

# **Python Internals**

## Rosario Di Somma

# Descriptor

A descriptor is an object attribute with “binding behavior”, one whose attribute access has been overridden by methods in the descriptor protocol.

*Raymond Hettinger*

- Protocollo dei descrittori
  - `__get__(obj, type=None) --> value`
  - `__set__(obj, type=None) --> value`
  - `__delete__(obj, type=None) --> value`
- Un oggetto e' considerato un descrittore se definisce almeno uno dei metodi elencati
- Due tipi di descrittori
  - overriding descriptor
    - definisce entrambi i metodi `__get__`, `__set__`
  - nonoverriding descriptor
    - define solo il metodo `__get__`

# Un semplice Descrittore

```
class Desc(object):

    def __get__(self, obj, objtype):
        print 'Invocation!'
        print 'Returning x+10'
        return obj.x+10

class A(object):

    def __init__(self, x):
        self.x = x

    plus_ten = Desc()
```

*Raymond Hettinger*

```

class Desc(object):

    def __get__(self, obj, objtype):
        print 'Invocation!'
        print 'Returning x+10'
        return obj.x+10

    >>> a = A(5)

class A(object):

    def __init__(self, x):
        self.x = x

    plus_ten = Desc()

    >>> a.x
    5

    >>> a.plus_ten
    Invocation!
    Returning x+10
    15

```

*Raymond Hettinger*

# Funzioni e Metodi

- Le caratteristiche object oriented di Python sono costruite su di un ambiente basato su funzioni
- Using descrittori nonoverriding, i due ambienti si integrano in modo trasparente

```
>>> class A(object):
    def f(self, str):
        print str

>>> print A.__dict__
{'__dict__': <attribute '__dict__' of 'A' objects>, '__module__':
 '__main__', '__weakref__': <attribute '__weakref__' of 'A' objects>,
 '__doc__': None, 'f': <function f at 0x10052b488>}

>>> A.__dict__['f']
>>> <function f at 0x10052b488>

>>> A.f
<unbound method A.f>

>>> a = A()
>>> a.f
<bound method A.f of <__main__.A object at 0x100525f90>
```

```
>>> hasattr(A.__dict__['f'], '__get__')
True

>>> A.__dict__['f'].__get__(None,A)
<unbound method A.f>

>>> A.__dict__['f'].__get__(a,A)
<bound method A.f of <__main__.A object at 0x100525f90>>
```

```
>>> A.f.im_
A.f.im_class    A.f.im_func    A.f.im_self

>>> A.f.im_class
<class '__main__.A'>
>>> A
<class '__main__.A'>

>>> A.f.im_func
<function f at 0x10052b488>
>>> A.__dict__['f']
<function f at 0x10052b488>

>>> A.f.im_self
None
```

```
>>> a.f.im_
a.f.im_class    a.f.im_func    a.f.im_self

>>> a.f.im_class
<class '__main__.A'>
>>> A
<class '__main__.A'>

>>> a.f.im_func
<function f at 0x10052b488>
>>> A.__dict__['f']
<function f at 0x10052b488>

>>> a.f.im_self
<__main__.A object at 0x100525f90>
>>> a
<__main__.A object at 0x100525f90>
```

- In python le funzioni sono dei descrittori
  - nonoverriding descriptors
- Funzioni e metodi sono definiti con lo stesso statement “def”
- Ogni Classe ha un dizionario in cui i metodi sono mantenuti come oggetti “function”
- L’interprete Python trasforma funzioni in metodi quando questi vengono invocati

# Ricapitoliamo

- I descrittori sono oggetti che definiscono \_\_get\_\_, \_\_set\_\_, \_\_del\_\_
- Vengono invocati solo attraverso quello che viene definito “dotted access”: A.f oppure a.f
- Devono essere mantenuti nel dizionario della classe, non sul dizionario dell’istanza

*!! Non vengono invocati attraverso un accesso al dizionario: A. dict ['f'] !!*

# \_\_getattribute\_\_

Perche' A.x e a.x invocano il descrittore(cioe' invocano il metodo \_\_get\_\_ sul descrittore), ma A.\_\_dict\_\_['f'] semplicemente ritorna l'oggetto descrittore ???

```
>>> A.__dict__['f']
>>> <function f at 0x10052b488>
```

```
>>> A.f
<unbound method A.f>
```

```
>>> a = A()
>>> a.f
<bound method A.f of <__main__.A object at 0x100525f90>
```

# \_\_getattribute\_\_

IL “dotted access” e’ diverso dall’accesso ad un dizionario

L’invocazione A.f viene eseguita dall’interprete come:

```
type.__getattribute__(A, 'f')
```

L’invocazione a.f viene eseguita dall’interprete come:

```
object.__getattribute__(a, 'f')
```

# Descrittori: Conclusioni

- I descrittori vengono invocati attraverso:
  - `type.__getattribute__`
  - `object.__getattribute__`
- La `__getattribute__`:
  - esegue la lookup chain per l'attributo
  - controlla se il risultato e' un descrittore
  - se e' un descrittore invoca il `__get__` sul descrittore restituendo il risultato(bound method, unbound method, ecc), altrimenti restituisce l'attributo
- Ogni classe puo' sovrascrivere il `__getattribute__`

# Metaclassi

## Cosa succede quando l'interprete Python incontra una definizione di classe??

- Legge il contenuto come fosse qualsiasi altro codice
- Crea un nuovo namespace per la classe (dizionario) ed esegue il codice al suo interno (una defizione di classe generalmente contiene metodi, altre classi, variabili, ecc)
- Terminata l'esecuzione del codice, il risultato (class object) viene inserite nel namespace dove la classe e' stata definita(generalmente il modulo che ne contiene la definizione)

Internamente la costruzione di un “class object”  
e’ affidata all’oggetto built-in *type*

Il costruttore di *type* accetta tre argomenti:

- name - il nome della classe
- bases - una tupla di classi da cui ereditare
- attrs - un dizionario contenente definizioni di metodi, variabili, altre classi, ecc

```
>>> ClassTest = type('ClassTest', (object,), {'x': 'foo'})  
>>> ClassTest  
<class '__main__.ClassTest'>  
>>> ClassTest.x  
'foo'  
  
>>> ClassTest(object):  
>>>     x = 'foo'
```

Python permette di definire le proprie metaclassi  
(generalmente subclassando *type*)!!

# Come python crea internamente istanze di una classe??

```
a = A('foo')
```

- La creazione di una istanza avviene attraverso due operazioni separate:
  - invocazione del metodo `__new__`
  - invocazione del metodo `__init__`
- Il metodo `__new__` crea l'istanza:
- Il metodo `__init__` la inizializza

# Singleton

```
class Singleton(object):

    _instance = None

    def __new__(cls, *args, **kwargs):
        if not cls._instance:
            cls._instance = super(Singleton, cls).__new__(
                cls, *args, **kwargs)
        return cls._instance
```

Non e' il modo migliore per implementare il design pattern Singleton in Python!!

## Django: un buon esempio di uso di metaclassi

- Un framework per il web scritto in Python
- django.form, package dedicato alla generazione di form
- django.form e' stato implementato seguendo i principi di quella che viene chiamata "sintassi dichiarativa"
  - principio secondo cui la sintassi deve descrivere cosa stiamo cercando di calcolare invece di come lo stiamo calcolando

# Django: example

```
from django import forms

class ContactForm(forms.Form):
    subject = forms.CharField(max_length=100)
    message = forms.CharField()
    sender = forms.EmailField()
    cc_myself = forms.BooleanField(required=False)
```

Se questa classe ereditasse da *object* e non da  
*forms.Form* non avrebbe molto senso

```

class Form(BaseForm):
    __metaclass__ = DeclarativeFieldsMetaclass

class DeclarativeFieldsMetaclass(type):
    def __new__(cls, name, bases, attrs):
        attrs['base_fields'] = get_declared_fields(bases, attrs)
        new_class = super(DeclarativeFieldsMetaclass,
                          cls).__new__(cls, name, bases, attrs)
        ...
        return new_class

def get_declared_fields(bases, attrs, with_base_fields=True):
    fields = [(field_name, attrs.pop(field_name)) for field_name, obj in attrs.items() if
               isinstance(obj, Field)]
    ...
    return SortedDict(fields)

class BaseForm(StrAndUnicode):
    def __init__(self, *args, **kwargs):
        ...
        self.fields = deepcopy(self.base_fields)

    def __getitem__(self, name):
        """Returns a BoundField with the given name."""
        try:
            field = self.fields[name]
        except KeyError:
            raise KeyError('Key %r not found in Form' % name)
        return BoundField(self, field, name)

```

## Come si dichiara una metaclasses?

- Ereditata da object(type) o da una delle sue superclassi se non definita
- Definita nella classe
- Definita come variabile globale nel modulo

## Riferimenti

- Raymond Hettinger
  - <http://users.rcn.com/python/download/Descriptor.htm>
  - [www.pycon.it/media/stuff/slides/descriptor-tutorial.ppt](http://www.pycon.it/media/stuff/slides/descriptor-tutorial.ppt)
- Python in a Nutshell, Alex Martelli
- Codice di Django

Questa presentazione e' distribuita con licenza

GNU Free Documentation Licence  
<http://www.gnu.org/copyleft/fdl.html>