Clase_8_1_propiedades

May 3, 2023

1 Seminario de Lenguajes - Python

1.1 Repaso de POO. Propiedades

2 Sobre los objetos

- Son los elementos fundamentales de la POO.
- Son entidades que poseen **estado interno** y **comportamiento**.

3 Pensemos en la clase Usuario

- Cuando creamos un objeto, creamos una **instancia de la clase**.
- Una **instancia** es un **objeto individualizado** por los valores que tomen sus atributos o propiedades.

4 Observemos este código: ¿qué diferencia hay entre villanos y enemigos?

[ ]:
```python
class SuperHeroe():
    """ Esta clase  define a un superheroe
    villanos:  representa a los enemigos de todos los superhéroes
    """
    villanos = []

    def __init__(self, nombre, alias):
        self.alias = alias
        self.nombre = nombre
        self.enemigos = []

    def get_enemigos(self):
        return self.enemigos

    def agregar_enemigo(self, otro_enemigo):
        "Agrega un enemigo a los enemigos del superhéroe"

        self.enemigos.append(otro_enemigo)
        SuperHeroe.villanos.append(otro_enemigo)
```

5 Variables de instancia vs. de clase

Una **variable de instancia** es **exclusiva de cada instancia** u objeto.

Una **variable de clase** es única y es **compartida por todas las instancias** de la clase.

6 Público y privado

- Antes de empezar a hablar de esto ….

“‘Private” instance variables that cannot be accessed except from inside an object don’t exist in Python.’”

- De nuevo.. en español..

“Las variables «privadas» de instancia, que no pueden accederse excepto desde dentro de un objeto, no existen en Python””

- ¿Y entonces?
- Más info: https://docs.python.org/3/tutorial/classes.html#private-variables

7 Hay una convención ..

Es posible **definir el acceso** a determinados métodos y atributos de los objetos, quedando claro qué cosas se pueden y no se pueden utilizar desde **fuera de la clase**.

- **Por convención**, todo atributo (**propiedad** o método) que comienza con ”_” se considera no público.
  - A partir de ahora hablaremos de **variables de instancia** y métodos. Dejamos el término **propiedad** para otro concepto.
- Pero esto no impide que se acceda. **Simplemente es una convención**.

[4]:
```
class Usuario():
    "Define la entidad que representa a un usuario en UNLPImage"
    def __init__(self, nom="Sara Connor", alias="mama_de_John"):
        self._nombre = nom
        self.nick = alias
        self.avatar = None
    #Métodos
    def cambiar_nombre(self, nuevo_nombre):
        self._nombre = nuevo_nombre

obj = Usuario()
print(obj.nick)
```

```
mama_de_John
```

8 getters y setters

[5]:
```
class Demo:
    def __init__(self):
        self._x = 0
        self.y = 10

    def get_x(self):
        return self._x

    def set_x(self, value):
        self._x = value
```

- ¿Cuántas **variables de instancia**?
- Por cada variable de instancia **no pública** tenemos un método **get** y un método **set**. O, como veremos más adelante: **propiedades**.

9 Propiedades

- Podemos definir a **x** como una **propiedad** de la clase. ¿Qué significa esto? ¿Cuál es la ventaja?

[6]:
```
class Demo:
    def __init__(self):
        self._x = 0

    def get_x(self):
        print("estoy en get")
        return self._x

    def set_x(self, value):
        print("estoy en set")
        self._x = value

    x = property(get_x, set_x)
```

[7]:
```
obj = Demo()
print(obj.x)
```

```
estoy en get
0
```

[8]:
```
obj.x = 10
print(obj.x)
```

```
estoy en set
estoy en get
10
```

10 La función property()

- **property()** crea una **propiedad** de la clase.
- Forma general:

```
property(fget=None, fset=None, fdel=None, doc=None)
```

- +Info

11 El ejemplo completo

[9]:
```
class Demo:
    def __init__(self):
        self._x = 0
    def getx(self):
        print("estoy en get")
        return self._x
    def setx(self, value):
        print("estoy en set")
        self._x = value
    def delx(self):
        print("estoy en del")
        del self._x

    x = property(getx, setx, delx, "x es una propiedad")
```

[10]:
```
obj = Demo()
obj.x = 10
print(obj.x)
del obj.x
```

```
estoy en set
estoy en get
10
estoy en del
```

12 Más sobre property()

- ¿Qué pasa con el siguiente código si la **propiedad x** se define de la siguiente manera?:

[13]:
```
class Demo:
    def __init__(self):
        self._x = 0

    def getx(self):
        return self._x
```

```
    def setx(self, value):
        self._x = value

    def delx(self):
        del self._x

    x = property(getx, setx)
```

[14]:
```
obj = Demo()
obj.x = 10
```

13 ¿Y esto?

[15]:
```
class Demo:
    def __init__(self):
        self._x = 0

    @property
    def x(self):
        return self._x
```

[16]:
```
obj = Demo()
print(obj.x)
```

```
0
```

- @property es un **decorador**.

14 ¿Qué es un decorador?

- Un **decorador es una función** que recibe una función como argumento y extiende el comportamiento de esta última función sin modificarla explícitamente.

14.0.1 RECORDAMOS: las funciones son objetos de primera clase

- **¿Qué significa esto?** Pueden ser asignadas a variables, almacenadas en estructuras de datos, pasadas como argumentos a otras funciones e incluso retornadas como valores de otras funciones.

15 Observemos el siguiente código

[17]:
```
def decimos_hola(nombre):
    return f"Hola {nombre}!"

def decimos_chau(nombre):
    return f"Chau {nombre}!"
```

[18]:
```
def saludo_a_Clau(saludo):
    return saludo("Clau")
```

[20]:
```
saludo_a_Clau(decimos_hola)
#saludo_a_Clau(decimos_chau)
```

[20]:
```
'Hola Clau!'
```

16 ¿Qué podemos decir de este ejemplo?

- Ejemplo sacado de https://realpython.com/primer-on-python-decorators/

[21]:
```
def decorador(funcion):
    def funcion_interna():
        print("Antes de  invocar a la función.")
        funcion()
        print("Después de invocar a la función.")

    return funcion_interna

def decimos_hola():
    print("Hola!")
```

[22]:
```
saludo = decorador(decimos_hola)
```

- ¿De qué tipo es saludo?

[ ]:
```
def decorador(funcion):
    def funcion_interna():
        print("Antes de  invocar a la función.")
        funcion()
        print("Después de invocar a la función.")
    return funcion_interna

def decimos_hola():
    print("Hola!")

saludo = decorador(decimos_hola)
```

- ¿A qué función hace referencia saludo?

[23]:
```
saludo()
```

```
Antes de  invocar a la función.
Hola!
Después de invocar a la función.
```

17 Otra forma de escribir esto en Python:

[24]:
```
def decorador(funcion):
    def funcion_interna():
        print("Antes de  invocar a la función.")
        funcion()
        print("Después de invocar a la función.")
    return funcion_interna

@decorador
def decimos_hola():
    print("Hola!")
```

[25]:
```
decimos_hola()
```

```
Antes de  invocar a la función.
Hola!
Después de invocar a la función.
```

18 Es equivalente a:

```
decimos_hola = decorador(decimos_hola)
```

- +Ínfo
- +Info en español

19 Dijimos que @property es un decorador

[26]:
```
class Demo:
    def __init__(self):
        self._x = 0

    @property
    def x(self):
        return self._x
```

[28]:
```
obj = Demo()
obj.x = 10 # Esto dará error: ¿por qué?
print(obj.x)
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
Cell In[28], line 2
      1 obj = Demo()
----> 2 obj.x = 10 # Esto dará error: ¿por qué?
      3 print(obj.x)
```

```
AttributeError: property 'x' of 'Demo' object has no setter
```

- ATENCIÓN: x no es un método, es una propiedad.
- +Info

20 El ejemplo completo

[29]:
```
class Demo:
    def __init__(self):
        self._x = 0
    @property
    def x(self):
        print("Estoy en get")
        return self._x

    @x.setter
    def x(self, value):
        print("Estoy en get")
        self._x = value
```

[30]:
```
obj = Demo()
obj.x = 10
print(obj.x)
#del obj.x
```

```
Estoy en get
Estoy en get
10
```

21 Algo más para leer

https://realpython.com/python-getter-setter/