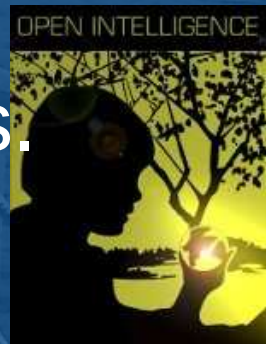


PROGRAMACIÓN ORIENTADA A OBJETOS



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.



Clases y objetos

- Las clases son prototipos a partir de los cuales pueden crearse objetos que adquieren las propiedades, características y comportamientos definidos por las clases.
- A la creación de un objeto a partir de una clase, se le llama *"instanciar"*.
- Todos los elementos de Python son instancias de al menos una clase.



object

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

Para corroborar si un objeto es instancia de una 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
```



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*.



Convención para nombres de clases

- Según la PEP-8, las clases deben utilizar mayúsculas al principio de cada palabra que conforma su nombre (conocidas como CapWords o CamelCase).
- No se deben usar caracteres especiales.
- Las clases de uso interno pueden ser precedidas por un guión bajo.
- Algunos ejemplos:
 - *ClaseBasica*
 - *CalculoNumerico*
 - *_ManejodeMemoria*
 - *Figura*



Creación de una clase básica

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

```
>>> dir(ClaseBasica)  
['_class_', '__delattr__', '__dict__', '__doc__', '__format__', '__getattribute__', '__hash__', '__init__', '__module__', '__new__', '__reduce__', '__reduce_ex__', '__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)



Clases de viejo estilo

- Hasta la versión 2.1 de Python, las clases y los tipos eran entidades distintas y las clases se definían de la siguiente forma:

```
class <ClaseNueva>:  
    ...  
    ...
```

- Por motivos de compatibilidad con versiones anteriores, esta sintaxis está aún permitida en las versiones más recientes de Python 2, pero no corresponde al "nuevo estilo" de clases en el que clases y tipos son iguales.
- Python 3 ya no soporta al viejo estilo, por lo que la sintaxis descrita crea un objeto que "*hereda*" a ***object*** sin necesidad de especificarlo.



Clases de viejo estilo en Python 2

```
>>> class ClaseNueva:
...     """Clase del viejo estilo. Se considera obsoleta."""
...     pass
...
>>> dir(ClaseNueva)
['_doc_', '_module_']
>>> help(ClaseNueva)
```

Help on class ClaseNueva in module __main__:

```
class ClaseNueva
| Clase del viejo estilo. Se considera obsoleta.
(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__', '__format__', '__ge__', '__getattr__', '__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*".

<objeto>.<atributo>

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

<objeto>.<atributo>(<parámetros>)



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  
>>> █
```



Atributos de clase

- Al igual que los objetos, es posible modificar un atributo de clase con el operador de identidad.
- Al modificar el atributo de clase, el dato modificado se reflejará en los atributos de los objetos instanciados que no hayan modificado dicho atributo.



Atributos de clase

```
>>> class GuardaDatos(object):  
...     dato = 3  
...  
>>> █
```

```
>>> datos = [GuardaDatos(), GuardaDatos(), GuardaDatos()]  
>>> for item in datos: print item.dato  
...  
3  
3  
3  
>>> datos[1].dato = 12  
>>> for item in datos: print item.dato  
...  
3  
12  
3  
>>> GuardaDatos.dato = "hola"  
>>> for item in datos: print item.dato  
...  
hola  
12  
hola  
>>> █
```



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__', '__getattribute__', '__hash__', '__init__', '__module__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', '__weakref__', 'despidete', 'saluda']
>>> print objeto.saluda, objeto.despidete
Hola Adios
>>> dir(otro_objeto)
['__class__', '__delattr__', '__dict__', '__doc__', '__format__', '__getattribute__', '__hash__', '__init__', '__module__', '__new__', '__reduce__', '__reduce_ex__', '__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 pertenecen, anteponiendo la palabra ***self*** y el operador de atributo "." antes del nombre del atributo en cuestión.



El script ambitos_metodos

```
1  #!/usr/bin/python
2  # -*- coding: utf-8 -*-
3  """Script que ejemplifica el modo en el que los objetos interactúan con los
4  distintos ámbitos."""
5
6  numero = 6
7
8
9  class ClaseconMetodo(object):
10     """Clase que despliega un saludo varias veces en función
11     de un nombre global."""
12
13     dato = "Buen día "
14
15     def saludo(self, nombre):
16         """Método que hace uso de nombres globales, atributos,
17         argumentos y nombres locales."""
18         mensaje = self.dato + nombre + ".\n"
19         print mensaje * numero
20
21 objeto = ClaseconMetodo()
22 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*

```
1  #!/usr/bin/python
2  # -*- coding: utf-8 -*-
3  """Script que ejemplifica el uso de los métodos __init__() y __del__()."""
4
5
6  class perico():
7
8      def __init__(self, nombre):
9          """Método que se ejecuta al instanciar un objeto."""
10         self.nombre = nombre.capitalize()
11         print "Salí del cascarón. Mi nombre es", self.nombre + "."
12
13     def habla(self):
14         """Método normal."""
15         print self.nombre.capitalize(), "quiere una galleta."
16
17     def __del__(self):
18         """Método que se ejecuta al descartar al objeto."""
19         print "¡Ack!", self.nombre, "ha muerto."
20
21 poli = perico("poli")
22 juancho = perico("Juancho")
23 choforo = perico("Choforito")
24 raw_input("\nPulse <INTRO> para que hable Poli.")
25 poli.habla()
26 raw_input("\nPulse <INTRO> para que se resfríe Juancho.")
27 #El objeto juancho es desechado durante la ejecución del script
28 del juancho
29 raw_input("\nPulse <INTRO> para que termine el programa.")
30 '''Al terminar de ejecutarse el script, todos los objetos son desechados.
31 ... Cuando el script es importado, los objetos existiran hasta que sean
32 ... desechados o hasta que el entorno interactivo se cierre.'''
```



El script *init_y_del.py* ejecutado en terminal

```
josech@x230:~/python_00P$ ./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_00P$ █
```



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_00P$
```



Decoradores

- Un decorador en Python permite modificar de forma dinámica a una función o un método mediante otra función.
- El funcionamiento detallado de los decoradores queda fuera de los alcances de este curso.
- Para invocar un decorador se utiliza el signo de arroba (@).
- Los decoradores en Python son discutidos y definidos en el PEP-318.
<http://www.python.org/dev/peps/pep-0318/>



Métodos de clase y métodos estáticos

- En ciertas ocasiones es necesario contar con métodos que interactúen con elementos de la clase de la cual el objeto es instanciado.
- Python permite definir métodos de clase y métodos dinámicos.



Métodos de clase

- Los métodos de clase son aquellos que están ligados directamente con los atributos definidos en la clase que los contiene.
- Para definir un método de clase se utiliza el decorador *@classmethod* y por convención se utiliza *cls* como argumento inicial en lugar de *self*.
- Del mismo modo, los métodos de clase utilizan el prefijo *cls* para referirse a los atributos de la clase.

```
class <Clase>(object):  
    ...  
    ...  
    @classmethod  
    def <metodo>(cls, <argumentos>):  
        ...  
        ...
```



El script *metodo_clase.py*

```
1  #!/usr/bin/python
2  # -*- coding: utf-8 -*-
3  '''Script que ejemplifica el uso de un método de clase.'''
4
5
6  class CaidaLibre(object):
7  ....'''clase que realiza el cálculo de la distancia recorrida por un objeto
8  ....al caer libremente desde un estado de reposo en un tiempo definido.'''
9  ....gravedad = 9.81 # coeficiente en metros sobre segundos cuadrados
10
11  ....@classmethod
12  ....def cambia_a_pies(cls):
13  .....'''Método que cambia el coeficiente gravitacional a pies sobre segundo
14  .....al cuadrado para todas las instancias de la clase'''
15  .....cls.gravedad = 32.174
16
17  ....def calculo(self, tiempo):
18  .....return (self.gravedad * tiempo ** 2) / 2
19
20  '''Instanciamos dos objetos de la misma clase'''
21  calculadora = CaidaLibre()
22  otra_calculadora = CaidaLibre()
23  tiempo = float(raw_input("Ingrese el tiempo de caída en segundos: "))
24  print "La distancia recorrida en metros es:", calculadora.calculo(tiempo)
25  print "\nCambiamos las unidades a pies en todas las instancias."
26  calculadora.cambia_a_pies()
27  print "\nLa distancia recorrida en pies es:", calculadora.calculo(tiempo)
28  print "La distancia recorrida en pies es:", otra_calculadora.calculo(tiempo)
```



El script *metodo_clase.py*

```
josech@x230:~/python_00P$ ./metodo_clase.py
Ingrese el tiempo de caída en segundos: 12
La distancia recorrida en metros es:, 706.32

Cambiamos las unidades a pies en todas las instancias.

La distancia recorrida en pies es: 2316.528
La distancia recorrida en pies es: 2316.528
josech@x230:~/python_00P$ █
```



Métodos estáticos

- Los métodos estáticos hacen referencia a las instancias y métodos de una clase.
- Para definir un método estático se utiliza el decorador *@staticmethod* y no utiliza ningún argumento inicial.
- Al no utilizar *self*, los métodos estáticos no pueden interactuar con los atributos y métodos de la instancia.
- Para referirse a los elementos de la clase, se debe utilizar el nombre de la clase como prefijo.

```
class <Clase>(object):  
    ...  
    ...  
    @staticmethod  
    def <metodo>(<argumentos>):  
        ...  
        ...
```



El script *metodo_estatico.py*

```
1  #!/usr/bin/python
2  #-*- coding: utf-8 -*-
3  '''Script que ejemplifica el uso de un método estático.'''
4
5
6  class Perico():
7      .... poblacion = 0
8
9      .... def __init__(self, nombre):
10         .... """Método que se ejecuta al instanciar un objeto."""
11         .... self.nombre = nombre.capitalize()
12         .... print "Salí del cascarón. Mi nombre es", self.nombre + "."
13         .... Perico.poblacion += 1
14
15     .... def habla(self):
16         .... """Método normal."""
17         .... print self.nombre.capitalize(), "quiere una galleta."
18
19     .... def __del__(self):
20         .... """Método que se ejecuta al descartar al objeto."""
21         .... print "¡Ack!", self.nombre, "ha muerto."
22         .... Perico.poblacion -= 1
23
24     .... @staticmethod
25     .... def modifica_poblacion(operador='+', numero=0):
26         .... Perico.poblacion = eval(str(Perico.poblacion) + operador + str(numero))
27     .... if Perico.poblacion < 0:
28         .... Perico.poblacion = 0
29     .... print "Ahora hay, %d pericos." % Perico.poblacion
```



El script *metodo_estatico.py*

```
>>> from metodo_estatico import Perico
>>> periquito = Perico("Pol1")
Salí del cascarón. Mi nombre es Pol1.
>>> print Perico.poblacion
1
>>> print periquito.poblacion
1
>>> perico = Perico("Choforo")
Salí del cascarón. Mi nombre es Choforo.
>>> print Perico.poblacion
2
>>> print perico.poblacion
2
>>> periquito.modifica_poblacion("+", 5)
Ahora hay, 7 pericos.
>>> print Perico.poblacion
7
>>> del perico
¡Ack! Choforo ha muerto.
>>> print Perico.poblacion
6
```



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>)
```



El script *secreto.py*

```
1  #-*- coding: utf-8 -*-
2  #!/usr/bin/python
3  """Módulo que contiene una clases con atributos de acceso restringido."""
4
5
6  class Confidencial(object):
7      """Esta clase crea objetos con una contraseña maestra."""
8      __password_maestro = "123qwe"
9
10     def __init__(self, usuario, password):
11         """Método que asigna un usuario y contraseña inicial."""
12         self.usuario = usuario
13         self.password = password
14         print "*click*... Dispositivo armado."
15
16     def acceso(self, usuario, password):
17         """Método al que se accede a los secretos del sistema."""
18         if usuario == self.usuario and password == self.password:
19             self.__ultima_barrera()
20         else:
21             print "Acceso denegado."
22             return
23
24     def __ultima_barrera(self):
25         """Validación antes de acceder a los secretos."""
26         clave = raw_input("Ingrese código maestro: ")
27         if clave == self.__password_maestro:
28             print ("Ahora tiene acceso a nuestros secretos.")
29         else:
30             print "¡KA-BOOOM!"
31             return
```



El script *secreto.py*

```
32
33 ▼ if __name__ == "__main__":
34     .... caja_fuerte = Confidencial("Bond", "Moneypenny")
35     .... usuario = raw_input("Ingrese su usuario: ")
36     .... password = raw_input("Ingrese su contraseña: ")
37     .... caja_fuerte.acceso(usuario, password)
38     .... print ".... \n.... \n...."
39     .... hackeado = caja_fuerte._Confidencial_password_maestro
40     .... print "¡Anonymous nos acaba de hackear!"
41     .... print "El código maestro es '%s'." % hackeado
42     .... caja_fuerte._Confidencial_ultima_barrera()
```



El script *secreto.py*

```
*click*... Dispositivo armado.  
Ingrese su usuario: Bond  
Ingrese su contraseña: Moneypenny  
Ingrese código maestro: 45345  
¡KA-B000M!  
...  
...  
...  
¡Anonymous nos acaba de hackear!  
El código maestro es ''123qwe''.  
Ingrese código maestro: 123qwe  
Ahora tiene acceso a nuestros secretos.
```



El script *secreto.py* en el entorno interactivo

```
>>> from secreto import Confidencial
>>> dir(Confidencial)
['_Confidencial__password_maestro', '_Confidencial__ultimaBarrera', '__class__',
 '__delattr__', '__dict__', '__doc__', '__format__', '__getattr__', '__hash__
__', '__init__', '__module__', '__new__', '__reduce__', '__reduce_ex__', '__repr__
__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', '__weakref__',
 'acceso']
>>> █
```

- Los atributos y métodos escondidos en realidad sólo ofuscan el modo en el que se puede acceder a ellos.

<objeto>._<Clase>__,nombre>



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*

```
1  #!/usr/bin/python
2  # -*- coding: utf-8 -*-
3
4  import math
5
6
7  class Forma(object):
8
9      def __init__(self):
10         pass
11
12     def superficie(self):
13         pass
14
15     def perimetro(self):
16         pass
17
18
19  class Circulo(Forma):
20
21      def __init__(self, radio=1):
22         self.radio = radio
23
24     def superficie(self):
25         return math.pi * self.radio ** 2
26
27     def perimetro(self):
28         return math.pi * 2 * self.radio
```

TELLIGENCE



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 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:

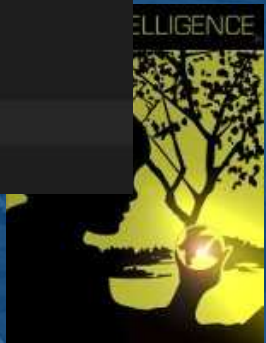
```
class <SuperClase>:  
    def <metodo>  
    ...  
    ...  
class <SubClase>(<SuperClase>):  
    def <metodo>  
        super().<metodo>  
    ...
```



secreto_heredado.py

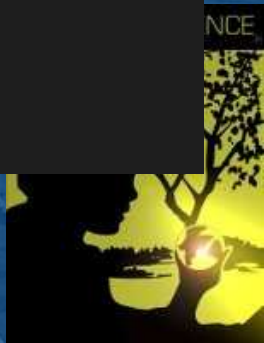
```
1 # -*- coding: utf-8. -*-
2 #!/usr/bin/python
3 """Módulo que contiene dos clases con atributos de acceso restringido."""
4
5
6 class Confidencial(object):
7     """Esta clase crea objetos con una contraseña maestra."""
8     __password_maestro = "123qwe"
9
10     def __init__(self, usuario, password):
11         """Método que asigna un usuario y contraseña inicial."""
12         self.usuario = usuario
13         self.password = password
14         print "*click*... Dispositivo armado."
15
16     def acceso(self, usuario, password):
17         """Método al que se accede a los secretos del sistema."""
18         if usuario == self.usuario and password == self.password:
19             self.__ultima_barrera()
20         else:
21             print "Acceso denegado."
22             return
23
24     def __ultima_barrera(self):
25         """Validación antes de acceder a los secretos."""
```

ELLIGENCE



secreto_heredado.py

```
26 ..... clave = raw_input("Ingrese código maestro: ")
27 ▾ ..... if clave == self.__password_maestro:
28 .....     print ("Ahora tiene acceso a nuestros secretos.")
29 ▾ ..... else:
30 .....     print "¡KA-BOOOM!"
31 .....     return
32
33
34 ▾ class Doble_Seguridad(Confidencial):
35 .....     """Sistema con doble factor de seguridad. Subclase de Confidencial."""
36 .....     __password_maestro = "123qwe"
37
38 ▾ ..... def __init__(self, nombre, usuario, password_1, password_2):
39 .....     self.segundo_factor = password_2
40 .....     print "Bienvenido a %s." % nombre
41 .....     super(Doble_Seguridad, self).__init__(usuario, password_1)
42
43 ▾ ..... def acceso(self, usuario, password_1, password_2):
44 .....     """Método al que se accede a los secretos del sistema con doble factor
45 .....     de seguridad."""
46 ▾ .....     if usuario == self.usuario and password_1 == self.password and \
47 .....     password_2 == self.segundo_factor:
48 .....         self.__ultima_barrera()
49 ▾ .....     else:
50 .....         print "Acceso denegado."
```



secreto_heredado.py

```
51 ..... return
52
53 ▾ ..... def __ultima_barrera(self):
54 ..... """Validación antes de acceder a los secretos.. Este método no pudo ser
55 ..... heredado automáticamente debido al "name mangling"""
56 ..... clave = raw_input("Ingrese código maestro: ")
57 ▾ ..... if clave == self.__password_maestro:
58 .....     print("Ahora tiene acceso a nuestros secretos.")
59 ▾ ..... else:
60 .....     print("¡KA-BOOOM!")
61 .....     return
62
63 |
64 ▾ if __name__ == "__main__":
65 ..... palacio = Doble_Seguridad("Palacio de Buckingham", "Bond", "Moneypenny",
66 .....     "Skyfall")
67 ..... usuario = raw_input("Ingrese su usuario: ")
68 ..... password = raw_input("Ingrese su contraseña: ")
69 ..... segundo_factor = raw_input("Ingrese segundo factor de seguridad: ")
70 ..... palacio.acceso(usuario, password, segundo_factor)
```



secreto_heredado.py

```
Bienvenido a Palacio de Buckingham.  
*click*... Dispositivo armado.  
Ingrese su usuario: Bond  
Ingrese su contraseña: Moneypenny  
Ingrese segundo factor de seguridad: Skyfall  
Ingrese código maestro: 123qwe  
Ahora tiene acceso a nuestros secretos.
```



secreto_heredado.py

Help on module secreto_heredado:

NAME

secreto_heredado - Módulo que contiene dos clases con atributos de acceso restringido.

FILE

/home/josech/Documentos/Cursos/Contenidos/python/codigo/secreto_heredado.py

CLASSES

__builtin__.object

Confidencial

Doble_Seguridad

class Confidencial(__builtin__.object)

| Esta clase crea objetos con una contraseña maestra.

| Methods defined here:

| __init__(self, usuario, password)

| Método que asigna un usuario y contraseña inicial.

| acceso(self, usuario, password)

| Método al que se accede a los secretos del sistema.

*| -----
| Data descriptors defined here:*

| __dict__

| dictionary for instance variables (if defined)

| __weakref__

| list of weak references to the object (if defined)



secreto_heredado.py

```
class Doble_Seguridad(Confidencial)
|   Sistema con doble factor de seguridad. Subclase de Confidencial.
|
|   Method resolution order:
|       Doble_Seguridad
|       Confidencial
|       __builtin__.object
|
|   Methods defined here:
|
|   __init__(self, nombre, usuario, password_1, password_2)
|
|   acceso(self, usuario, password_1, password_2)
|       Método al que se accede a los secretos del sistema con doble factor
|       de seguridad.
|
|   -----
|   Data descriptors inherited from Confidencial:
|
|   __dict__
|       dictionary for instance variables (if defined)
|
|   __weakref__
|       list of weak references to the object (if defined)
```

(END)



Herencia Múltiple

- Python permite la herencia múltiple.
- Los atributos y métodos de la superclase que es referida primero al definir la subclase tienen precedencia sobre los atributos y métodos las superclases siguientes.



herencia_multiple.py

```
1  #!/usr/bin/python
2  # -*- coding: utf-8 -*-
3
4
5  class Animal(object):
6
7      def __init__(self, nombre):
8          print "El animal %s acaba de nacer" % nombre
9          self.nombre = nombre
10
11     def reproduccion(self):
12         pass
13
14
15     class Mamifero(Animal):
16
17         def produce_leche(self):
18             print "Aquí hay un poco de leche."
19
20         def reproduccion(self, tiempo):
21             self.tiempo_gestacion = tiempo
22             print "Después de %d meses nacen las crías." % self.tiempo_gestacion
23
24
```



herencia_multiple.py

```
25 class Reptil(Animal):
26
27     def produce_veneno(self, venenoso):
28         self.veneno = venenoso
29         if venenoso:
30             print("Soy venenoso.")
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 venenoso.*



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__', '__getattribute__', '__hash__', '__init__', '__module__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', '__weakref__']
>>> objeto.saludo = funcion
>>> objeto.saludo()
'Hola'
>>> dir(objeto)
['__class__', '__delattr__', '__dict__', '__doc__', '__format__', '__getattribute__', '__hash__', '__init__', '__module__', '__new__', '__reduce__', '__reduce_ex__', '__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.
- El 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*

```
1  #!/usr/bin/python
2  # -*- coding: utf-8 -*-
3  '''Script que ilustra el uso de duck typing'''
4
5
6  class CapitalesMundiales(object):
7      ... capitales = {
8          ... "México": "Distrito Federal",
9          ... "Argentina": "Buenos Aires",
10         ... "Uruguay": "Montevideo",
11         ... "Brasil": "Sao Paulo",
12         ... "Estados Unidos": "Washington, D.C."
13     ... }
14
15     ... def __init__(self, pais="México"):
16         ... self.pais = pais
17
18     ... def capitalize(self):
19         ... if self.pais in self.capitales:
20             ... return self.capitales[self.pais]
21         ... else:
22             ... return 'País desconocido'
23
24
25 def capital(objeto):
26     ... return objeto.capitalize()
27
28 pais = CapitalesMundiales("Bolivia")
29 mensaje = "HOLA"
30
31 print capital(pais)
32 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 procurando 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*

```
1  #!/usr/bin/python
2  # -*- coding: utf-8 -*-
3  """Script que ilustra la persistencia de objetos mediante el uso
4  del módulo pickle"""
5
6  import pickle
7  lista = [[1, 2, 3], [4, 5, 6]]
8  """Se guarda el objeto"""
9  with open("objeto.bin", "wb") as archivo:
10 ... pickle.dump(lista, archivo)
11 """Se recupera el objeto"""
12 with open("objeto.bin", "rb") as archivo:
13 ... otra_lista = pickle.load(archivo)
14 print "La lista es:", lista
15 print id(lista)
16 print id(otra_lista)
17 if lista == otra_lista:
18 ... print "Estas listas son idénticas."
19 else:
20 ... print "Estas listas no son iguales."
```



DISTRIBUCIÓN DE SOFTWARE



Distribución de código mediante distutils

- El módulo distutils permite "empacar" el código de un proyecto de software para ser redistribuido.
- La configuración del paquete se realiza en el archivo *setup.py*



setup.py

```
1  #!/usr/bin/python
2  # -*- encoding: utf-8 -*-
3
4  from distutils.core import setup
5  setup(name="Control",
6        version="0.1",
7        description="Caso de Estudio",
8        author="José Luis Chiquete",
9        author_email="josech@gmail.com",
10       url="http://twitter/josech",
11       license="GPL",
12       packages=["caso"]
13  )
```



Algunos datos en setup.py

- name="Control",
- version="0.1",
- description="Caso de Estudio",
- author="José Luis Chiquete",
- author_email="josech@gmail.com",
- url="<http://twitter/josech>",
- download_url="http://openintelligence.mx/descargas"
- license="GPL",
- packages=["caso"]
- py_modules=["altas.py"]



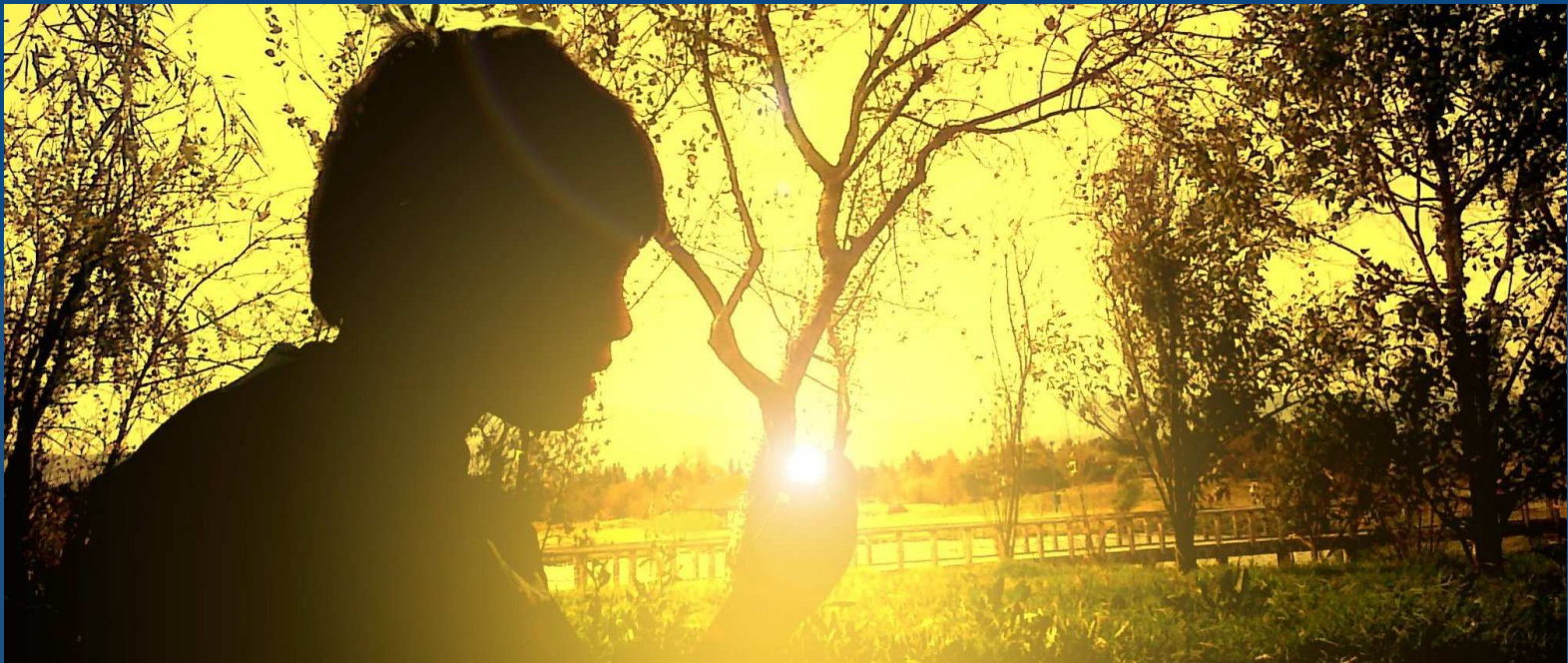
Ejecución de setup.py

```
josech@oi-x230:~/Documentos/Cursos/Contenidos/python/codigo$ python setup.py sdist --formats=zip
running sdist
running check
warning: sdist: manifest template 'MANIFEST.in' does not exist (using default file list)

warning: sdist: standard file not found: should have one of README, README.txt

writing manifest file 'MANIFEST'
creating Control-0.1
creating Control-0.1/caso
making hard links in Control-0.1...
hard linking setup.py -> Control-0.1
hard linking caso/__init__.py -> Control-0.1/caso
hard linking caso/altas.py -> Control-0.1/caso
hard linking caso/clase.py -> Control-0.1/caso
hard linking caso/datos.py -> Control-0.1/caso
hard linking caso/listado.py -> Control-0.1/caso
hard linking caso/valida.py -> Control-0.1/caso
creating 'dist/Control-0.1.zip' and adding 'Control-0.1' to it
adding 'Control-0.1/PKG-INFO'
adding 'Control-0.1/setup.py'
adding 'Control-0.1/caso/__init__.py'
adding 'Control-0.1/caso/listado.py'
adding 'Control-0.1/caso/datos.py'
adding 'Control-0.1/caso/altas.py'
adding 'Control-0.1/caso/clase.py'
adding 'Control-0.1/caso/valida.py'
removing 'Control-0.1' (and everything under it)
```





Open Intelligence®
<http://openintelligence.mx>
info@openintelligence.mx
+52(55) 3548-1400

Algunos derechos reservados. Esta obra puede ser modificada, redistribuida y comercializada bajo los términos de la licencia Atribución 2.5 México (CC BY 2.5)
<http://creativecommons.org/licenses/by/2.5/mx/>

