# Tema 2 Orientación a Objetos

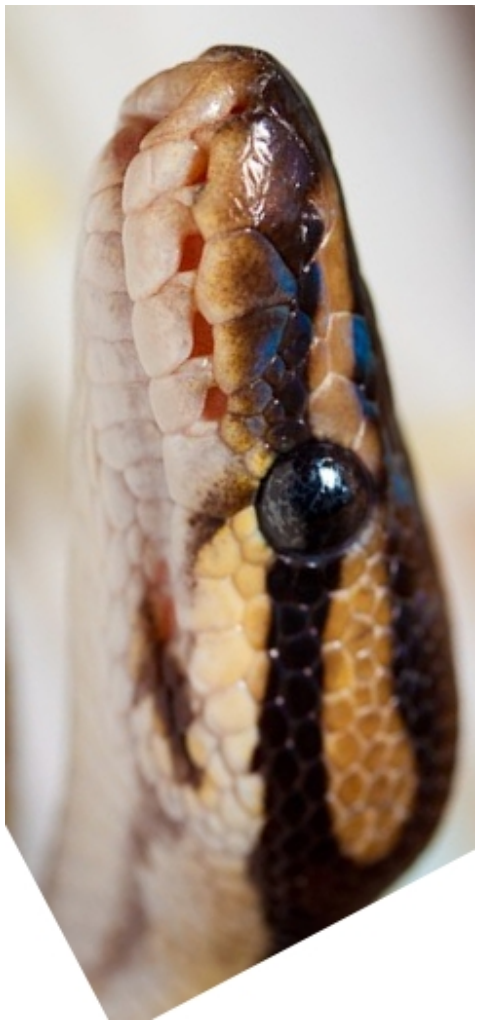
## Curso de Python Avanzado

Juan Pedro Bolívar Puente

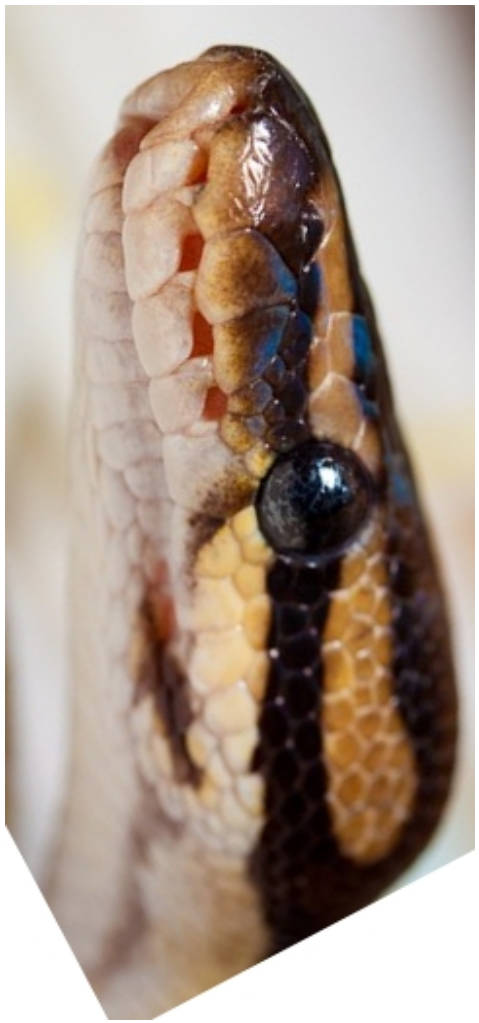
Instituto de Astrofísica de Andalucía

Mayo de 2011

# Índice



# Índice



# Objetos

## Definición clásica

**Objeto** = Estado (Atributos)
+ Comportamiento (Funciones)

Como las funciones son valores normales, simplificamos...

# **Objeto** = Conjunto de Atributos

Pensad en un diccionario al que se accede mediante el operador '.'

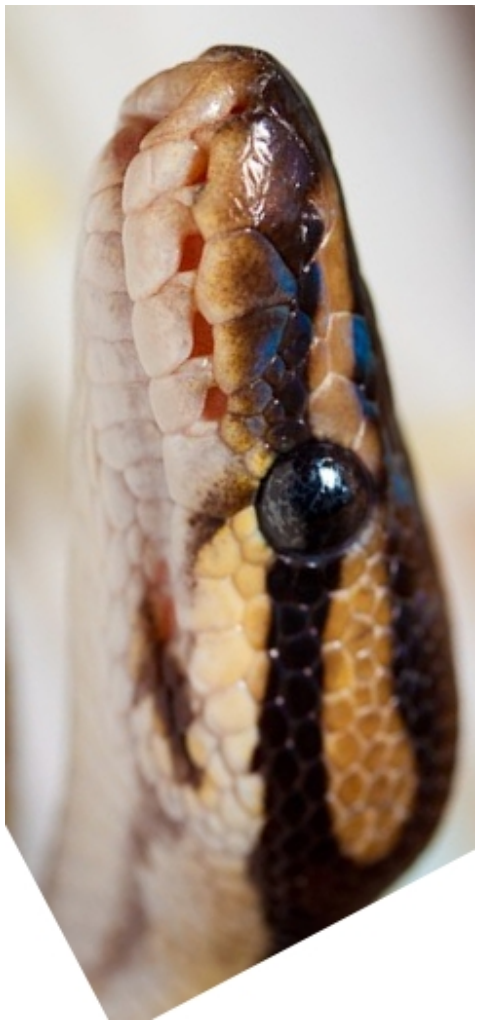
# Objetos en Python

## **Todo** es un objeto

1. Valores simples
2. Clases
3. Funciones
4. Módulos

```
print (123).__class__
print zip.__class__
print list.__class__
import os; print os.__class__
```

# Índice



# Clases en Python

## **Todo** objeto tiene clase

**Clase** = Arquetipo del objeto.

### Sintáxis

```
class Clase (Base1, Base2, ...):
    <sentencia1>
    ...
```

Las sentencias que definen nombres (`def`, `class`, `=`) los instalan en la clase.

# Instanciación

**Clase** = función

Ejecutar clase = **Instanciación**

- Produce un nuevo valor.
- El objeto tiene los mismos atributos que la clase salvo el *operador ()* (...aprox...)
- Inicializa el objeto con el método `__init__`
- Le pasa objeto + parámetros a la clase.

# Paréntesis terminológico

## Función: Función libre.

```
def funcion (...): ...
```

## Método: Función "asociada" a un objeto.

```
class UnaClase (object):
    def metodo (self, ...): ...
```

- Primer parametro es la instancia.
- `self` por convención.

# Paréntesis metodológico

## Python $\geq$ 2.2 $\Rightarrow$ "New style classes"

- Muchas cosas sólo funcionan con ellas.

  `super`, *slots*, propiedades, ...
- Heredar **siempre** de `object`.

**Mal**

```
class Base:
    pass
class Deriv (Base):
    pass
```

**Bien**

```
class Base (object):
    pass
class Deriv (Base):
    pass
```

# Ejemplo ...

## Definiendo una clase

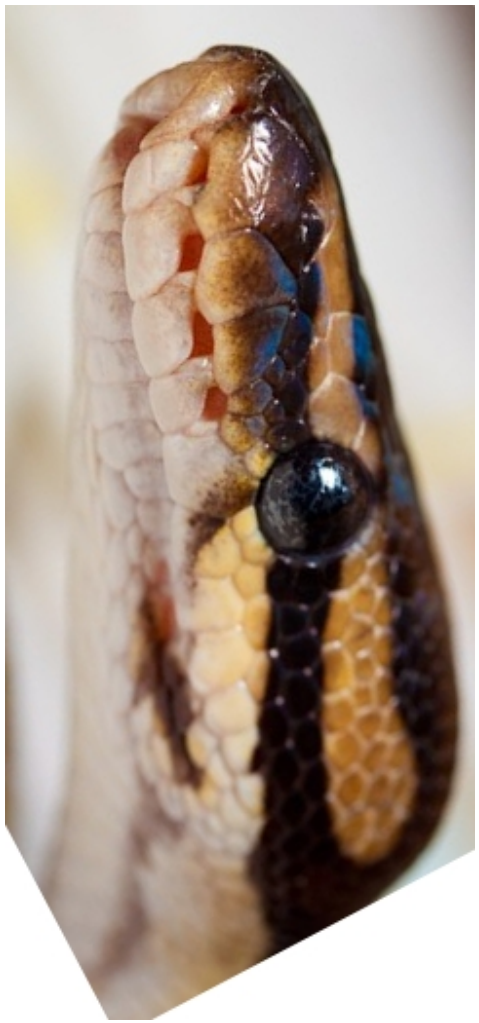
```
class Foo (object):
    un_atributo = 3

    def __init__ (self, otro = None):
        self.otro_atributo = otro

    def metodo (self):
        print "Metodo de: ", self
```

# Ejemplo ...

## Usando la clase ...

```
obj = Foo ()
print obj
print obj.un_atributo
print obj.otro_atributo

obj = Foo (10)
print obj.otro_atributo

obj.un_atributo = 15
print obj.un_atributo
print Foo.un_atributo
obj.metodo ()
```

# Índice



# Atributos

Los atributos pueden añadirse, borrarse, modificarse,
**en cualquier momento**

```
obj.nuevo_atributo = 2
print obj.nuevo_atributo

del obj.nuevo_atributo
print hasattr (obj, 'nuevo_atributo')

obj.metodo = 3
print obj.metodo ()
```

# Acceso atributos

## Funciones de acceso sirven para ...

- Saltarnos las reglas de identificadores.
- Generar dinamicamente nombre atributos.

getattr (obj, atributo)
: Acceder al atributo en `obj`.

setattr (obj, atributo, valor)
: Establecer atributo en `obj`.

delattr (obj, atributo)
: Borrar atributo en `obj`

# Ejemplo ...

## Operaciones de acceso

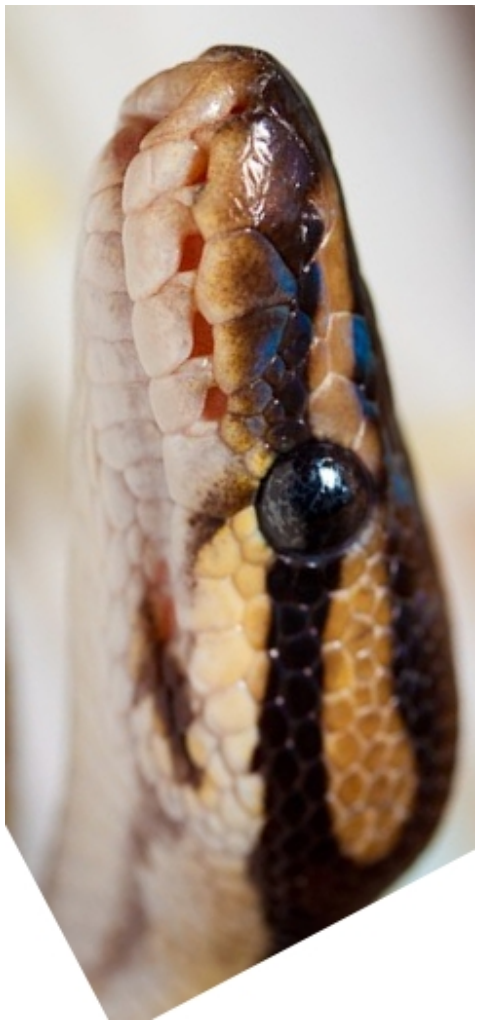
```
class Bar (object):
    def func_one (self):
        print "Primera␣funcion."

    def func_two (self):
        print "Segunda␣funcion."
```

# Ejemplo ...

## Operaciones de acceso

```
obj = Bar ()

from random import choice
cadena = choice (['one', 'two'])
getattr (obj, 'func_' + cadena ) ()

setattr (obj, 'una cosa!', 123)
print getattr (obj, 'una cosa!')
```

# Índice



# Encapsulación

## Idea

**Interfaz $\neq$ Implementación**

No exponer atributos (mantenibilidad)

Mientras tanto en Python...

- Todo es público
- Por convención, lo privado empieza por '_'

  Ejemplo: `self._atributo`
- Atributos intercambiables por metodos

# Encapsulación ...

## ¡Python **no** es Java!

### No hacer "setters" y "getters"

```
class BadExample (object):
    _data = None
    def set_data (self, data):
        self._data = data
    def get_data (self):
        return self._data
```

# Propiedades

En Python, un atributo puede ser público si ha de ser accederse desde fuera

## Ejemplo version 1

```
class GoodExample (object):
    data = None

obj = GoodExample ()
obj.data = 3
print obj.data
```

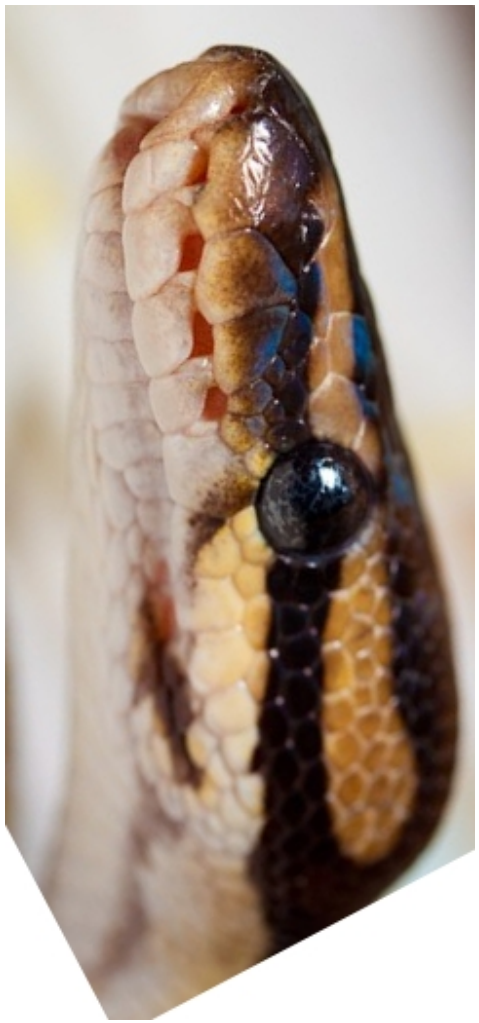
# Propiedades ...

Al cambiar los requisitos, usamos:
`property([fget[, fset[, fdel[, doc]]]])`

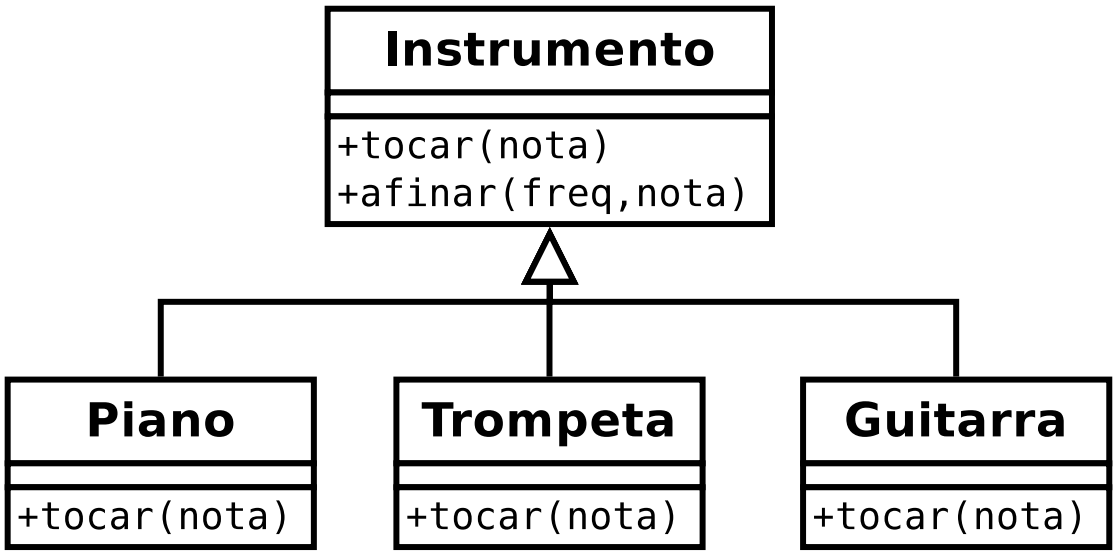
## Ejemplo version 2

```
class GoodExample (object):
    _data = None
    def _set_data (self, value):
        print "Setting data"
        self._data = value
    def _get_data (self):
        print "Getting data"
        return self._data
    data = property (_get_data, _set_data)
```

# Índice



# Herencia




Una clase puede ...

- Heredar los atributos de su padre.
- Sobrecargar (redefinir) los atributos de su padre.

# Ejemplo ...

## Terminología

- Clase base → clase derivada
- Super-clase → sub-clase

```
class Base (object):
    def method (self):
        print "base.method␣()"
    def other (self):
        print "base.other␣()"

class Deriv (Base):
    def method (self):
        print "deriv.method␣()"
```

# Ejemplo ...

## Substituibilidad ...

```
def invoca_metodo (obj):
    obj.metodo ()

obj = Base ()
obj.method ()
obj.other ()
invoca_metodo (obj)

obj = Deriv ()
obj.method ()
obj.other ()
invoca_metodo (obj)
```

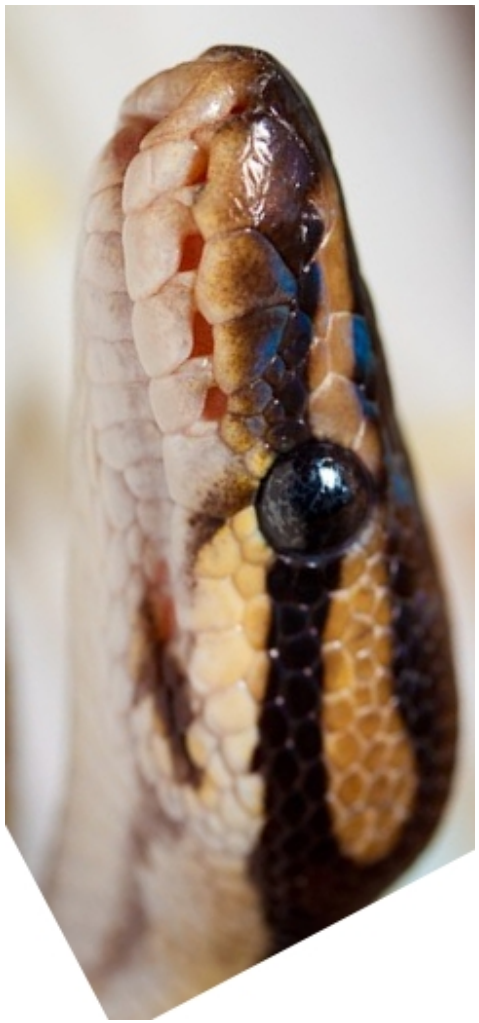
# Recordemos: *duck typing*

¡No hace falta herencia para tener substituibilidad!

**invoca_metodo depende de method no de Base**

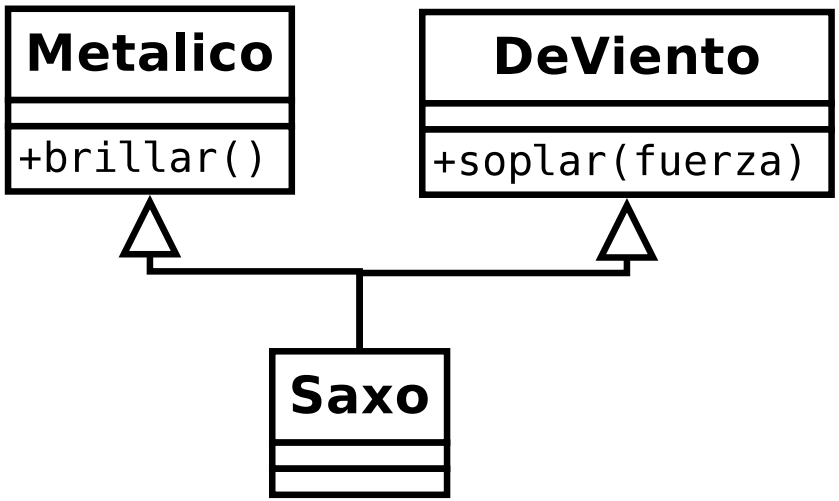
```
class Otra (object):
    def method (self):
        print "otra.method␣()"

obj = Otra ()
invoca_metodo (obj)
```

# Índice



# Herencia múltiple

## Podemos heredar de varios padres




### Sobrecarga

- Abajo → arriba
- Izquierda → derecha

# Ejemplo ...

## Los padres ...

```
class A (object):
    def method (self):
        print "A.method"

class B (object):
    def method (self):
        print "B.method"
```

# Ejemplo ...

## Los padres ...

```
class Mix1 (A, B):
    pass

class Mix2 (A, B):
    def method (self):
        print "Mix.method"

Mix1 ().method ()
Mix2 ().method ()
```

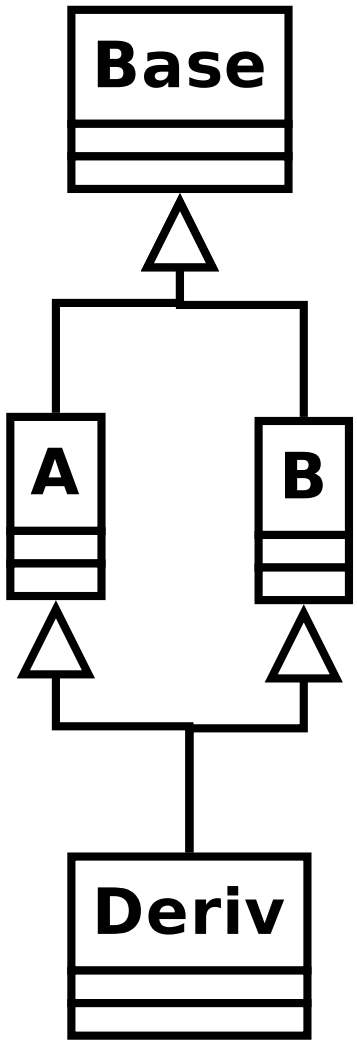
# Constructores y herencia

Si sobrecargamos el constructor hay que llamar al constructor del padre

**¡Pero no así!**

```
class Deriv (Base):
    def __init__ (self):
        Base.__init__ (self)
        ...
```

# El problema del diamante ...




```
class Base (object):
    def __init__ (self):
        self.x = 0
class A (Base):
    def __init__ (self):
        Base.__init__ (self)
        self.x = 7
class B (Base): pass
class Deriv (A, B):
    def __init__ (self):
        A.__init__ (self)
        B.__init__ (self)
print Deriv ().x
```

# ¡Solución: super!

super (clase, objeto) devuelve un *proxy* de `objeto` con la clase "siguiente" más próxima.

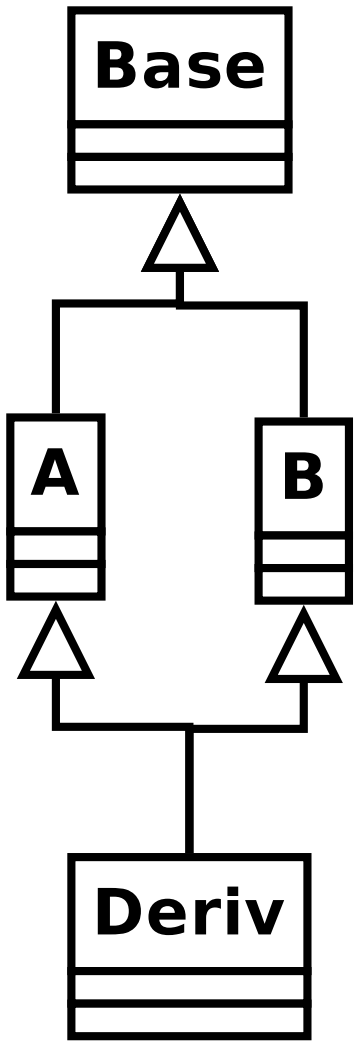
## Reglas clase "siguiente" (algoritmo C3)

- A está más arriba que B $\Rightarrow$ A sigue a B
- A está más a la derecha que B $\Rightarrow$ A sigue a B

## Ejemplo del diamante

```
>>> print Deriv.__mro__
(Deriv, A, B, Base, object)
```

# El problema de la linearización dinámica ...




## ¡Casi pero no!

```
class Base (object):
  def __init__ (self):
    self.x = 0
class A (Base):
  def __init__ (self):
    super (A, self).__init__ ()
    self.x = 7
class B (Base): pass
class Deriv (A, B):
  def __init__ (self):
    super(Deriv,self).__init__()
print Deriv ().x
```

# El problema de la linearización dinámica ...

## ¡No sabemos qué tipo tiene super!

```
class Base (object):
    def __init__ (self):
        print "Base.__init__"
class OtraBase (object):
    def __init__ (self):
        print "OtraBase.__init__"
class Deriv (Base, OtraBase):
    def __init__ (self):
        print "Deriv.__init__"
        super (Deriv, self).__init__ ()
Deriv ()
```

# El problema de la linearización dinámica ...

## Llamamos a super aunque sólo heredemos de object

```
class Base (object):
    def __init__ (self):
        print "Base.__init__"
        super (Base, self).__init__ ()
class OtraBase (object):
    def __init__ (self):
        print "OtraBase.__init__"
        super (OtraBase, self).__init__ ()
class Deriv (Base, OtraBase):
    def __init__ (self):
        print "Deriv.__init__"
        super (Deriv, self).__init__ ()
```

# ¿Qué pasa con los parámetros?

```
class Base (object):
    def __init__ (self, param):
        super (Base, self).__init__ ()
        self.param = param

class Deriv (Base):
    def __init__ (self):
        super(Deriv, self).__init__( ?!?!? )
```

- Base necesita un parametro.
- Deriv lo sabe pero no sabe quién es super.

# ¿Qué pasa con los parámetros?

- Usamos **siempre** parámetros por clave.
- **Siempre** redireccionamos las sobras.

```
class Base (object):
    def __init__ (self, param=None, *a, **k):
        super (Base, self).__init__ (*a, **k)
        assert param is not None
        self.param = param

class Deriv (Base):
    def __init__ (self, *a, **k):
        super(Deriv, self).__init__ (
            param=3, *a, **k)
```

# Moraleja ...

1. Decidir las restricciones de diseño.
   ¿Herencia múltiple? ¿Qué pueden sobrecargar las subclases?
2. Definir convenciones en consecuencia.

## Una convención universal ...

1. `object` siempre en la base.
2. Llamar siempre al constructor de `super`
3. Usar siempre parámetros por clave en el constructor.
4. Reenviar siempre los parámetros sobrantes a `super`.

# Es decir...

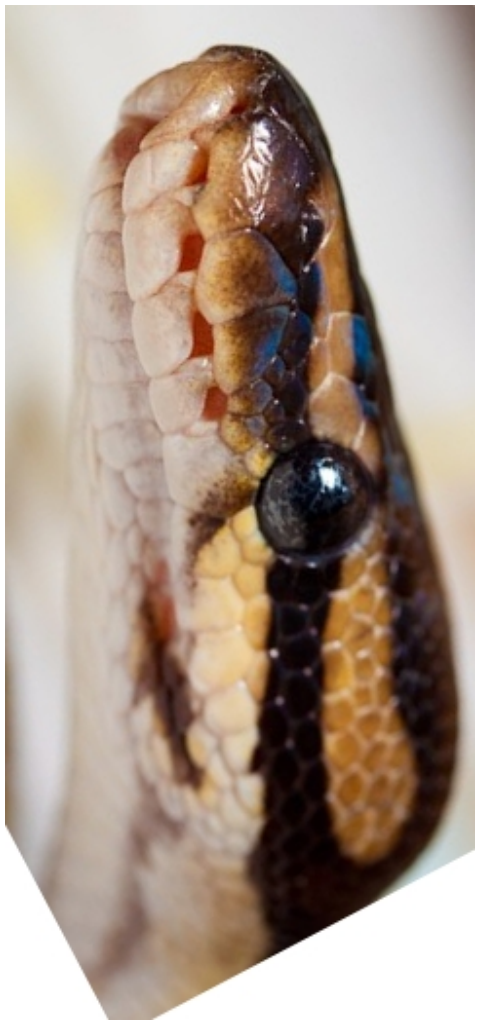
Una convención universal ...

1. `object` siempre en la base.
2. Llamar siempre al constructor de `super`
3. Usar siempre parámetros por clave en el constructor.
4. Reenviar siempre los parámetros sobrantes a `super`.

**¡Memorizad esto a fuego!**

```
def UnaClase (...):
  def __init__(self,param1=None,...,*a,**k):
    super(UnaClase,self).__init__(...,*a,**k)
        ...
```

# Índice



# Objetos ligeros

¡Los objetos son pesados!

1. La mayor parte del peso se va en el diccionario
2. Lo evitamos prefijando los atributos ...

Atributo prefijado = ¡Slots!

# Usando slots

Definir `__slots__` con una secuencia de cadenas

## Objeto pesado ...

```
class Point (object):

    def __init__ (self, x=0, y=0, ...
        super(Point,self).__init__...
        self.x = x
        self.y = y
```

# Usando slots

Definir `__slots__` con una secuencia de cadenas

## Objeto ligero ...

```
class Point (object):

    __slots__ = 'x', 'y'

    def __init__ (self, x=0, y=0, ...
        super(Point,self).__init__...
        self.x = x
        self.y = y
```

# Limitaciones de los `slots`

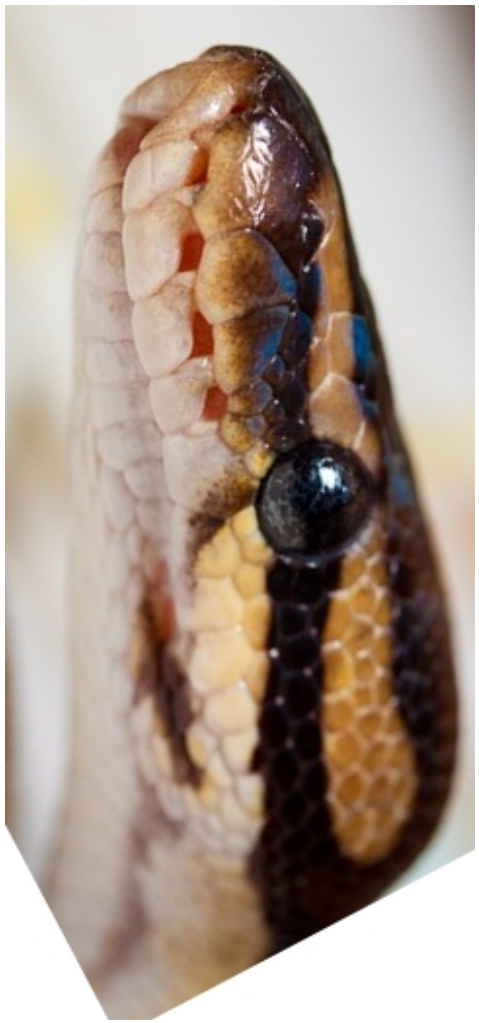
## Acceder a otro elemento produce error

$\Rightarrow$

- No se pueden heredar clases con y sin *slots*.
- Usar cuando número de instancias $> 10^6$

```
obj = Point (1, 2)
print obj.x, obj.y
try:
    obj.z = 3
except AttributeError, e:
    print repr (e)
```

# Índice



# Métodos especiales

## Atributos con forma `__metodo__`

# Métodos especiales

Ya hemos visto `__init__`, `__slots__`

# Representación de objetos

`__str__(self)`

Da una representación legible por humanos

Se invoca con `str(...)`

`__repr__(self)`

Da una representación sintácticamente Python

Si no es posible: `'<...>'`

Se invoca con `repr(...)`

`print` primero intenta `str` luego `repr`

# Aritmética

`__add__ (self, other)`
: Devuelve un nuevo objeto
: Se invoca con `a + b`

`__radd__ (self, other)`
: Devuelve un nuevo objeto
: Se invoca con `a + b` cuando *a* no tiene suma

`__iadd__ (self, other)`
: Guarda el resultado en `self`
: Se invoca con `a += b`

Hay muchos otros ... `__sub__` `__mul__` `__floordiv__` `__div__` `__pow__` `__mod__` `__divmod__` `__and__` `__or__` `__xor__`

```
import os
class MyPath (object):
    def __init__ (self, sep=os.sep, *a, **k):
        super (MyPath, self).__init__ (**k)
        self.sep = sep
        self._str = self.sep.join (a)
    def __rdiv__ (self, other):
        return MyPath (other, self._str)
    def __div__ (self, other):
        return MyPath (self._str, other)
    def __idiv__ (self, other):
        self._str = self.sep.join (
            (self._str, str (other)))
        return self
    def __str__ (self):
        return self._str
```

# Ejemplo ...

```
p = MyPath ('raskolnikov')
print p

p = '' / 'home' / p
print p

p /= 'dev'
print p

p = p / 'curso' / '03-objetos'
print p
```

# Comparadores

## Comparadores "ricos"

- `__gt__` (>=) `__ge__` (>) `__eq__` (==) `__ne__` (!=) `__lt__` (<) `__le__` (<=)
- Son operadores binarios, reciben `(self, other)`

### Comparador genérico

`__cmp__ (self, other)` es llamado por defecto:

- 0 sii *self* == *other*
- $< 0$ sii *self* $<$ *other*
- $> 0$ sii *self* $>$ *other*

# Estructuras de datos

`__getitem__(self, key)`
: Se llama con `obj[...]`

`__setitem__(self, key, val)`
: Se llama con `obj[...]=val`

`__delitem__(self, key)`
: Se llama con `del obj[...]`

`__contains__(self, item)`
: Se llama con `item in obj[...]`

`__len__(self, item)`
: Se llama con `len (obj)`

# Acceso atributos

## Permiten sobrecargar el operador '.'

`__getattr__(self, key)`
: Se llama con `obj.key`
: Sii `key` $\notin$ `obj.__dict__`

`__getattribute__(self, key)`
: Se llama con `obj.key` (casi) siempre

`__setattr__(self, key, val)`
: Se llama con `obj.key = val`

`__delattr__(self, key)`
: Se llama con `del obj.key`

# Ejemplo ...

## *Proxy* genérico

```
class Proxy (object):
    def __init__(self, proxied=None, *a,**k):
        super(Proxy, self).__init__(*a,**k)
        self.proxied = proxied
    def __getattr__ (self, key):
        return getattr (self.proxied, key)

a = Proxy (list ())
a.append (1)
print a
print a.proxied
```

# Functores

**Functor** = Objeto que se comporta como una función

Sobrecarga el operador

```
__call__ (self, parametros como funcion ...)
```

Podemos comprobar si un objeto es un *ejecutable* con

```
callable (obj)
```

# Un ejemplo

```
class Contador (object):
    def __init__(self, inic=0, *a,**k):
        super(Contador, self).__init__(*a,**k)
        self.cuenta = inic
    def __call__ (self):
        actual = self.cuenta
        self.cuenta += 1
        return actual

cnt = Contador ()
print cnt ()
print cnt ()
print callable (cnt)
```

# Recursos adicionales

Introduction to Object-Oriented Programming
Timthy Budd
Addison Wesley, 3rd Edition, October 2001

Python's Super is nifty, but you can't use it
James Y. Knight
http://fuhm.net/super-harmful/

The Python 2.3 Method Resolution Order
Michele Simionato
http://www.python.org/download/releases/2.3/mro/

# ¿Preguntas?

# Muchas gracias por su atención.