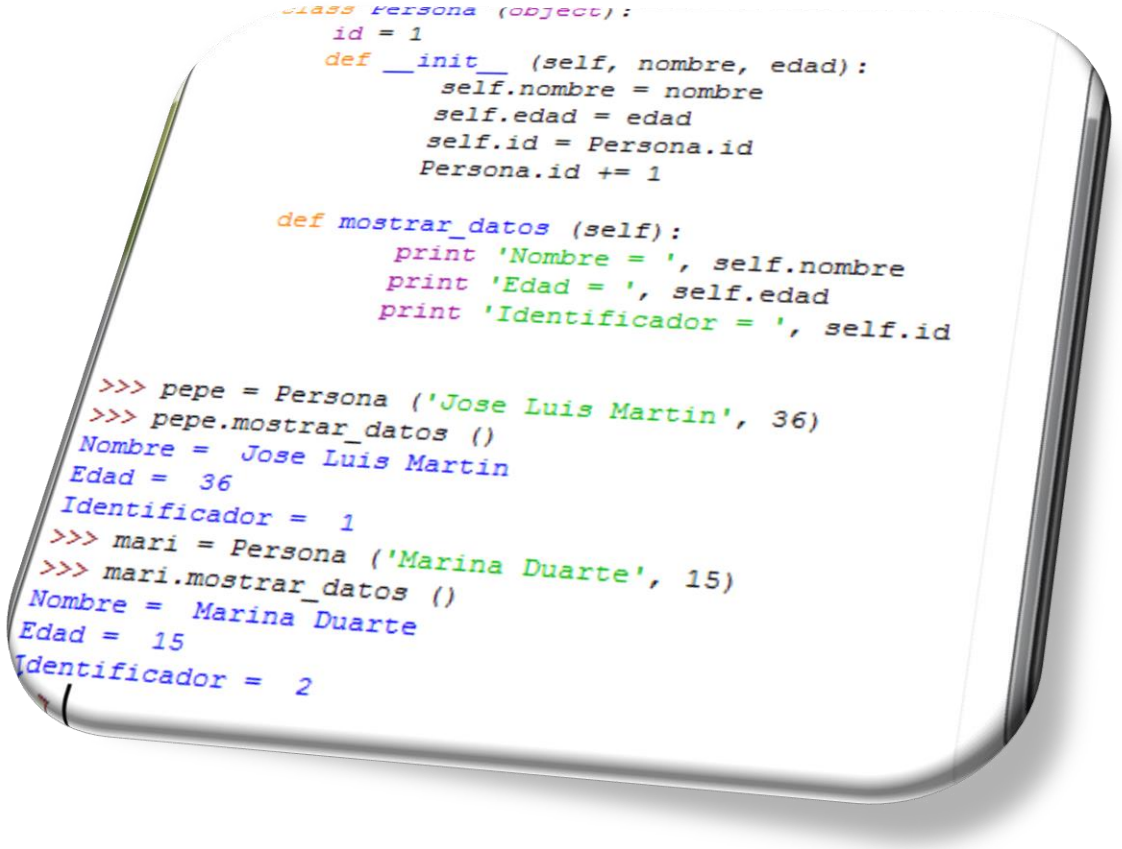


Curso de Python Inicial

Clases

Curso organizado por el Gabinete de Formación del CSIC

Abril 2013 **Granada**

César Husillos & Víctor Terrón

Contenidos

1. Paradigmas de la Programación
2. Programación Orientada a objetos
3. Clases
4. Objetos
5. Encapsulación
6. Herencia
7. Funciones para clases y objetos
8. Métodos internos de la clase (*builtin-methods*)

Paradigmas de Programación

*Un **paradigma** es el resultado de un proceso social en el cual un grupo de personas desarrolla nuevas ideas y crea principios y prácticas alrededor de estas ideas.*

Resumiendo: un paradigma es una metodología de trabajo.

- En programación, se trata de un enfoque concreto de desarrollar y estructurar el desarrollo de programas.
- Hasta el momento, el trabajo que han realizado ha obedecido al paradigma de programación imperativa.
- En esta presentación, introduciremos la programación imperativa (aunque a estas alturas del curso no le sonará a algo nuevo) y desarrollaremos el *paradigma orientado a objeto* a través de Python.

Paradigmas de programación: Imperativo

Paradigma imperativo

- Consiste en una secuencia de instrucciones que el ordenador debe ejecutar.
- Los elementos más importantes en esta forma de programar son:
  1. *Variables*, zonas de memoria donde guardamos información.
  2. *Tipos de datos*, son los valores que se pueden almacenar.
  3. *Expresiones*, corresponde a operaciones entre variables (del mismo o distinto tipo)
  4. *Estructuras de control*, que permiten ejecutar un conjunto de instrucciones varias veces, ejecutar una parte del código u otra en función de que se cumpla una condición o abortar la ejecución del programa.

Paradigmas de programación: Funcional

Pese a que trabajamos con funciones, el modelo desarrollado hasta ahora no verifica todos los requisitos del paradigma de **programación funcional** ya que, en nuestro caso existe el concepto de variable, que no se da en programación funcional.

Sí que verificamos que:

- Nuestros programas pueden hacer uso de funciones que realizan su tarea como si de una caja negra se tratase: metemos parámetros de entrada y obtenemos algo a la salida.
- Nuestras funciones pueden servir como parámetros de entrada para otras funciones.

Paradigmas de programación: Orientado a Objetos

- Es el más popular en la actualidad.
- Se fundamenta en la “fusión” de datos y funciones que operan sobre esos datos dentro de un nuevo tipo de dato.
- Al nuevo tipo de dato se le llama CLASE.
- A cada variable de una clase se le llama OBJETO.

Paradigmas de programación: Orientado a Objetos (I)

Propiedades del paradigma orientado a objetos

1. **Encapsulamiento**
   - Significa que los datos pertenecen a un objeto (espacio de nombres del objeto).
   - Podemos ir más allá y ocultar los datos de un objeto a cualquier otro objeto o código que trate de hacer uso de ellos. Serían sólo accesibles al propio objeto y, en algunos casos, a objetos de sus clases descendientes.
2. **Herencia**
   - Es la propiedad de crear nuevos datos a partir de los ya existentes (progenitores). Heredamos sus atributos y métodos. Podemos sobrescribirlos para adaptarlos a la clase heredada (clase hija).
3. **Polimorfismo**
   - Hace referencia a la llamada de una función de una clase por parte de un objeto. Cuando se produce, se ejecuta la correspondiente al tipo del objeto que lo llama, no al de sus progenitores.

Clases

Una clase es un nuevo tipo de dato. Contiene :

- otros datos (que pueden ser de cualquier tipo)
- Funciones, que operan sobre esos datos.

Se declaran en el código de la siguiente forma:

```
class Nueva_clase (object):
    código_de_la_clase
```

Donde el `código_de_la_clase` incluye la declaración de variables y funciones.

- `object` es la clase base para cualquier objeto creado en Python.

Clases (I): Atributos

- Las variables incluidas en una clase se denominan ATRIBUTOS.
- Existen múltiples formas de crear atributos en una clase. La más simple:

```
class Nueva_clase (object):
    atributo1 = valor1
    atributo2 = valor 2
    …
```

Clases (II): Métodos

Las clases pueden contener funciones. A éstas se les denomina MÉTODOS.

La forma de crearlos en Python es en la declaración de la clase

```
class Nueva_clase(object):
      def metodo1(self,[parametros]):
            codigo_metodo1
```

donde `self`

- Es el primer parámetro de cualquier método.
- Hace referencia a la propia clase (y a su contenido).
- Nunca se pasa como parámetro cuando se llama a un método. Es un *parámetro implícito*.

La llamada a este método en el código se haría tras la creación de un objeto. La sintaxis:

```
Objeto.metodo1([parametros])
```

Clases (III): Creación de objetos

Una vez definida la clase, crear un objeto es tarea sencilla.

Basta con ejecutar la instrucción de asignación

`objeto = Nombre_clase ()`

donde `objeto` será una nueva variable del tipo `Nombre_clase`.

```
>>> class Saludo (object):
        nombre = 'Jose Luis'
        apellidos = 'Montero Fuentes'
        def saludar (self):
                print 'Hola. Soy %s %s' % (self.nombre, self.apellidos)

>>> objeto = Saludo ()
>>> objeto.saludar ()
Hola. Soy Jose Luis Montero Fuentes
>>>
```

Creación de objeto (instanciación)

Ejecución de método

Clases (IV): Inicialización de atributos

La clase anterior es un buen ejemplo para empezar, pero no sirve de mucho desde un punto de vista práctico.

Vamos a complicarlo un poco. Crearemos una clase en la que, al declarar un objeto, inicialicemos sus atributos.

¿Cómo?

Mediante el método implícito de la clase `__init__`

Clases (IV): Inicialización de atributos

`__init__`

- es la primera función que se ejecuta al crear un objeto, y lo hace de forma automática. Para los programadores de C++ o JAVA, éste sería su *constructor*.
- Podemos crear allí los atributos de la clase y pasarle los valores con los que inicializarlos en el momento de crear cada objeto.

Clases (IV): Inicialización de atributos

```
>>> class Calculadora (object):
        def __init__ (self, operando1, operando2):
                self.op1 = operando1
                self.op2 = operando2
        def sumar (self):
                return self.op1 + self.op2
        def restar (self):
                return self.op1 - self.op2
        def multiplicar (self):
                return self.op1 * self.op2
        def dividir (self):
                return self.op1 / self.op2


>>> o = Calculadora (10, 2)
>>> o.sumar ()
12
>>> o.restar ()
8
>>> o.multiplicar ()
20
>>> o.dividir ()
5
>>>
```

Constructor
Recibe por parámetros a operando1 y operando2, con los que se inicializa a los atributos op1 y op2.
ATENCIÓN al 'self.'

Creación de objeto con parametros 10 y 2

Llamadas a métodos de la clase.
Los valores con los que se trabaja son los atributos del objeto 'o'

EJERCICIOS: 1,2,3,4 Y 5

Encapsulación

"Significa que los datos pertenecen a un objeto (espacio de nombres del objeto).

Podemos ir más allá y ocultar los datos de un objeto a cualquier otro objeto o código que trate de hacer uso de ellos. Serían sólo accesibles al propio objeto y, en algunos casos, a objetos de sus clases descendientes."

- Python implementa bien el nivel de ocultación de variables, pero no es efectivo en cuanto a la protección de las variables ocultas de accesos externos.
- Según el manual de "Guía de aprendizaje de Python" (**Guido van Rossum**):

Las clases de Python no ponen una barrera absoluta entre la definición y el usuario, sino que más bien se fían de la buena educación del usuario para no "invadir la definición".

Encapsulación (II)

```
>>> class Oculta_x (object):
        def __init__ (self):
                self.__x = 0
        def mostrar_x (self):
                return self._Oculta_x__x
        def incrementa_x (self):
                self._Oculta_x__x += 1


>>> o = Oculta_x ()
>>> dir (o)
['_Oculta_x__x', '__class__', '__delattr__', '__dict__', '__doc__', '__format__', '__getattribute__
', '__hash__', '__init__', '__module__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__s
etattr__', '__sizeof__', '__str__', '__subclasshook__', '__weakref__', 'incrementa_x', 'mostrar_x']
>>> o.mostrar_x ()
0
>>> o.incrementa_x ()
>>> o.mostrar_x ()
1
```

Manipulación interna (por métodos de la clase) de atributos del objeto

El mecanismo de ocultación de Python funciona!

```
>>> o.__x

Traceback (most recent call last):
  File "<pyshell#108>", line 1, in <module>
    o.__x
AttributeError: 'Oculta_x' object has no attribute '__x'
```

Python protege la variable de modificaciones externas

```
>>> o._Oculta_x__x
1
>>> o._Oculta_x__x += 2
>>> o._Oculta_x__x
3
>>>
```

Si conocemos la existencia de la variable y la forma en que Python hace referencia a esa variable, la protección no sirve de nada

EJERCICIOS: 6

Herencia

"Es la propiedad de crear nuevos datos a partir de los ya existentes (progenitores). ***Heredamos*** *sus atributos y métodos. Podemos* ***sobrescribirlos*** *para adaptarlos a la clase heredada (clase hija)."*

- La herencia es el mecanismo de reutilización de código por excelencia en Programación Orientada a Objetos.
- Sirve para ampliar, particularizar o mejorar determinadas clases en otras nuevas. Las clases padre/madre siguen vigentes, por lo que no es necesario retocar el código que ya funcionaba.

Herencia: ¿Cómo se hace en Python?

Dada una clase Madre podemos crear otra clase Hija de la siguiente forma:

```
class Hija(Madre):
    codigo_hija
```

- El código de la hija puede sobreescribir métodos de la madre e introducir nuevos atributos, si se necesitan.
- Según se ve, toda clase que hemos creado hasta ahora es hija de la clase `object`.

Herencia + Sobrecarga de métodos

```
>>> class Madre (object):
        def __init__ (self, parametro1):
                self.par1 = parametro1
        def metodo1 (self):
                print 'metodo1 de la clase Madre.'
        def metodo2 (self):
                print 'metodo2 de la clase Madre.'


>>> dir (Madre)
['__class__', '__delattr__', '__dict__', '__doc__', '__format__', '__getattribute__', '__hash__', '
__init__', '__module__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__si
zeof__', '__str__', '__subclasshook__', '__weakref__', 'metodo1', 'metodo2']
>>> objeto_madre = Madre ('Madre1')
>>> dir (objeto_madre)
['__class__', '__delattr__', '__dict__', '__doc__', '__format__', '__getattribute__', '__hash__', '
__init__', '__module__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__si
zeof__', '__str__', '__subclasshook__', '__weakref__', 'metodo1', 'metodo2', 'par1']
>>> class Hija (Madre):
        'Clase Hija. Deriva de Madre'
        def metodo1 (self):
                'metodo1: imprime un mensaje simple'
                print 'metodo1 de la clase Hija'
```

Sobrecarga o sobrescritura de métodos heredados

```
>>> objeto_hija = Hija ()

Traceback (most recent call last):
  File "<pyshell#145>", line 1, in <module>
    objeto_hija = Hija ()
TypeError: __init__() takes exactly 2 arguments (1 given)
>>> objeto_hija = Hija (24)
>>> objeto_hija.par1
24
>>> objeto_hija.metodo1 ()
metodo1 de la clase Hija
>>> objeto_hija.metodo2 ()
metodo2 de la clase Madre.
>>>
```

La hija necesita un parametro (como la madre)

Se inicializa como lo haría la madre (usa el contructor de la madre)

Sobrecargado

No sobrecargado (simplemente heredado)

Herencia y nuevos atributos

Cuando la clase hija tiene nuevos atributos

¿Cómo los inicializamos?

Tenemos 2 posibilidades:

1. Iniciamos todos
   - Sencillo en el caso de pocos atributos.
2. Utilizamos la inicialización de la clase madre para los atributos heredados y nueva inicialización para los nuevos.
   - Implica un diseño más elaborado, pero una programación orientada a objetos más reutilizable y organizada.

Herencia y nuevos atributos (II)

En el caso 1: sobrecargamos el constructor (`__init__`) de la clase hija para redefinir toda la inicialización.

herencia.py: Bloc de notas

Archivo Edición Formato Ver Ayuda

```
#!/usr/bin/env python
#-*- coding: UTF-8 -*-

class madre (object):
        def __init__ (self, par):
                self.uno = par

class hija (madre):
        def __init__ (self, par1, par2):
                self.uno = par1
                self.dos = par2

if __name__ == '__main__':
        m = madre ('madre')
        h = hija ('hija1', 'hija2')
        print 'atributo de la madre =', m.uno
        print 'atributo uno de la hija =', h.uno
        print 'atributo dos de la hija =', h.dos
```

Administrador: C:\Windows\system32\cmd.exe

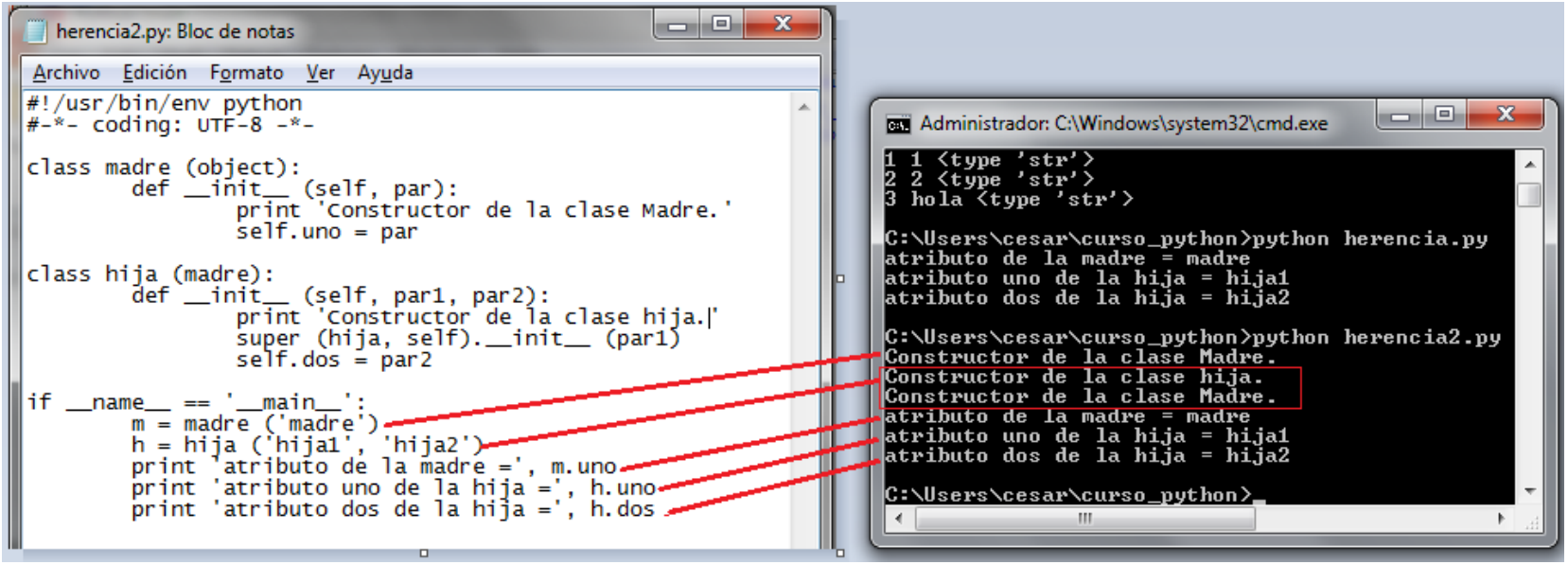
```
['parametros.py', '1', '2', 'hola']
0 parametros.py <type 'str'>
1 1 <type 'str'>
2 2 <type 'str'>
3 hola <type 'str'>

C:\Users\cesar\curso_python>python herencia.py
atributo de la madre = madre
atributo uno de la hija = hija1
atributo dos de la hija = hija2

C:\Users\cesar\curso_python>
```

Herencia y nuevos atributos (III)

En el caso 2: llamamos al constructor de la clase madre y le pasamos los parámetros necesarios. El resto se inicializa en el constructor de la hija.

- El constructor de la clase madre se llama de la siguiente forma:

```
super(clase_hija, self).__init__([parametros])
```



EJERCICIOS: 7 Y 8.

Herencia múltiple

Podemos heredar de más de una clase. Sintaxis:

```
class Clase_hija (claseMadre1, claseMadre2,…):
      codigo_Clase_hija
```

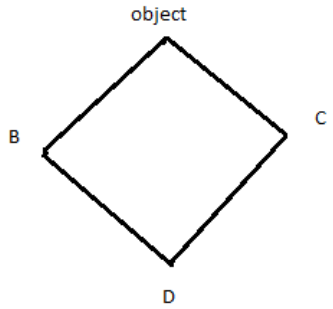
- La clase hija tendrá todos los atributos definidos en sus clases madres.

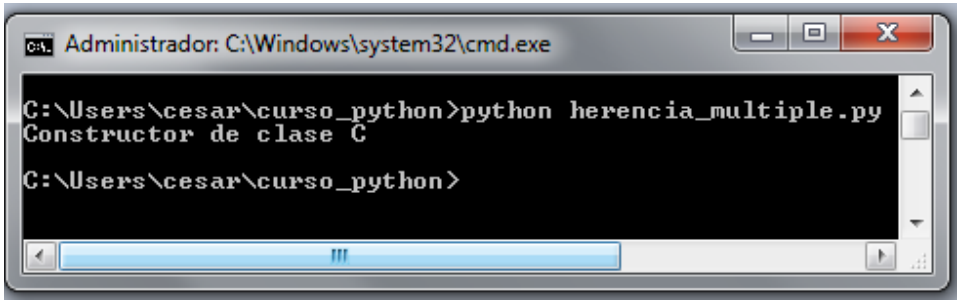
En cuanto a los métodos de la clase hija,

¿qué sucede si dos de las madres comparten un método con el mismo nombre?

Herencia múltiple (II)

Diagrama de herencia





Sin título: Bloc de notas

Archivo Edición Formato Ver Ayuda

```
#!/usr/bin/env python
#-*- coding: UTF-8 -*-

class B (object):
        pass

class C (object):
        def __init__ (self):
                print "Constructor de clase C"

class D (B, C):
        pass

if __name__ == '__main__':
        d = D ()
```

¿Por qué llama al constructor de C y no al de B?

Porque se llama al método de la clase cuya sobrecarga esté más próxima a la clase hija. En caso de igualdad, toma el de la clase madre más a la derecha en la definición.

EJERCICIOS: 9 Y 10.

Funciones OO: para Clases y objetos

Función	Descripción
issubclass (sub, sup)	Devuelve *True*, si la clase *sub* tiene como ancestro a la clase *sup*. *sup* puede ser una lista o tupla de clases.
isinstance (obj1, obj2)	Devuelve *True* si *obj1* es una instancia de *obj2*. Obj2 puede ser una lista o tupla de clases.
getattr (obj, attr [,default])	Devuelve el valor del atributo *attr* del objeto *obj*. Si no tiene ese atributo devuelve default.
setattr (obj, attr, val)	Sobrescribe con val el atributo *attr* del objeto *obj*. Si no existe, lo crea y asigna *val.*
delattr (obj, attr)	Elimina el atributo *attr* del objeto *obj*.
dir (obj=None)	Muestra los atributos y métodos del objeto *obj*. Si vale *None*, devuelve variables locales y globales del espacio de nombres local.
super (type, obj)	Devuelve una referencia a la clase madre del objeto de tipo *type*.
vars (obj=None)	Devuelve un diccionario de atributos y valores del objeto *obj*. Si es *None*, devuelve un diccionario con las variables locales del espacio de nombres.

Métodos especiales

- Se emplean para extender la funcionalidad de las clases en Python.
- Algunos tienen funcionalidad por defecto (`__init__`, constructor y `__del__`, destructor)

Posibilitan:

- La emulación de tipos estándar.
- La sobrecarga|sobrescritura de operadores.

Métodos especiales (I)

- Son funciones de Python disponibles para su sobrecarga dentro de clases.

Built-in Functions				
abs()	divmod()	input()	open()	staticmethod()
all()	enumerate()	int()	ord()	str()
any()	eval()	isinstance()	pow()	sum()
basestring()	execfile()	issubclass()	print()	super()
bin()	file()	iter()	property()	tuple()
bool()	filter()	len()	range()	type()
bytearray()	float()	list()	raw_input()	unichr()
callable()	format()	locals()	reduce()	unicode()
chr()	frozenset()	long()	reload()	vars()
classmethod()	getattr()	map()	repr()	xrange()
cmp()	globals()	max()	reversed()	zip()
compile()	hasattr()	memoryview()	round()	__import__()
complex()	hash()	min()	set()	apply()
delattr()	help()	next()	setattr()	buffer()
dict()	hex()	object()	slice()	coerce()
dir()	id()	oct()	sorted()	intern()

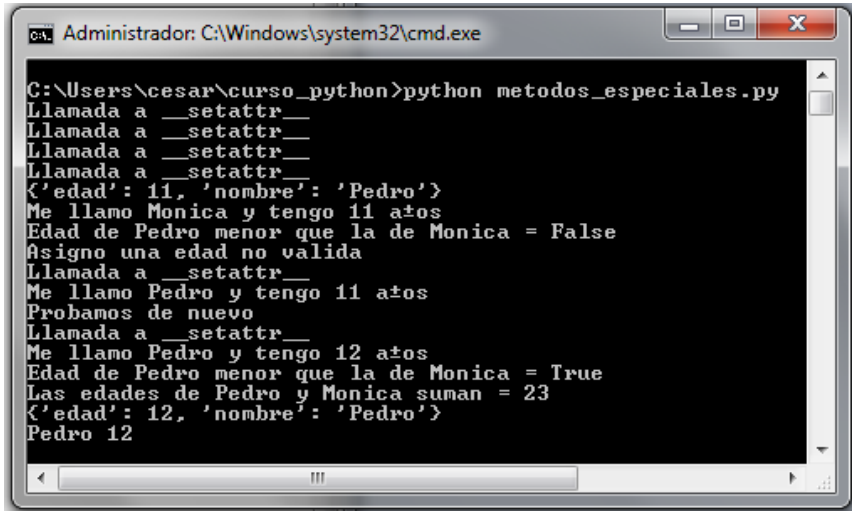
Métodos especiales (II)

Para especificar su comportamiento hay que sobrecargar las cabeceras de cada método que se quiera personalizar.

Operador	Sobrecarga	Cabecera método
nonzero (true o false)	__nonzero__	def __nonzero__(self):
== (comparación)	__cmp__	def __cmp__(self,otro):
= (asignación)	__eq__	def __eq__ (self,otro):
!= (comparación)	__ne__	def __ne__(self,otro):
< / <= (comparación)	__lt__ / __le__	def __lt__(self,otro):
> / >= (comparación)	__gt__/__ge__	def __gt__(self,otro):
+ / - (suma,resta)	__add__ / __sub__	def __add_(self,otro):
* / '/' (multiplicación,división)	__mul__ / __div__	def __mul__ (self,otro):
<< (desplazamiento)	__lshift__	def __lshift__(self, otro):
>> (desplazamiento)	__rshift__	def __rshift__(self, otro)
AND (AND lógico)	__and__	def __and__(self,otro):
OR (OR lógico)	__or__	def __or__(self, otro)
XOR (XOR lógico)	__xor__	def __xor__(self,otro):
len (longitud de elemento)	__len__	def __len__ (self,otro):
invert (invertir)	__invert__	def __invert__(self):
call (llamada a objetos/métodos)	__call__	def __call__(self,[args]):
[ ] (acceso lectura elemento)	__getitem__	def __getitem__(self, key):
[ ] (acceso escritura elemento)	__settitem__	def __setitem__(self, key, item):
del a[b] (borrado objeto)	__delitem__	def __delitem__(self, clave):

Métodos especiales (III): Ejemplo

```
#!/usr/bin/env python
#-*- coding: UTF-8 -*-

class Persona (object):
        def __init__ (self, nombre, edad):
                self.nombre = nombre
                self.edad = edad

        def __repr__ (self):
                return 'Me llamo %s y tengo %d años' % \
(self.nombre, self.edad)

        def __lt__ (self, obj):
                resultado = False
                if self.edad < obj.edad:
                        resultado = True
                return resultado

        def __getattr__ (self, edad):
                return float(self.__dict__[edad])

        def __setattr__ (self, clave, val):
                print 'Llamada a __setattr__'
                if clave == 'edad':
                        if isinstance (val, int):
                                self.__dict__[clave] = val
                else:
                        self.__dict__[clave] = val
                #print 'self.__dict__ =', self.__dict__


        def __add__ (self, obj):
                return self.edad + obj.edad


if __name__ == '__main__':
        p = Persona ('Pedro', 11)
        dir(p)
        m = Persona ('Monica', 11)

        print p.__dict__
        print m
        print 'Edad de Pedro menor que la de Monica =', p > m

        print 'Asigno una edad no valida'
        p.edad = 'palabra'
        print p

        print 'Probamos de nuevo'
        p.edad = 12

        print p

        print 'Edad de Pedro menor que la de Monica =', p > m

        print 'Las edades de Pedro y Monica suman =', p + m

        print p.__dict__
        print p.__dict__['nombre'],
```



EJERCICIOS: 11 Y 12

Para finalizar…

Listas con clase…
(las listas que hemos estado manejando desde el día 1 del curso son clases)

```
>>> dir (list ())
['__add__', '__class__', '__contains__', '__delattr__', '__delitem__', '__delslice__', '__doc__', '
__eq__', '__format__', '__ge__', '__getattribute__', '__getitem__', '__getslice__', '__gt__', '__ha
sh__', '__iadd__', '__imul__', '__init__', '__iter__', '__le__', '__len__', '__lt__', '__mul__', '_
_ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__reversed__', '__rmul__', '__setatt
r__', '__setitem__', '__setslice__', '__sizeof__', '__str__', '__subclasshook__', 'append', 'count'
, 'extend', 'index', 'insert', 'pop', 'remove', 'reverse', 'sort']
>>> help (list)
Help on class list in module __builtin__:

class list(object)
 |  list() -> new empty list
 |  list(iterable) -> new list initialized from iterable's items
 |
 |  Methods defined here:
 |
 |  __add__(...)
 |      x.__add__(y) <==> x+y
 |
```

Para finalizar…

Diccionarios con clase…
(También existe la clase diccionario. ¿Sorprendido/a?)

```
>>> dir (dict)
['__class__', '__cmp__', '__contains__', '__delattr__', '__delitem__', '__doc__', '__eq__', '__form
at__', '__ge__', '__getattribute__', '__getitem__', '__gt__', '__hash__', '__init__', '__iter__', '
__le__', '__len__', '__lt__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__se
tattr__', '__setitem__', '__sizeof__', '__str__', '__subclasshook__', 'clear', 'copy', 'fromkeys',
'get', 'has_key', 'items', 'iteritems', 'iterkeys', 'itervalues', 'keys', 'pop', 'popitem', 'setdef
ault', 'update', 'values', 'viewitems', 'viewkeys', 'viewvalues']
>>> help (dict)
Help on class dict in module __builtin__:

class dict(object)
 |  dict() -> new empty dictionary
 |  dict(mapping) -> new dictionary initialized from a mapping object's
 |      (key, value) pairs
 |  dict(iterable) -> new dictionary initialized as if via:
 |      d = {}
 |      for k, v in iterable:
 |          d[k] = v
 |  dict(**kwargs) -> new dictionary initialized with the name=value pairs
 |      in the keyword argument list.  For example:  dict(one=1, two=2)
 |
 |  Methods defined here:
 |
 |  __cmp__(...)
 |      x.__cmp__(y) <==> cmp(x,y)
 |
```

FIN