Programación orientada a objetos - Parte 3

En la Parte 2 introducimos las ideas de la OOP en su forma clasica: clase, inicilización, atributos privados, metodos setters, metodos getters y metodos.

Sin embargo, en Python la OOP tiene un enfoque menos restrictivo y mas laxo. Para empezar, todos los atributos son publicos. ¡Lo cual es una ventaja enorme cuando se trata de hacer que el código sea fácil de leer (y recuerde, en Python la simplicidad esta por encima de todo). Imaginemos que por alguna razon quiere sumar el peso de los dos perros que tiene, Boby y Rolf. Utilizando la definicion clasica de OOP sera necesario requerir a los setters para obtener los pesos, por lo que la operación para realizar este calculo será:

```
peso_perros = Boby.retornaPeso() + Rolf.retornaPeso()
```

Ugh... No sería más natural hacerlo asi:

```
peso_perros = Boby.peso + Rolf.peso
```

Es decir, sumando los atributos en lugar de los resultados de unas funciones que retornan los atributos. Esto se logra porque los atributos en Python siempre son publicos.

Entonces, simplifiquemos nuestra clase Perro:

In [4]:

```
class Perro():
    def __init__(self, nombre = "Pluto", peso = 10, raza = "Chusco"):
        self.nombre = nombre
        self.peso = peso
        self.raza = 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", 40, "Pastor Aleman")
Rolf = Perro("Rolf", raza = "Doberman", peso = 20)   # Se puede utilizar el nombre de los atributos
```

Ahora podemos acceder a los artibutos de los objetos directamente:

In [6]:

```
print("Mascota 1:", Boby.nombre, " Peso:", Boby.peso)
print("Mascota 2:", Rolf.nombre, " Peso:", Rolf.peso)
print("Suma de los pesos:", Boby.peso + Rolf.peso, " kg")
```

```
Mascota 1: Boby  Peso: 40
Mascota 2: Rolf  Peso: 20
Suma de los pesos: 60  kg
```

Ademas de hacer mas sencillas ciertas operaciones. Confeccionemos una lista de perros (una lista de objetos clase Perro):

In [10]:

```
lista_perros = []
lista_perros.append(Perro("Boby", 30, "Pastor Aleman"))
lista_perros.append(Perro("Rolf", 25, "Doberman"))
lista_perros.append(Perro("Junior", 10, "Chow chow"))
lista_perros.append(Perro("Bolt", 15, "Beagle"))
# Se muestra que lista_perros es una lista de objetos clase Perro
print(lista_perros)
```

```
[<__main__.Perro object at 0x000001B00E8A9AC8>, <__main__.Perro object at 0x000001B00E8A9A58>, <__main__.Perro object at 0x000001B00E8A9908>, <__main__.Perro object at 0x000001B00E8C64E0>]
```

¿Cómo puedo saber que perros empiezan con la letra 'B'?

In [13]:

```
index = 1
for perro in lista_perros:
    if perro.nombre[0] == 'B':
        print(index, "-", perro.nombre)
        index += 1
```

```
1 - Boby
2 - Bolt
```

Utilizando los getters esto quedaría:

```
index = 1
for perro in lista_perros:
    if perro.obtieneNombre[0] == 'B':
        print(index, "-", perro.obtieneNombre())
        index += 1
```

Ugh...

Yo no son necesarios los setters tampoco ya que al ser publicos los atributos, se puede acceder directamente a ellos:

In [14]:

```
print(Boby.raza)
Boby.raza = "Labrador"
print(Boby.raza)
```

```
Pastor Aleman
Labrador
```

Y esto nos regresa al problema de la razon de ser de los setters: validar que los atributos tengan valores correctos. Python tiene una solución elegante para esto: el "decorador" @property. El código puede ser confuso al principio pero hay que tener en cuenta la forma como funciona el método @property y la nomenclatura:

- La palabra "@property" se coloca antes de crear un getter y este tendra el mismo nombre que el atributo a proteger.
- La palabra "@atributo.setter" se coloca antes de crear un setter con el nombre del atributo y se especifican las validaciones

Modifiquemos la clase anterior y vamos a validar los datos:

In [53]:

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

    # El getter
    @property
    def nombre(self):
        return self._nombre

    #El setter
    @nombre.setter
    def nombre(self, value):
        if isinstance(value, str):
            self._nombre = value
        else:
            raise ValueError(str(value) + " no es un tipo de atributo valido. Debe ser un str")

    # El getter
    @property
    def peso(self):
        return self._peso

    #El setter
    @peso.setter
    def peso(self, value):
        if isinstance(value, int):
            self._peso = value
        else:
            raise ValueError(str(value) + " no es un tipo de atributo valido. Debe ser un str")

    # El getter
    @property
    def raza(self):
        return self._raza

    #El setter
    @raza.setter
    def raza(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"
```

Ahora probemos (tanto al momento de instanciar un objeto nuevo como al momento de modificar un atributo):

In [54]:

```
Junior = Perro("Junior", 30, 10)      # La raza no es correcta
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-54-925e785d6cef> in <module>()
----> 1 Junior = Perro("Junior", 30, 10)

<ipython-input-53-dc94e8ace092> in __init__(self, nombre, peso, raza)
      3         self.nombre = nombre
      4         self.peso = peso
----> 5         self.raza = raza
      6
      7     # El getter

<ipython-input-53-dc94e8ace092> in raza(self, value)
     42             self._raza = value
     43         else:
---> 44             raise ValueError(str(value) + " no es un tipo de atributo valido. Debe ser un str")
     45
     46

ValueError: 10 no es un tipo de atributo valido. Debe ser un str
```

In [57]:

```
Junior = Perro("Junior", 30)      # Se define un peso inicial de 30 kg
Junior.peso = '10'                # El peso debe ser un int y no un str
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-57-6cebec28c6b6> in <module>()
      1 Junior = Perro("Junior", 30)
----> 2 Junior.peso = '10'

<ipython-input-53-dc94e8ace092> in peso(self, value)
     29             self._peso = value
     30         else:
---> 31             raise ValueError(str(value) + " no es un tipo de atributo valido. Debe ser un str")
     32
     33     # El getter

ValueError: 10 no es un tipo de atributo valido. Debe ser un str
```

¿Como funciona @property? Veamos el caso del atributo nombre:

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

# El getter
@property
def nombre(self):
    return self._nombre

#El setter
@nombre.setter
def nombre(self, value):
    if isinstance(value, str):
        self._nombre = value
    else:
        raise ValueError(str(value) + " no es un tipo de atributo valido. Debe ser un str")
```

Cuando se instancia el objeto de la forma Junior = Perro("Junior"), se llama al metodo "__init__" donde se ejecuta la instrucción self.nombre = nombre. Esto debería de cargar en el atributo self.nombre el valor "Junior"; sin embargo, *nombre* además de ser un atributo, también es un método. Entonces cuando se inicializa el valor de nombre, se llama al método *def nombre(self, value)* debajo de @nombre.setter. Esto, luego de validar el dato, lo asigna a self.*nombre. Note el caracter '*" antes de nombre: es una variable diferente a la que se asigna en "__init__".

Es la misma variable que se consulta cuando se llama al atributo de la forma Junior.nombre, ya que en este caso se llama a la función *def nombre(self)* debajo de @property, que retorna el valor de self._nombre.

Por lo tanto, el atributo *.nombre* es su descripción desde fuera de la clase y el atributo *._nombre* es su descripcion dentro de la clase. Por lo tanto se tienen los getters y setters originales en una construcción más sencilla, manteniendo los atributos publicos para mantener un código más legible.

Esta es la forma de hacer un clase en Python. The Python Way!