Programación orientada a objetos - Parte 4

Existen algunos detalles adicionales a considerar cuando se trata de objetos y clases. Retomemos la clase Perro, pero esta vez agregemos la palabra *object* como parametro en la definición de la clase:

In [3]:

```
class Perro(object):
    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..."

Pluto = Perro()    # Objeto con los atributos por defecto
print(Pluto)
```

```
<__main__.Perro object at 0x000002C273BEB898>
```

Cuando se imprime el objeto, lo que se tiene es una descripcion que indica que existe un objeto en una dirección de memoria. Cuando se invoca la función *print* sobre un objeto realiza lo que este definido dentro de un método *privado* llamado "__str__". Este método es *heredado* de una clase *padre*. Esta clase se llama *object*. En la línea:

```
class Perro(object):
```

Lo que sucede es que se crea un objeto utiliazando la clase *object* como plantilla base. Esto es lo que se llama "nuevo-estilo" de creación de clases en Python. Si no se declara en la forma class Perro() por defecto se asume que se utilizara *object* como clase de referencia. *object* es la raiz de todas las clases. Es un poco confuso: una busqueda en Internet de porque colocar *object* arrojara diferentes respuestas... Hay que recordar que entre el Zen de Python (import this), explicito es major que implicito, y por lo tanto es preferible explicitar *object* que asumir que este valor se cargara.

Lo importante a recordar es que hay ciertos métodos *privados* como __init__. No se puede llamar a este método desde fuera de la clase. Cuando se ejecuta la instrucción:

```
Pluto = Perro()
```

se esta llamando al método __init__. Así tambien, cuando se ejecuta la instrucción:

```
print(Pluto)
```

se esta llamando al método __str__ (realmente se esta ejecutando la instruccion *Pluto.__str__()*). ¿De donde apareció este metodo ya que no esta definido dentro de la clase? Fue *heredado* de la clase *object* donde si esta definido. Todas las clases utilizan a *object* como base para su definición por lo que todos los objetos responden algo ante la función print. Por ejemplo:

In [4]:
```
mapa = map(lambda x: x*2, [1, 2, 3, 4])
print(mapa)
```

```
<map object at 0x000002C273BEBFD0>
```

Un mapa también es un objeto y responde algo muy parecido al objeto Perro. En cambio:

In [5]:
```
lista = [1, 2, 3, 4]
print(lista)
```

```
[1, 2, 3, 4]
```

El objeto lista responde con los elementos de la lista. Dentro de la definición de ambas clases existe el método privado __str__, pero este realiza acciones diferentes. A esto se le llama *sobrecarga*; es decir, cuando un método retorna resultados diferentes en función del atributo de entrada (en este caso, según la clase a la corresponda un objeto). Podemos cambiar nuestra clase para que tenga una respuesta especial ante la impresion:

In [7]:

```
class Perro(object):
    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..."


    # Metodo privado de impresion
    def __str__(self):
```

```
        return "Perro[nombre={}, peso={}, raza={}]".format(self.nombre, self.peso, self.raza)


Pluto = Perro()    # Objeto con los atributos por defecto
print(Pluto)
```

```
Perro[nombre=Pluto, peso=10, raza=Chusco]
```

Ahora se tiene una impresión más informativa.

Todas las clases en Python utilizan la clase *object* como referencia (es bastante desafortunado que la clase raíz se llame object, confundiendo las nociones de clase y objeto). Por lo tanto todas las clases en Python son hijas de la clase padre *object* y por lo tanto heredan los atributos y métodos de *object*. Se puede crear una clase que herede los atributos y metodos de otra clase (ademas de *object* que siempre es la clase padre así no se especifique). Por ejemplo, si se quiere crear una clase para especificar gatos:

```
class Gato():
```

No tiene mucho sentido volver a definir toda la clase nuevamente; se puede utilizar la clase Perro como referencia:

In [12]:

```
class Gato(Perro):
    def __init__(self, nombre, peso, raza, vidas = 7):
        super().__init__(nombre, peso, raza)
        self.vidas = vidas
```

En este caso se ha creado la clase Gato utilizando como referencia la clase Perro. La clase Gato tiene un atributo adicional (vidas) que se asigna de la misma como se ha visto anteriormente, pero los atributos anteriores se asigan utilizando la clase Pero (con la instrucción *super().**init**(atributos)*).

In [13]:

```
Michi = Gato("Silvestre", 6, "Siames", 7)
print(Michi)
```

```
Perro[nombre=Silvestre, peso=6, raza=Siames]
```

Como se puede observar, la impresión del objeto Michi devolvio una respuesta igual a la impresión del objeto Pluto porque la clase Gato heredo el método __str__ de la clase Perro. Sin embargo, el atributo *vidas* no se muestra en la impresión, por lo que podemos sobrecargarlo:

In [14]:

```
class Gato(Perro):
    def __init__(self, nombre, peso, raza, vidas = 7):
        super().__init__(nombre, peso, raza)
        self.vidas = vidas

    def __str__(self):
        return "Perro[nombre={}, peso={}, raza={}, vidas={}]".format(self.nombre, self.peso, self.raza,
self.vidas)


Michi = Gato("Silvestre", 6, "Siames", 7)
print(Michi)
```

```
Perro[nombre=Silvestre, peso=6, raza=Siames, vidas=7]
```

Y ademas ha heredado los metodos de Perro:

In [15]:

```
print(Michi.caminar())
print(Michi.correr())
```

```
Silvestre camina
Silvestre corre
```

Esto ahorra mucho codigo y es una de las principales ventajas de la OOP... aunque hay que tener muy claro como se van a relacionar las clases ya que se puede sobrecargar un método pero no se puede evitar que un método de una clase padre sea heredada por una clase hija.

In [17]:

```
print(Michi.ladrar())
```

```
Guau, guau, guau
```

Ugh... Esto es el resultado de un muy mal diseño de clases y no hay manera de corregirlo (se puede sobrecargar el método pero este no dejará de llamarse "ladrar"). La OOP tiene que ver más con el diseño que con la codificación.

La OOP es un tema muy amplio, extenso y complejo (con terminos cada vez más extraños como herencias multiples, polimorfismo, etc.). Sin embargo, lo expuesto hasta ahora es suficiente como para volverlo competente en el tema. Cuando se enfrente a un problema de programación que involucre objetos reales piense en la OOP y tal vez pueda resolver el problema de una forma más sencilla.