Programación orientada a objetos - Parte 2

La idea detrás de la OOP es modelar objetos del mundo real. Según la teoría de la OOP, todos los objetos a modelar tienen tres caracteristicas:

- Su propia referencia
- Atributos
- Acciones

Pongamos como ejemplo que quiero crear una mascota virtual, un perro llamado "Boby". "Boby" no será cualquier perro, será "Boby" y solo ese, es decir, esta referenciado. Puedo tener luego otra mascota llamada "Ralf" que también sera un perro, pero diferente de "Boby" (esto es bastante evidente, pero no lo será cuando lo codifiquemos).

Adicionalmente, un perro tiene ciertos atributos: tiene un nombre, tiene un peso y tiene una raza. Aunque existen mas atributos para un perro, establezcamos que para nuestro perro virtual estos tres atributos son suficientes.

Para finalizar, un perro realiza ciertas acciones, como caminar, correr, dormir y ladrar. Ciertamente puede morder, pero dejemos como en el caso anterior la lista de acciones reducida.

En la terminología de OOP, los atributos se llaman *Fields* y las acciones se llaman *Métodos*. Tanto "Boby" como "Ralf" serán *objetos* que se crear a partir de una plantilla llamada *clase* que especifica los atributos, acciones y un mecanismo para que el objeto de referencie a si mismo. El objeto "Boby" será una *instancia* de la clase "Perro".

Por lo tanto, toda la OOP gira en torno a la definicion de un clase. Una clase se crea con la siguiente nomenclatura:

```
class NombreClase():
    # Inicializacion del objeto



    # Metodos
```

Para crear la clase Perro se puede iniciar con la siguiente estructura:

```
class Perro():
    def __init__(self, nombre, peso, raza):
        self.__nombre = nombre
        self.__peso = peso
        self.__raza = raza
```

Dentro de la defincion de clase lo primero que se encontrara será un método especial llamado "__init__". Este metodo es llamado cada vez que se instancia un objeto. Este metodo se encarga de inicializar el objeto con los atributos especificos. De forma tal que para crear a "Boby" se tendrá que ejecutar la instrucción:

```
Boby = Perro("Boby", 20, "Pastor Aleman")
```

¿Qué paso con el parámetro *self*? Ese paramtero no se indica en el momento de la instanciación de un objeto. ¿Entonces para que esta? Para referenciar al objeto como si mismo.

Esta idea suele ser la primera complicación en la OOP. "self" se puede entender como si fuera la espcificación de "yo", de tal forma que en el momento de la inicializacion se toman los atrbutos y se asignan de la forma:

```
def __init__(self, nombre, peso, raza):      -> Yo soy un perro que tiene un nombre, un peso y
una raza
    self.__nombre = nombre      -> Yo me llamo "Boby"
    self.__peso = peso          -> Yo peso 20 kilos
    self.__raza = raza          -> Yo pertenezco a la raza "Pastor Aleman"
```

Luego, lo que se debe de hacer es definir las acciones de un perro. Estas serán funciones (o *métodos* en la terminología de OOP). Por ejemplo:

```
def caminar(self):     -> Yo camino
    return self.nombre + " camina"

def correr(self):      -> Yo corro
    retrun self.nombre + " corre"

def ladrar(self):      -> Yo ladro
    return "Guan, guau, guau..."
```

Veamos como funciona todo el conjunto:

In [52]:

```
class Perro():
    def __init__(self, nombre = "Pluto", peso = 10, raza = "Chusco"):
        self.__nombre = nombre
        self.__peso = peso
        self.__raza = raza

    def caminar(self):
        return self.__nombre + " camina"

    def correr(self):
        return self.__nombre + " corre"

    def ladrar(self):
        return "Guau, guau, guau..."
```

Ahora vamos a instanciar el objeto Boby:

In [53]:

```
Boby = Perro("Boby", 20, "Pastor Aleman")
print("Boby ->",type(Boby))
print("Metodos ->",dir(Boby))
```

```
Boby -> <class '__main__.Perro'>
Metodos -> ['_Perro__nombre', '_Perro__peso', '_Perro__raza', '__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__format
__', '__ge__', '__getattribute__', '__gt__', '__hash__', '__init__', '__init_subclass__', '__le__', '__lt__', '__module__', '__ne__', '__new_
_', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', '__weakref__', 'caminar', 'correr',
'ladrar']
```

Como puede observar, "Boby" es un objeto y tiene un numero de métodos, entre los que estan caminar, correr y ladrar.

In [54]:

```
print(Boby.caminar())
print(Boby.correr())
print(Boby.ladrar())
```

```
Boby camina
Boby corre
Guau, guau, guau...
```

Puedo crear un objeto diferente de la misma clase, utilizando algunas propiedades por defecto:

In [55]:

```
Rolf = Perro("Rolf")
print(Rolf.caminar())
print(Rolf.correr())
print(Rolf.ladrar())
```

```
Rolf camina
Rolf corre
Guau, guau, guau...
```

Observe los dos caracteres "__". Esto indica que estos atributos son privados, por lo que no puedo acceder a ellos. Por ejemplo, Rolf no tiene raza (porque esta es la raza por defecto). Y no es posible acceder a ese atributo:

In [56]:

```
Rolf.raza
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-56-b12fd1cfcbb9> in <module>()
----> 1 Rolf.raza

AttributeError: 'Perro' object has no attribute 'raza'
```

Para acceder a este atributo, habrá que definir unos métodos especiales que permitan obtener los atributos de esta clase (en inglés, get the attributes). A estos métodos se les conoce como "getters". Completemos la clase agregandole getters:

In [57]:

```
class Perro():
    def __init__(self, nombre = "Pluto", peso = 10, raza = "Chusco"):
        self.__nombre = nombre
        self.__peso = peso
        self.__raza = raza

    # getters
    def retornaNombre(self):
        return self.__nombre

    def retornaPeso(self):
        return self.__peso

    def retornaRaza(self):
        return self.__raza

    # Metodos
    def caminar(self):
        return self.__nombre + " camina"

    def correr(self):
        return self.__nombre + " corre"

    def ladrar(self):
        return "Guau, guau, guau..."

Boby = Perro("Boby", 20, "Pastor Aleman")
Rolf = Perro("Rolf")
```

Ahora podemos obtener los atributos ambos objetos por medio de sus métodos getters:

In [58]:

```
print(Boby.retornaNombre())
print(Boby.retornaPeso())
print(Boby.retornaRaza())
print()
print(Rolf.retornaNombre())
print(Rolf.retornaPeso())
print(Rolf.retornaRaza())
```

```
Boby
20
Pastor Aleman

Rolf
10
Chusco
```

Si se quiere cambiar algun atributo de un objeto, se necesita contar con otros métodos especiales, en este caso que puedan modificar los atributos de la clase (en inglés, set de attributes). A estos métodos se les conoce como *setters*. Completemos la clase agregandole *setters*:

In [59]:

```
class Perro():
    def __init__(self, nombre = "Pluto", peso = 10, raza = "Chusco"):
        self.__nombre = nombre
        self.__peso = peso
        self.__raza = raza

    # getters
    def retornaNombre(self):
        return self.__nombre

    def retornaPeso(self):
        return self.__peso

    def retornaRaza(self):
        return self.__raza

    # setters
    def asignaNombre(self, value):
        self.__nombre = value

    def asignaPeso(self, value):
        self.__peso = value

    def asignaRaza(self, value):
        self.__raza = value

    # Metodos
    def caminar(self):
        return self.__nombre + " camina"

    def correr(self):
        return self.__nombre + " corre"

    def ladrar(self):
        return "Guau, guau, guau..."

Boby = Perro("Boby", 20, "Pastor Aleman")
Rolf = Perro("Rolf")
```

Ahora veamos si podemos modificar las opciones de Rolf:

In [60]:

```
print(Rolf.retornaRaza())

Rolf.asignaRaza("Siberiano")
Rolf.asignaPeso(40)

print(Rolf.retornaRaza())
print(Rolf.retornaPeso())
```

```
Chusco
Siberiano
40
```

Esta forma de definir una clase de clasica de la OOP: los atributos se encuentran protegidos dentro de la clase ya que no pueden ser revisados a menos que sea a través de un getter ni modificados a menos que sea a través de un setter. A esto se le denomina *encapsulamiento*. Por ejmplo, podemoss mejorar la clase protegiendo los atributos:

In [61]:

```
class Perro():
    def __init__(self, nombre = "Pluto", peso = 10, raza = "Chusco"):
        self.__nombre = nombre
        self.__peso = peso
        self.__raza = raza

    # getters
    def retornaNombre(self):
        return self.__nombre

    def retornaPeso(self):
        return self.__peso

    def retornaRaza(self):
        return self.__raza

    # setters
    def asignaNombre(self, value):
        if isinstace(value, str):
            self.__nombre = value
        else:
            raise ValueError(str(value) + " no es un tipo de atributo valido. Debe ser un str")

    def asignaPeso(self, value):
        if isinstance(value, int):
            if value <= 0:
                raise ValueError (str(value) + " no es un tipo de atributo valido. Debe ser int mayor q
ue 0")
            else:
                self.__peso = value
        else:
            raise ValueError(str(value) + " no es un tipo de atributo valido. Debe ser un int")

    def asignaRaza(self, value):
        if isinstance(value, str):
            self.__raza = value
        else:
            raise ValueError(str(value) + " no es un tipo de atributo valido. Debe ser un str")

    # Metodos
    def caminar(self):
        return self.__nombre + " camina"

    def correr(self):
        return self.__nombre + " corre"

    def ladrar(self):
        return "Guau, guau, guau..."

Rolf = Perro("Rolf")
```

Ahora pruebe a crear diferentes perros con diferentes atibutos:

In [62]:

```
# Por ejemplo, un perro con -10 kg de peso
Rolf.asignaPeso(-10)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-62-114caef11f11> in <module>()
      1 # Por ejemplo, un perro con -10 kg de peso
----> 2 Rolf.asignaPeso(-10)

<ipython-input-61-68bc9d073b09> in asignaPeso(self, value)
     25         if isinstance(value, int):
     26             if value <= 0:
---> 27                 raise ValueError (str(value) + " no es un tipo de atributo valido. Debe ser int mayor que 0")
     28             else:
     29                 self.__peso = value

ValueError: -10 no es un tipo de atributo valido. Debe ser int mayor que 0
```

In [63]:

```
# O un perro cuya raza sea un valor numérico
Rolf.asignaRaza(0)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-63-e3b86c132018> in <module>()
      1 # O un perro cuya raza sea un valor numérico
----> 2 Rolf.asignaRaza(0)

<ipython-input-61-68bc9d073b09> in asignaRaza(self, value)
     35             self.__raza = value
     36         else:
---> 37             raise ValueError(value + " no es un tipo de atributo valido. Debe ser un str")
     38
     39     # Metodos

TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

In [64]:

```
# Y responder correctamente a la asignación de atributos correctos
Rolf.asignaPeso(40)
Rolf.asignaRaza("Chiwawa")
```

Sin embargo, existe un problema: estas validaciones solo sirven en el momento de asignar los aributos por medio de los setters y no en el momento de instanciar un objeto:

In [65]:

```
Junior = Perro("Junior", -5, 10)
print(Junior.retornaRaza())
```

```
10
```

Esto se debe al hecho de que los atributos de la clase Perro son *publicos* y no *privados*. En lenguajes de programación como C++ o Java, existe una distinción entre atributos púbicos y privados, donde estos últimos solo pueden tener valores a través de un setter. Sin embargo, la implementación de la OOP es Python es menos estricta y mas laxa.

Hay una forma de aproximarse a la programación OOP en Python: la forma *pythonica* de la OOP. En esta todos los atributos son publicos pero con un control de asignación. Y no hay necesidad de getters ya que al ser los atributos publicos se puede acceder a ellos directamente de la forma *objeto.atributo*.

Pero esto se verá en la siguiente parte.