SG VIRTUAL CONFERENCE
6ta edición

Introducción a Python con Orientación a Objetos

Presentado por:
José Luis Chiquete

Particularidades de OOP en Python

- Todo es un objeto, incluyendo los tipos y clases.
- Permite herencia múltiple.
- No existen métodos ni atributos privados.
- Los atributos pueden ser modificados directamente.
- Permite "monkey patching".
- Permite "duck typing".
- Permite la sobrecarga de operadores.
- Permite la creación de nuevos tipos de datos.

object

Todo, incluyendo las clases y tipos de Python son instancias de ***object***.

Para corroborar si un objeto es instancia de unas clase se utiliza la función ***isinstance()***.

```
Help on class object in module __builtin__:

class object
 |  The most base type
(END)
```

```
>>> isinstance(int, object)
True
>>> isinstance(tuple, object)
True
>>> isinstance(tuple, int)
False
```

SG VIRTUAL CONFERENCE 6ta edición

Definición de una clase

- Para definir una clase se utiliza la expresión ***class***.

```
class <ClaseNueva>(object):
    ...
    ...
    ...
```

- Con la sintaxis anterior, la clase nueva "*hereda*" los métodos y atributos de ***object***.

Creación de una clase básica

```
>>> class ClaseBasica(object):
...     """Definición de una clase básica."""
...     pass
...
```

```
>>> dir(ClaseBasica)
['__class__', '__delattr__', '__dict__', '__doc__', '__format__', '__getattribut
e__', '__hash__', '__init__', '__module__', '__new__', '__reduce__', '__reduce_e
x__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', '_
_weakref__']
>>> help(ClaseBasica)
```

```
Help on class ClaseBasica in module __main__:

class ClaseBasica(__builtin__.object)
 |  Definición de una clase básica.
 |
 |  Data descriptors defined here:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)
(END)
```

Definición de clases en Python 3

```
>>> class ClaseNueva:
...     """Definición de clase en Python 3. No es necesario especificar
...     la referencia a object."""
...     pass
...
>>> dir(ClaseNueva)
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__form
at__', '__ge__', '__getattribute__', '__gt__', '__hash__', '__init__', '__le__',
 '__lt__', '__module__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__
repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', '__weakref_
_']
>>> help(ClaseNueva)
```

```
Help on class ClaseNueva in module __main__:

class ClaseNueva(builtins.object)
 |  Definición de clase en Python 3. No es necesario especificar
 |  la referencia a object.
 |
 |  Data descriptors defined here:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)
(END)
```

Instanciamiento de un objeto a partir de una clase

- Para instanciar un objeto a partir de una clase se utiliza el operador de asignación "**=**".
- El objeto instanciado es ligado al nombre en el espacio de nombres.
- Es posible crear objetos dentro de un objeto, tal como es el caso de las listas.
- Cada objeto tiene su propio identificador interno, el cual puede ser consultado con la función ***id()***.

Instanciamiento de un objeto a partir de una clase

```
>>> class ClaseBasica(object):
...     """Clase básica. Hereda los métodos y atributos de object,
...     pero no añade nada más"""
...     pass
...
>>> objeto = ClaseBasica()
>>> print objeto
<__main__.ClaseBasica object at 0x7ff9052dfc50>
>>> id(objeto)
140707510484048
>>> help(objeto)
```

```
Help on ClaseBasica in module __main__ object:

class ClaseBasica(__builtin__.object)
 |  Clase básica. Hereda los métodos y atributos de object,
 |  pero no añade nada más
 |
 |  Data descriptors defined here:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)
(END)
```

Instanciamiento de objetos dentro de un objeto

```
>>> lista_obj = [ClaseBasica(), ClaseBasica(), ClaseBasica()]
>>> print lista_obj
[<__main__.ClaseBasica object at 0x7f5c2788be50>, <__main__.ClaseBasica
at 0x25a2090>, <__main__.ClaseBasica object at 0x25a20d0>]
>>> id(lista_obj)
140033776902800
>>> print lista_obj[0]
<__main__.ClaseBasica object at 0x7f5c2788be50>
>>> id(lista_obj[0])
140033776991824
>>> print lista_obj[1]
<__main__.ClaseBasica object at 0x25a2090>
>>> id(lista_obj[1])
39461008
>>> print lista_obj[2]
<__main__.ClaseBasica object at 0x25a20d0>
>>> id(lista_obj[2])
39461072
>>>
```

Atributos y métodos

- Un objeto cuenta con elementos que almacenan datos y otros que ejecutan acciones.
- A los elementos que almacenan datos dentro de un objeto se les denomina "*atributos*".

- A las piezas de código que realizan ciertas tareas inherentes del objeto se conocen como "*métodos*".

Ejemplo de atributos y métodos

- Los objetos de tipo ***complex*** cuentan con los atributos ***real*** e ***imag***, los cuales contienen los componentes reales e imaginarios del número correspondientemente.
- Además, dichos objetos cuentan con el método ***conjugate()***, el cual regresa el número conjugado del objeto.

```
>>> complejo = 15.901 + 81.07j
>>> complejo
(15.901+81.07j)
>>> type(complejo)
<type 'complex'>
>>> complejo.real
15.901
>>> complejo.imag
81.07
>>> complejo.conjugate()
(15.901-81.07j)
>>>
```

Nombres de atributos y métodos

- Sin guiones, son atributos normales.
  - ***dibuja***, ***superficie***, ***desp_datos()***,
- Encerrados entre dobles guiones bajos son atributos especiales.
  - ***__init__()***, ***__name__***, ***__del__()***, ***__doc__***
- Con dobles guiones bajos al principio son atributos "escondidos".
  - ***__privado***, ***__no_tocar***

Definición de atributos

- Un atributo se define de la siguiente manera:

```
class <Clase>(object):
    ...
    <nombre> = <contenido>
    ...
```

```
>>> class GuardaNumero(object):
...     numero = 3
...
>>> dato = GuardaNumero()
>>> dato.numero
3
>>>
```

Uso de atributos

- En Python es posible modificar el atributo de un objeto sin necesidad de acceder a éste por medio de un método.
- Lo único que se requiere para modificar un atributo es un operador de asignación.

```
>>> class GuardaNumero(object):
...     numero = 3
...
>>> cuatro = GuardaNumero()
>>> cuatro.numero
3
>>> cuatro.numero = 4
>>> cuatro.numero
4
>>>
```

Adición de atributos a los objetos

```
>>> class ObjetoBasico(object):
...     pass
...
>>> objeto = ObjetoBasico()
>>> otro_objeto = ObjetoBasico()
>>> objeto.saluda = "Hola"
>>> objeto.despidete = "Adios"
>>> dir(objeto)
['__class__', '__delattr__', '__dict__', '__doc__', '__format__', '__getattribut
e__', '__hash__', '__init__', '__module__', '__new__', '__reduce__', '__reduce_e
x__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', '_
_weakref__', 'despidete', 'saluda']
>>> print objeto.saluda, objeto.despidete
Hola Adios
>>> dir(otro_objeto)
['__class__', '__delattr__', '__dict__', '__doc__', '__format__', '__getattribut
e__', '__hash__', '__init__', '__module__', '__new__', '__reduce__', '__reduce_e
x__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', '_
_weakref__']
>>> print otro_objeto.saluda, otro_objeto.despidete
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'ObjetoBasico' object has no attribute 'saluda'
>>>
```

Definición de métodos

- La única diferencia sintáctica entre la definición de un método y la definición de una función es que el primer parámetro del método por convención debe ser el nombre "***self***".

```
class <Clase>(object):
    ...
    ...
    def <nombre>(self, <argumentos>):
        ...
        ...
    ...
    ...
```

Definición de métodos

```
>>> class ClaseSaluda(object):
...     def saluda(self):
...         return "Hola"
...
>>> cortesia = ClaseSaluda()
>>> cortesia.saluda()
'Hola'
>>>
```

Ámbito de los métodos

- Los métodos cuentan con un espacio de nombres propio.
- En caso de no encontrar un nombre en su ámbito local, buscará en el ámbito superior hasta encontrar alguna coincidencia.
- Los métodos pueden acceder y crear atributos dentro del objeto al que pertencen, anteponiendo la palabra ***self*** y el operador de atributo "**.**" antes del nombre del atributo en cuestión.

El script ambitos_metodos

```
#! /usr/bin/python
# -*- coding: utf-8 -*-
"""Script que ejemplifica el modo en el que los objetos interactúan con los
distintos ámbitos."""

numero = 6


class ClaseconMetodo(object):
    """Clase que despliega un saludo varias veces en función
    de un nombre global."""

    dato = "Buen día "

    def saludo(self, nombre):
        """Método que hace uso de nombres globales, atributos,
        argumentos y nombres locales."""
        mensaje = self.dato + nombre + ".\n"
        print mensaje * numero

objeto = ClaseconMetodo()
objeto.saludo("Juan")
```

Métodos especiales

- Las clases en Python cuentan con múltiples métodos especiales , los cuales se encuentran entre dobles guiones bajos __<*metodo*>__()
- Los métodos especiales más utilizados son ***__init__()*** y ***__del__()***
- El método ***__init__()*** se ejecuta tan pronto como un objeto de una clase es instanciado.
- El método ***__del__()*** se ejecuta cuando un objeto es desechado.

El método ***__init__()***

- El método ***__init__()*** es un método especial, el cual se ejecuta al momento de instanciar un objeto.
- El comportamiento de ***__init__()*** es muy similar a los "*constructores*" en otros lenguajes.
- Los argumentos que se utilizan en la definición de ***__init__()*** corresponden a los parámetros que se deben ingresar al instanciar un objeto.

El método ***__del__()***

- El método ***__del__()*** es un método especial, el cual se ejecuta al momento de que un objeto es descartado por el intérprete.
- El comportamiento de ***__del__()*** es muy similar a los "*destructores*" en otros lenguajes.

El script ***init_y_del.py***

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
"""Script que ejemplifica el uso de los métodos __init__() y __del__()"""


class perico():

    def __init__(self, nombre):
        """Método que se ejecuta al instanciar un objeto."""
        self.nombre = nombre.capitalize()
        print "Salí del cascarón. Mi nombre es", self.nombre + "."

    def habla(self):
        """Método normal."""
        print self.nombre.capitalize(), "quiere una galleta."

    def __del__(self):
        """Método que se ejecuta al descartar al objeto."""
        print "¡Ack!", self.nombre, "ha muerto."

poli = perico("poli")
juancho = perico("Juancho")
choforo = perico("Choforito")
raw_input("\nPulse <INTRO> para que hable Poli.")
poli.habla()
raw_input("\nPulse <INTRO> para que se resfríe Juancho.")
#El objeto juancho es desechado durante la ejecución del script
del juancho
raw_input("\nPulse <INTRO> para que termine el programa.")
'''Al terminar de ejecutarse el script, todos los objetos son desechados.
   Cuando el script es importado, los objetos existiran hasta que sean
   desechados o hasta que el entorno interactivo se cierre.'''
```

El script ***init_y_del.py*** ejecutado en terminal

```
josech@x230:~/python_OOP$ ./init_y_del.py
Salí del cascarón. Mi nombre es Poli.
Salí del cascarón. Mi nombre es Juancho.
Salí del cascarón. Mi nombre es Choforito.

Pulse <INTRO> para que hable Poli.
Poli quiere una galleta.

Pulse <INTRO> para que se resfríe Juancho.
¡Ack! Juancho ha muerto.

Pulse <INTRO> para que termine el programa.
¡Ack! Choforito ha muerto.
¡Ack! Poli ha muerto.
josech@x230:~/python_OOP$
```

El script ***init_y_del.py*** importado

```
>>> import init_y_del
Salí del cascarón. Mi nombre es Poli.
Salí del cascarón. Mi nombre es Juancho.
Salí del cascarón. Mi nombre es Choforito.

Pulse <INTRO> para que hable Poli.
Poli quiere una galleta.

Pulse <INTRO> para que se resfríe Juancho.
¡Ack! Juancho ha muerto.

Pulse <INTRO> para que termine el programa.
>>> init_y_del.choforo.habla()
Choforito quiere una galleta.
>>> del init_y_del.poli
¡Ack! Poli ha muerto.
>>> init_y_del.poli.habla()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'module' object has no attribute 'poli'
>>> exit()
¡Ack! Choforito ha muerto.
josech@x230:~/python_OOP$
```

Estado de un objeto

- Al conjunto de datos y objetos relacionados con un objeto en un momento dado, se le conoce como "*estado*".
- Un objeto puede tener múltiples estados a lo largo de su existencia conforme se relaciona con su entorno y otros objetos.

Interfaces

- La manera en que los métodos de un objeto pueden ser accedidos por otros objetos se conoce como "***interfaz***".
- Una interfaz bien definida permite a objetos de distinta índole interactuar entre sí de forma modular.
- La interfaz define el modo en que los objetos intercambian información.

Implementaciones

- Una implementación corresponde al mecanismo interno que se desencadena en un método cuando éste es invocado.
- Las implementaciones procesan las entradas proveniente de las interfaces y actúan en consecuencia ya sea:
  - Modificando el estado del objeto.
  - Transfiriendo la información resultante del proceso interno a través de la interfase.

Encapsulamiento

- El encapsulamiento en OOP se refiere a la capacidad que tienen los objetos de interactuar con otros por medio de las interfaces:
  - Independientemente de la implementación.
  - De forma modular e intercambiable.
  - Con la información suficiente de entrada y de salida.
- En algunos otros lenguajes de programación el encapsulamiento también se refiere a restringir el acceso a los elementos de los objetos al mínimo posible.

Encapsulamiento en Python

- En Python, el encapsulamiento consiste en crear interfaces eficaces antes que en esconder la implementación de los objetos.
- A diferencia de otros lenguajes, NO existen atributos ni métodos privados dentro de Python.
- Python permite acceder a los atributos de un objeto sin necesidad de que haya un método de por medio.

Name mangling

- En el caso de querer restringir de algún modo el acceso a ciertos atributos o métodos, éstos se pueden esconder mediante una técnica conocida como "name mangling".
- Los atributos que utilizan "name mangling" se comportan de forma muy parecida a un método estático.
- Los atributos no son despelgados usando ***help()***, pero sí son listados con ***dir()***

Name mangling

class <Clase>:

    __<atributo_restringido>

...

...

    def   __<metodo_restringido>(self,<parámetros>):

        ...

        ...

<objeto> = <Clase>()

<variable> = <objeto>._<Clase>__<atributo_restringido>

<objeto>._<Clase>__<campo_restringido>(<parámetros>)

Relaciones de objetos en Python

- En OOP existen 2 tipos de relaciones principales.
  - Relación "***es un***", la cual se realiza mediante la herencia.
  - Relación "***tiene un***", la cual se realiza mediante la asociación de los objetos.

Herencia

- Es posible crear nuevas clases a partir de una o varias clases mediante la herencia.
- La clase original se denomina superclase.
- La clase que hereda los atributos y métodos de la superclase se denomina subclase.
- Se pueden definir atributos y métodos adicionales a la superclase e incluso se pueden sobrescribir los atributos y métodos heredados en la subclase.

Herencia no es instanciamiento

- La herencia es una relación exclusiva entre clases.
- Todas las clases y tipos en Python son subclases de ***object***.
- El instanciamiento crea objetos a partir de una clase, pero no es posible heredar de un objeto a una clase.

issubclass()

- La función ***issubclass()*** comprueba si una clase es subclase de otra.

```
>>> class clase(object):
...     pass
...
>>> issubclass(clase, object)
True
```

Herencia

- La herencia de clases en Python es muy simple.

class <SuperClase>(object):

...

...

class <SubClase>(<SuperClase>, <OtraSuperClase>,...):

...

...

El script ***herencia.py***

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

import math


class Forma(object):

    def __init__(self):
        pass

    def superficie(self):
        pass

    def perimetro(self):
        pass


class Circulo(Forma):

    def __init__(self, radio=1):
        self.radio = radio

    def superficie(self):
        return math.pi * self.radio ** 2

    def perimetro(self):
        return math.pi * 2 * self.radio
```

El script ***herencia.py***

```
29
30
31  class Rectangulo(Forma):
32
33      def __init__(self, base=1, altura=2):
34          self.base = base
35          self.altura = altura
36
37      def superficie(self):
38          return self.base * self.altura
39
40      def perimetro(self):
41          return 2 * (self.base + self.altura)
42
43  FormaRectangular = Rectangulo()
44  FormaCircular = Circulo(45)
45  print "La superficie del rectángulo es", FormaRectangular.superficie()
46  print "La superficie del círculo es", FormaCircular.superficie()
47
```

El script ***herencia.py***

```
La superficie del rectángulo es 2
La superficie del círculo es 6361.72512352
```

El script ***herencia.py***

```
FILE
    /home/josech/Documentos/Cursos/Contenidos/python/codigo/herencia.py

CLASSES
    __builtin__.object
        Forma
            Circulo
            Rectangulo

    class Circulo(Forma)
     |  Method resolution order:
     |      Circulo
     |      Forma
     |      __builtin__.object
     |
     |  Methods defined here:
     |
     |  __init__(self, radio=1)
     |
     |  perimetro(self)
     |
     |  superficie(self)
     |
     |  ----------------------------------------------------------------------
     |  Data descriptors inherited from Forma:
     |
     |  __dict__
     |      dictionary for instance variables (if defined)
     |
     |  __weakref__
     |      list of weak references to the object (if defined)
```

El script ***herencia.py***

```
class Forma(__builtin__.object)
 |  Methods defined here:
 |
 |  __init__(self)
 |
 |  perimetro(self)
 |
 |  superficie(self)
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors defined here:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)

class Rectangulo(Forma)
 |  Method resolution order:
 |      Rectangulo
 |      Forma
 |      __builtin__.object
 |
 |  Methods defined here:
 |
 |  __init__(self, base=1, altura=2)
 |
 |  perimetro(self)
 |
 |  superficie(self)
```

El script ***herencia.py***

```
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors inherited from Forma:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)

DATA
    FormaCircular = <herencia.Circulo object>
    FormaRectangular = <herencia.Rectangulo object>
```

Abstracción

- En el script ***herencia.py*** puede observarse que la clase ***Forma*** define una serie de métodos, pero ninguno de éstos realiza alguna acción.
- La clase ***Forma*** únicamente define las *interfaces* de los métodos, mientras que sus subclases definen las *implementaciones* de dichos métodos mediante la sobrescritura de éstos.
- A esta técnica se le conoce como "*abstracción*".

Abstracción en Python

- En otros lenguajes se pueden crear clases y métodos abstractos de forma explícita.
- Python no requiere de una definición explícita de una clase o método abstracto.
- El módulo ***abc*** permite el uso explícito de clases abstractas básicas tal como se especifica en en el PEP 3119 (http://www.python.org/dev/peps/pep-3119/)

Extensión de métodos sobrescritos

- Es común que los métodos de una subclase no requieran de sobrescribir por completo el método de la superclase, sino más bien extenderlo.
- Python permite reutilizar el código contenido en un método de una superclase mediante la función ***super()***.

La función ***super()*** en Python 2

La sintaxis en Python 2 de la función ***super()*** es la siguiente:

```
class <SuperClase>:
    def <metodo>(self, <argumentos>)
    ...
    ...
class <SubClase>(<SuperClase>):
    def <metodo>
        super(<SubClase>, self).<metodo>
        ...
```

La función ***super()*** en Python 3

La sintaxis en Python 3 de la función ***super()*** es la siguiente:

```
class <SuperClase>:
    def <metodo>
    ...
    ...
class <SubClase>(<SuperClase>):
    def <metodo>
        super().<metodo>
        ...
```

herencia_multiple.py

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


class Animal(object):

    def __init__(self, nombre):
        print "El animal %s acaba de nacer" % nombre
        self.nombre = nombre

    def reproduccion(self):
        pass


class Mamifero(Animal):

    def produce_leche(self):
        print "Aquí hay un poco de leche."

    def reproduccion(self, tiempo):
        self.tiempo_gestacion = tiempo
        print "Despueś de %d meses nacen las crías." % self.tiempo_gestacion


```

herencia_multiple.py

```
25  class Reptil(Animal):
26
27      def produce_veneno(self, venenoso):
28          self.veneno = venenoso
29          if venenoso:
30              print "Soy venenenoso."
31
32      def reproduccion(self):
33
34          print "Aquí hay un huevo."
35
36
37  class Ornitorrinco(Reptil, Mamifero):
38      def __init__(self, nombre):
39          super(Ornitorrinco, self).__init__(nombre)
40          print "¡qué demonios!"
41
42  perry = Ornitorrinco("Agente P")
43  perry.reproduccion()
44  perry.produce_veneno(True)
```

herencia_multiple.py

```
El animal Agente P acaba de nacer
¡qué demonios!
Aquí hay un huevo.
Soy venenenoso.
```

herencia_multiple.py

```
Help on module herencia_multiple:

NAME
    herencia_multiple - # -*- coding: utf-8 -*-

FILE
    /home/josech/Documentos/Cursos/Contenidos/python/codigo/herencia_multiple.py

CLASSES
    __builtin__.object
        Animal
            Mamifero
            Reptil
                Ornitorrinco(Reptil, Mamifero)

    class Animal(__builtin__.object)
     |  Methods defined here:
     |
     |  __init__(self, nombre)
     |
     |  reproduccion(self)
     |
     |  ----------------------------------------------------------------------
     |  Data descriptors defined here:
     |
     |  __dict__
     |      dictionary for instance variables (if defined)
     |
     |  __weakref__
     |      list of weak references to the object (if defined)
```

herencia_multiple.py

```
class Mamifero(Animal)
 |  Method resolution order:
 |      Mamifero
 |      Animal
 |      __builtin__.object
 |
 |  Methods defined here:
 |
 |  produce_leche(self)
 |
 |  reproduccion(self, tiempo)
 |
 |  ----------------------------------------------------------------------
 |  Methods inherited from Animal:
 |
 |  __init__(self, nombre)
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors inherited from Animal:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)
```

herencia_multiple.py

```
class Ornitorrinco(Reptil, Mamifero)
 |  Method resolution order:
 |      Ornitorrinco
 |      Reptil
 |      Mamifero
 |      Animal
 |      __builtin__.object
 |
 |  Methods defined here:
 |
 |  __init__(self, nombre)
 |
 |  ----------------------------------------------------------------------
 |  Methods inherited from Reptil:
 |
 |  produce_veneno(self, venenoso)
 |
 |  reproduccion(self)
 |
 |  ----------------------------------------------------------------------
 |  Methods inherited from Mamifero:
 |
 |  produce_leche(self)
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors inherited from Animal:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)
```

herencia_multiple.py

```
class Reptil(Animal)
 |  Method resolution order:
 |      Reptil
 |      Animal
 |      __builtin__.object
 |
 |  Methods defined here:
 |
 |  produce_veneno(self, venenoso)
 |
 |  reproduccion(self)
 |
 |  ----------------------------------------------------------------------
 |  Methods inherited from Animal:
 |
 |  __init__(self, nombre)
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors inherited from Animal:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)

DATA
    perry = <herencia_multiple.Ornitorrinco object>
```

Asociación de objetos

- La agregación y la composición representan una forma en la que se asocian los objetos con otros objetos.
- En el caso de la agregación, los objetos contenidos pueden existir independientemente de la existencia del contenedor.
- En el caso de la composición, los objetos contenidos en el objeto principal, perduran mientras exista dicho objeto.
- La composición es un caso particular de agregación.

Composición y agregación en Python

- En vista de que en Python los atributos de un objeto también son objetos los cuales están vinculados al objeto original en su espacio de nombres; la composición y la agregación sólo dependen de las referencias que se haga a los objetos en los distintos espacios de nombres.

```
>>> lista = [[1, 2, 3], True, ['Saludo', 'Despedida']]
>>> print lista[0]
[1, 2, 3]
>>> id(lista[0])
140146622677648
>>> numeros = lista[0]
>>> id(numeros)
140146622677648
>>> del lista
>>> print numeros
[1, 2, 3]
>>>
```

"Monkey patching"

- Debido a que las funciones son objetos en Python, es posible añadirlas a un objeto como si fueran atributos. A ésto se le conoce como "Monkey patching".
- Es posible añadir métodos a una clase de forma dinámica mediante "Monkey patching".
- Los métodos son un tipo particular de atributo en Python.

"Monkey patching"

```
>>> class ClaseMaestra(object):
...     pass
...
>>> def funcion():
...     return "Hola"
...
>>> objeto = ClaseMaestra()
>>> dir(objeto)
['__class__', '__delattr__', '__dict__', '__doc__', '__format__', '__getattribut
e__', '__hash__', '__init__', '__module__', '__new__', '__reduce__', '__reduce_e
x__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', '_
_weakref__']
>>> objeto.saludo = funcion
>>> objeto.saludo()
'Hola'
>>> dir(objeto)
['__class__', '__delattr__', '__dict__', '__doc__', '__format__', '__getattribut
e__', '__hash__', '__init__', '__module__', '__new__', '__reduce__', '__reduce_e
x__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', '_
_weakref__', 'saludo']
>>>
```

"Monkey patching"

```
>>> class ClaseMaestra(object):
...     pass
...
>>> def metodo_superpuesto(self):
...    return "Hola"
...
>>> objeto = ClaseMaestra()
>>> otro_objeto = ClaseMaestra()
>>> ClaseMaestra.metodo = metodo_superpuesto
>>> objeto.metodo()
'Hola'
>>> otro_objeto.metodo()
'Hola'
>>> metodo_superpuesto()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: metodo_superpuesto() takes exactly 1 argument (0 given)
>>>
```

Polimorfismo

- Polimorfismo es la característica de los objetos de comportarse de maneras distintas ante una interfaz dada, dependiendo de la información dada y un contexto específico.
- La sobrecarga de operadores es un tipo de polimorfismo.

```
>>> 2 * 5
10
>>> '2' * 5
'22222'
>>> 2 * '5'
'55'
>>>
```

Polimorfismo

- El polimorfismo se basa en la aplicación de diversas implementaciones a partir de una clase abstracta que define una interfaz estándar.
- E polimorfismo se da exclusivamente entre objetos instanciados de clases con una superclase común.

Polimorfismo en Python

- Python puede sobrescribir cualquier atributo definido por una superclase, incluyendo los métodos relativos a los operadores.
- De ese modo, es posible hacer diversas implementaciones del atributo para cada subclase.

"Duck typing"

- Python permite que cualquier objeto que tenga una interfaz sintácticamente compatible pueda ser ejecutado sin necesidad de tener una superclase común.
- A esta técnica se le conoce como "duck typing".

El script ***ducktyping.py***

```python
#! //urs/bin/python
# -*- coding: utf-8 -*-
'''Script que ilustra el uso de duck typing'''


class CapitalesMundiales(object):
    capitales = {
                "México": "Distrito Federal",
                "Argentina": "Buenos Aires",
                "Uruguay": "Montevideo",
                "Brasil": "Sao Paulo",
                "Estados Unidos": "Washington, D.C."
                }

    def __init__(self, pais="México"):
        self.pais = pais

    def capitalize(self):
        if self.pais in self.capitales:
            return self.capitales[self.pais]
        else:
            return 'País desconocido'


def capital(objeto):
    return objeto.capitalize()

pais = CapitalesMundiales("Bolivia")
mensaje = "HOLA"

print capital(pais)
print capital(mensaje)
```

Copia de objetos

- El módulo ***copy()*** permite realizar copias de objetos de forma superficial o intensiva.
- ***copy.copy()*** hace la copia superficial, creando un objeto nuevo, pero con las referencias de sus contenidos intactas.
- ***copy.deepcopy()*** hace una copia a profundidad porcurando crear nuevos objetos a partir de los contenidos.

Copia de objetos

```
>>> import copy
>>> lista = [[1, 2], [3, 4], [5,6]]
>>> lista_1 = copy.copy(lista)
>>> lista_2 = copy.deepcopy(lista)
>>> del lista[1]
>>> lista
[[1, 2], [5, 6]]
>>> lista_1
[[1, 2], [3, 4], [5, 6]]
>>> lista_2
[[1, 2], [3, 4], [5, 6]]
>>> lista[1].append(12)
>>> lista
[[1, 2], [5, 6, 12]]
>>> lista_1
[[1, 2], [3, 4], [5, 6, 12]]
>>> lista_2
[[1, 2], [3, 4], [5, 6]]
```

Persistencia de objetos

- Una vez que se termina de ejecutar un programa, los objetos y los estados en los que se encuentran son destruidos.
- Python puede preservar los objetos en un archivo mediante el módulo ***pickle***

El script ***persistencia.py***

```python
#! /usr/bin/python
# -*- coding: utf-8 -*-
"""Script que ilustra la persistencia de objetos mediante el uso
del módulo pi8kle"""

import pickle
lista = [[1, 2, 3], [4, 5, 6]]
"""Se guarda el objeto"""
with open("objeto.bin", "wb") as archivo:
    pickle.dump(lista, archivo)
"""Se recupera el objeto"""
with open("objeto.bin", "rb") as archivo:
    otra_lista = pickle.load(archivo)
print "La lista es:", lista
print id(lista)
print id(otra_lista)
if lista == otra_lista:
    print"Estas listas son idénticas."
else:
    print"Estas listas no son iguales."
```

SG VIRTUAL CONFERENCE
6ta edición

José Luis Chiquete


@josech


josech@gmail.com

http://slideshare.net/josech