


Curso Python en 8 clases

Clase 6: Introduccion a POO

Autor: Sebastián Bassi  sbassi@gmail.com

Versión: 1.4

Licencia: Creative Commons BY-NC-SA 2.5. ([ver texto completo](#))



Introducción

Si bien Python es un lenguaje orientado a objetos (Programación Orientada a Objetos, POO), es posible utilizarlo sin tener en cuenta explícitamente este hecho. Por eso se dice que Python es un lenguaje "Multi paradigma". Los objetos ya han sido usados de manera implícita: Los tipos de datos (str, dict, listas) son objetos. Cada uno tiene funciones asociadas (llamadas métodos en la jerga) y sus atributos. Por ejemplo `str.lower()` devuelve un str en minúscula, porque todos los objetos de esta clase (str) tienen el método `lower()` asociados.

Una clase se puede usar para definir un tipo de datos, y si bien los tipos de datos incluidos en Python son variados, su capacidad para hacer representaciones (modelos) del mundo real es limitada. Es por eso que hace falta poder definir nuestras propias clases.

Jerga de POO

Clases: Generadores de objetos

Es una estructura que se usa como plantilla para crear objetos (de una clase). Esta plantilla describe tanto el estado como el comportamiento de los objetos que se crean a partir de ella. El estado es mantenido via los *atributos* y el comportamiento via los *métodos*.

Instancia: Implementación particular de una clase

Cada instancia es una implementación de una clase. Si tenemos la clase *orca*, una instancia puede ser *Willy*. Es posible crear varias instancias independientes una de otra (como *Shamu* que es independiente de *Willy*).

Atributos (o variables de instancia): Características de los objetos

Cada objeto tiene sus atributos, como por ejemplo *peso*. *Willy* tendrá un peso distinto de *Shamu*. Aunque ambas instancias pertenecen a la misma clase (*orca*). Comparten el tipo de atributo. Podríamos tener la clase *perro* con instancias *Lassie* y *Laika* con el atributo *color_de_pelo* que no será compartido con las instancias de la clase *orca*.

Métodos: Comportamiento de los objetos

Un método es una función que pertenece a una clase. Los métodos definen como se portan los objetos derivados de una clase. La clase *perro* puede tener el método *ladrar*. Algunos métodos pueden usar parámetros (como *ladrar(enojado)*).

Herencia: Las propiedades se transmiten de clases relacionadas

Las clases pueden relacionarse entre ellas y no ser entidades aisladas. Es posible tener una clase *mamífero* con propiedades comunes a la clase *orca* y la clase *perro*. Por ejemplo el método *reproducción* puede ser definido para la clase *mamífero*. No será necesario crear para ellos el método *reproducción* ya que lo heredarán.

Variable de clases: características de las clases

Son variables asociadas a todos los objetos de una clase. Cuando un objeto es creado, el objeto hereda la variable de clase.

Polimorfismo

Es la habilidad para los distintos objetos de responder al mismo método de manera distinto. Es lo que permite por ejemplo al iterar sobre una lista, un set, diccionario, archivo, etc de la misma manera:

```
for nombre in secuencia:
    # hacer algo
```

Encapsulación

Ocultar la operación interna de un objeto y dar acceso a los programadores por métodos públicos. La encapsulación en Python no es total, sino mas bien nominativa (también conocida como pseudo-encapsulación). Los métodos de uso interno se marcan como tales y los programadores eligen (o no) respetar.

Creando clases

Modelo:

```
class NOMBRE:
    [cuerpo]
```

Una clase que hace algo:

```
class Square:
    def __init__(self):
        self.side=1
```

Uso:

```
>>> Bob=Square() # Bob es una instancia de Square.
>>> Bob.side # Veamos el valor de side.
1
>>> Bob.side=5 # Asignar un valor nuevo a side.
>>> Bob.side # Ver el valor de side.
5
```

Este cambio es específico para una instancia, al crearse una nueva, se ejecuta `__init__` de nuevo:

```
>>> Krusty=Square()
>>> Krusty.side
1
```

Asignando una variable a la clase (y no solo a la instancia):

```
>>> class Square:
        side=1
>>> Square.side
1
```

Es posible usar las variables de clases para controlar cuantas instancias de una clase han sido creadas:

```
class Square:
    cuenta = 0
    def __init__(self):
        Square.cuenta += 1
        print "Objeto creado OK"
```

Uso:

```
>>> Bob = Square()
"Objeto creado OK"
>>> Patricio = Square()
"Objeto creado OK"
>>> Square.cuenta
2
```

Ejemplo con la clase *Secuencia*. En biología una secuencia nucleotídica es un fragmento de ADN. Por ejemplo: "ATGCA". Esto representa una secuencia de 5 nucleótidos (Adenina, Timina, Guanina, Citosina y nuevamente Adenina). Esta secuencia podría ser representada como un string:

```
secuencia = "ATGCA"
```

Esta representación no nos permite almacenar funciones asociadas a este tipo de datos (métodos). Una función propia de las secuencia es su *transcripción*, esto es: el proceso por el cual el ADN se convierte en ARN. En dicha transcripción los nucleótidos se convierten siguiendo la siguiente tabla:

ADN	ARN
A	U
T	A
C	G
G	C

Con estos datos podemos hacer un programa que define una clase con el método *transcripcion*:

```
import string

class Secuencia:
    tabla = string.maketrans('ACTG', 'UGAC')
    def __init__(self, cadena):
        self.cadena = cadena.upper()
    def transcripcion(self):
        tt = string.translate(self.cadena, self.tabla)
        return tt

miseq = Secuencia('ACAGTGTA')
print miseq.transcripcion()
```

Esto permite definir instancias de *secuencia* y luego transcribirlas:

```
>>> virus_peligroso = Secuencia('atggagagccttgcttcttggtgtcaa')
>>> virus_peligroso.cadena
'ATGGAGAGCCTTGCTTCTTGGTGTCAA'
```

```
>>> virus_peligroso.transcripcion()  
'UACCUCUCGGAACAAGAACCACAGUU'
```

Podemos agregar entonces el método `restriccion` que requiere un parámetro:

```
def restriccion(self,enz):  
    if enz in Secuencia.enz_d:  
        return self.cadena.count(Secuencia.enz_d[enz])  
    else:  
        return 0
```

Resultado:

```
>>> virus_inocuo = Secuencia('atgatatcggagaggatatcggtgtcaa')  
>>> virus_inocuo.restriccion('EcoRV')  
2
```

Herencia en acción

La herencia de una clase implica que una *clase hija* hereda los métodos y atributos de la clase padre. Por ejemplo existe un tipo de secuencia de ADN denominada plásmido, que son independientes del ADN principal (cromosómico) de la célula. Esto puede ser modelado como una clase nueva (Plasmido) que tiene las mismas propiedades de `Secuencia`, mas métodos propios. En este caso se dice que la clase `Plasmido` hereda de `Secuencia`.

Uno de los métodos que son propios de los plásmidos es la resistencia a antibioticos. El método `ab_res` verifica si un plasmido particular tiene o no resistencia a un antibiotico:

```
class Plasmido(Secuencia):  
    ab_res_d = {"Tet": "CTAGCAT", "Amp": "CACTACTG"}  
    def __init__(self,cadena):  
        Secuencia.__init__(self,cadena)  
    def ab_res(self,ab):  
        if ab in self.ab_res_d and self.ab_res_d[ab] in self.cadena:  
            return True  
        else:  
            return False
```

Resultado:

```
>>> x5 = Plasmido('TACGTCACACTAG')  
>>> x5.ab_res('Amp')  
True
```

Métodos especiales

Algunos métodos se ejecutan bajo condiciones pre-establecidas. Se denota por un doble guión bajo antes y despues del nombre. Por ejemplo el método `__init__`, que es ejecutado cuando se crea una instancia nueva. Otros métodos especiales se ejecutan solo bajo determinadas circunstancias, lo que modificamos es como se comporta el objeto ante dicha condición.

Por ejemplo el método `__len__` se activa cuando se llama la función `len()`. Lo que el método devuelve depende del programador. Si ahora hacemos un `len(secuencia)` tenemos un error:

```
>>> len(miseq)
Traceback (most recent call last):
  File "<pyshell#25>", line 1, in <module>
    len(miseq)
AttributeError: Secuencia instance has no attribute '__len__'
```

Por lo que si queremos que ese tipo de objeto devuelva algo al aplicarle *len*, tenemos que definir el método especial `__len__`:

```
def __len__(self):
    return len(self.cadena)
```

Ahora si podemos aplicar la función *len*:

```
>>> M13 = Secuencia("ACGACTCTCGACGGCATCCACCCTCTCTGAGA")
>>> len(M13)
32
```

Algunos métodos especiales

- `__str__`: Se invoca cuando hace falta una representación en string de un objeto. Esta representación se obtiene con *str(objeto)*.
- `__repr__`: Se invoca con la función *repr()*. Es una representación imprimible del objeto.
- `__getitem__`: Para acceder a un objeto secuencialmente o usando un subíndice tipo *objeto[n]*. Este método requiere 2 parametros.
- `__iter__`: Permite iterar sobre una secuencia, como cuando se usa *for secuencia*.
- `__setitem__`: Para asignarle el valor a una clave.
- `__delitem__`: Implementa el borrado de un objeto.

Para ver todos los métodos especiales: docs.python.org/reference/datamodel.html.

Ejemplo con clase "Reversa":

```
class Reversa:
    def __init__(self, data):
        self.data = data
        self.indice = len(data)
    def __iter__(self):
        return self
    def next(self):
        if self.indice == 0:
            raise StopIteration
        self.indice -= 1
        return self.data[self.indice]
```

Crear un tipo de datos en base a uno anterior

Hay que heredar el objeto original y cambiarle algún método:

```
class Zdic(dict):
    """ Objeto tipo diccionario que devuelve 0 si
```

```
se accede a una clave inexistente. """
def __missing__(self, x):
    return 0
```

Manteniendo el código privado

Si hay métodos o atributos que tienen que usarse para "consumo interno" (solo dentro de las clases), se los puede proteger de manera nominal, esto es, indicar que se los protege. No es una protección "real", pero sirve a efectos prácticos. Se trata de agregar 2 guiones bajos al principio del nombre del objeto.

Veamos una clase que define 2 métodos, *a* y *b*:

```
class ClaseTest:
    def a(self):
        pass
    def __b(self):
        # mangleado como _TestClass__b
        pass
```

De esta manera no podemos acceder a `__b`:

```
>>> MyObject = ClaseTest()
>>> MyObject.a()
>>> MyObject.__b()

Traceback (most recent call last):
  File "<pyshell#14>", line 1, in <module>
    MyObject.__b()
AttributeError: TestClass instance has no attribute '__b'
```

Es posible acceder a *a* pero no a `__b`, al menos no directamente. La notación que hay que usar es:

```
>>> MyObject._TestClass__b()
```

Algunos programadores prefieren usar solo un guión bajo (*_nombre*) y luego aclarar en la documentación que se trata de una función interna.

Recursos adicionales

- Python Programming/OOP: en.wikibooks.org/wiki/Python_Programming/OOP.
- Introduction to OOP with Python: www.voidspace.org.uk/python/articles/OOP.shtml.
- Dive into Python. Capítulo 5. diveintopython.org/object_oriented_framework.
- Python Objects: www.effbot.org/zone/python-objects.htm.
- Java Tutorial: OOP Concepts: java.sun.com/docs/books/tutorial/java/concepts.

Agradecimientos

- Roberto Alsina (rst, inspiración de rst y otros)
- Lucio Torre (errores de todo tipo)
- Alejandro Santos (código de ejemplo en polimorfismo y sugerencia sobre código privado)

- Rafael Moyano (typo)