ORIENTACIÓN A OBJETOS

2º Entrega: Atributos Y Métodos

Variables De Instancia

Los atributos se corresponden con las características de los objetos del mundo real.
- En la terminología de Python, a estos atributos se les denomina **variables de instancia** y siempre deben ir precedidos de la referencia **self**.

Con el ejemplo anterior de la moto, ahora modelamos una clase que contenga variables de instancia:

```
class Moto():
    def __init__ (self, marca, modelo, color):
            self.marca=marca
            self.modelo=modelo
            self.color=color
```

Los atributos de la clase se crean al inicializar el objeto, utilizando tres variables diferentes que son pasadas como parámetros del constructor.

Ahora podríamos crear e inicializar un objeto de la siguiente forma:

```
>>> bmw_r1000=Moto("BMW","R100","blanca")
>>> suzuki_gsx=Moto("Suzuki","gsx","negra")
```

Importante:
Dado que self es empleado en todos los métodos de instancia, podemos acceder a los atributos en cualquiera de estos métodos.
-Esta es una de las ventajas de usar self, ya que cuando lo hacemos seguido de un atributo, tenemos la seguridad de que nos referimos al mismo y no a una variable definida en el espacio de nombres del método en cuestión-

Ej:

```
class Moto ():
    def __init__(self, marca, modelo, color):
            self.marca=marca
            self.modelo=modelo
            self.color=color

    def get_marca(self):
            marca="Nueva marca"
            print(self.marca)

>>>honda_cbr=Moto("Honda","CBR","Roja")
>>> honda_cbr.get_marca()
Honda
>>>
```

Aquí accedemos a un atributo de la clase..

```
>>> print(bmw_r1000.marca)
BMW
>>> print(bmw_r1000.color)
blanca
>>> print(bmw_r1000.modelo)
R100
```

Métodos De Instancia

Este tipo de métodos son aquellos que definen las operaciones que pueden realizar los objetos.
Ejemplo: (get_marca) es un ejemplo de ello.

En gral a este tipo de métodos se les llama simplemente métodos…

- Luego veremos que hay otros tipos de métodos que pueden definirse en una clase.
- Además de self, un método de instancia puede recibir un numero “n” de parámetros.

Ahora a la clase Moto le añadimos un nuevo método que nos permita acelerar:

```
def acelerar (self, km):
        print("acelerando {0} km". format(km))
```

si llamamos al nuevo método quedaría lo siguiente:

```
>>> honda_cbr.acelerar(20)
acelerando 20 km
```

Nota:
Formalmente, un método de instancia es una función que se define dentro de una clase y cuyo primer argumento es siempre una referencia a la instancia que lo invoca.

Variables De Clase

También existen en Python las variables de clase.

- Están realcionadas con los atributos
- Se caracterizan porque no forman parte de una instancia concreta, si no de la clase en general.

"Esto implica que pueden ser invocados sin necesidad de hacerlo a través de una instancia."

Para declararlas hay que crear una variable justo después de la definición de la clase y fuera de cualquier método.

Ej: Declaramos las variable num_ruedas, que contendra un valor para cada clase

```
class Moto:
num_ruedas=2
def __init__(self, marca, modelo, color):
        self.marca=marca
        self.modelo=modelo
        self.color=color
```

Así podemos invocar a la variable de clase directamente:

```
>>> Moto. num_ruedas
2
>>>
```

Ademas, una instancia también puede hacer uso de la variable de clase:

```
>>> m=Moto("honda","biz","negra")
>>> m. num_ruedas
2
>>>
```

Nota:
Estas variables de clases de Python son parecidas a las declaradas en Java o C++ a través de la palabra clave static.
Pero en Python las variables de clase pueden ser modificadas porque trata la visibilidad de componentes.

Propiedades

Las variables de instancia o atributos:

- definen una serie de características que poseen los objetos.
- se declaran haciendo uso de la referencia a la instancia a través de self.

Cuando se necesita un tratamiento inicial

Python ofrece la posibilidad de utilizar un método alternativo que resulta útil cuando estos atributos requieren de un procesamiento inicial en el momento de ser accedidos.
Para implementar este mecanismo, Python emplea un decorador llamado **property**.

Nota:
Este decorador o decorator es, básicamente, un modificador que permite ejecutar una versión modificada o decorada de una función o método concreto.

Supongamos que tenemos una clase que representa un circulo y deseamos tener un atributo que se refiera al área de mismo. Dado que esta área puede ser calculada en función del radio del circulo, parece lógico utilizar **property** para llevar a cabo el procesamiento requerido.

Ej:

```
from math import pi
class circle:
    def __init__(self, radio):
            self.radio=radio
    @property
    def area(self):
            return pi* (self.radio**2)
```

Dada la anterior definición, podemos acceder directamente a area, tal y como haríamos con cualquier variable de instancia.
Ej:

```
>>> c=circle(25)
>>> print(c.area)
1963.4954084936207
>>>
```

Alternativa

En lugar de emplear el decorator, se puede definir un método en la clase y luego emplear la función property() para convertirlo en un atributo.
El código equivalente al método decorado seria el siguiente:

```
>>> from math import pi
>>> class circle:
...     def __init__(self, radio):
...             self.radio=radio
...     def area(self):
...             return pi*(self.radio**2)
...     area=property(area)
...
>>>
>>> c=circle(20)
>>> c.area
1256.6370614359173
>>>
```

Importante:
Ya que la declaración y uso de decorator es mas sencilla y fácil de leer, recomendamos su utilización en detrimento de la mencionada función property()

Modificando atributos

Nuestro declarador en Python, solo sirve para acceder al atributo, pero no para modificarlo. Al ejecutar la siguiente sentencia, obtendremos un error:

```
>>>
>>> c.area=23
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: can't set attribute
>>>
```

Para modificar, Python permite utilizar otros métodos disponibles a través del decorador property.

Ej:
- el código anterior calcula el área y devuelve su valor directamente a través del decorador
- para ilustrar el uso de setters y getters en Python, nos centraremos en el atributo radio de nuestra clase Circulo.

-Entonces, para modificar el atributo radio haremos una serie de cambios en nuestra clase original.

1- En primer lugar, eliminaremos el constructor de la misma.
2- Después, añadiremos el siguiente método, donde, hemos creado un atributo privado empleando el doble guion bajo (__)

```
 from math import pi
class circle:
    @property
    def radio(self):
            return self.__radio
    def radio(self,radio):
            self.__radio=radio
```

Ahora veremos como emplear nuestra clase con estos cambios, para ello simplemente instanciaremos un nuevo objeto y modificaremos su valor...

```
>>>circulo=circle()
>>> circulo.radio=32
>>> print(circulo.radio)
32
>>>
```

Importante:
hemos utilizado el concepto de atributo privado, lo que implica que estamos trabajando con un modificador de visibilidad.