.org/pycon/dc2004/papers/24/metaclasses-pycon.pdf : Python Metaclasses: Who? Why? When?
- http://users.rcn.com/python/download/Descriptor.htm : How-To Guide for Descriptors
- http://www.python.org/dev/peps/pep-0318/ : PEP 318
- http://mail.python.org/pipermail/python-dev/2005-November/057906.html : [Python-Dev] Class decorators vs metaclasses
- http://docs.python.org/lib/built-in-funcs.html#l2h-55 : property builtin