Propiedades – @property – Getter, Setter, Deleter – Encapsulamiento

Índice del contenido

Anteriormente sin haber hablado de encapsulamiento en Python había publicado una entrada sobre propiedades en python. Pero he decidido partir desde el comienzo con este concepto, así que aquí va de nuevo. Esta vez un poco más claro:

¿Qué es el encapsulamiento en Python?

La pregunta que deberíamos hacernos es, **¿Qué es el encapsulamiento en la programación orientada a objetos?, ¿Cómo lo trabaja Python?**

En otros lenguajes como, por ejemplo, Java cuando declaramos una variable dentro de una clase, como un atributo o un método podemos utilizar una palabra reservada para indicar si esta es **privada (private)** o **pública (public)**. Esto en Python se debe hacer mediante el **uso de guiones bajos ("_" , "__")**.

Para empezar a explicar esto debemos entender el ***concepto de encapsulamiento*** y porque es utilizado en la programación orientada a objetos.

"En programación modular, y más específicamente en programación orientada a objetos, se denomina **encapsulamiento** al ocultamiento del estado, es decir, de los datos miembros de un objeto de manera que sólo se pueda cambiar mediante las operaciones definidas para ese objeto.

Cada objeto está aislado del exterior, es un módulo natural, y la aplicación entera se reduce a un agregado o rompecabezas de objetos. El aislamiento protege a los datos asociados de un objeto contra su modificación por quien no tenga derecho a acceder a ellos, eliminando efectos secundarios e interacciones.

De esta forma el usuario de la clase puede obviar la implementación de los métodos y propiedades para concentrarse solo en cómo usarlos.

Por otro lado, se evita que el usuario pueda cambiar su estado de maneras imprevistas e incontroladas."

Como has podido leer, el encapsulamiento se define como el "ocultamiento del estado" de "los datos miembros de un objeto". Hablamos, por lo tanto, de ocultar **los datos de atributos o métodos**, más específicamente de **protegerlos** para que sólo puedan ser cambiados mediante "**operaciones definidas para tal fin"**. Esto nos asegura, por ejemplo, que "no podremos modificar un atributo si no es a través de un método que hallamos creado específicamente para ello" y aquí es donde nacen los famosos "***Getter, Setter, Deleter***".

¿Pero qué es el ámbito "¿Protegido, privado"?

Anteriormente creamos clases con sus **atributos y métodos públicos**, es decir, que son accesibles desde su instancia. Basta con crear un objeto a partir de esa clase y podremos acceder y modificar libremente sus atributos. Y no sólo desde su instancia sino también desde otra clase que herede de la anterior, por lo que un atributo que almacena información sensible puede ser modificado fácilmente.

Cuidado:

En Python las propiedades y métodos privados no existen como tal, por lo que son fácilmente sobre-escribibles.

Entonces, ¿debo seguir leyendo?

Depende, si vienes de otro lenguaje como Java a programar a Python, seguro te habrás preguntado sobre "Encapsulamiento", pero si no, te recomiendo que no sigas leyendo a menos que sea para organizar tu código de esta manera por propia elección. Es decir que en realidad esto va de la decisión del desarrollador utilizarlo, pero no es obligatorio porque como ya dijimos **en Python de todas maneras "lo privado no existe"**. Y si a ti de todas maneras no te interesa aplicar este modelo, leer sobre encapsulamiento sólo te servirá en el caso que te encuentres con código de otro desarrollador que si lo aplique, lo que es bastante normal.

Habiendo aclarado esto podemos continuar.

Atributos protegidos en Python ("_").

A menudo cuando andamos buscando ejemplos de algún código podemos encontrarnos con cosas como estas:

```
1.  #!/usr/bin/env python
2.  # -*- coding: utf-8 -*-
3.  #
4.  #  encaps.py
5.  #
6.  #  Copyright 2020 Pyr0 Maniac <Pyro@Pyro>
7.
8.  class usuario (object):
9.      def __init__(self, nombre, clave):
10.         self.nombre = nombre
11.         self._clave = clave
12.
13. Usuario1 = usuario ("Roberto", "qwerty")
14.
15. print (Usuario1.nombre, Usuario1._clave)
```

Resultado:

('Roberto', 'qwerty')

Como ves el atributo "clave" está precedido por un guion bajo, lo que indica que es un atributo protegido (_). Lo cual **establece que sólo puede ser accedido por esa clase y sus sub-clases**, es decir, aquellas que hereden de la clase padre. Se suele ver muy a menudo como una buena práctica para atributos o métodos de uso interno y también para evitar la colisión de los mismos nombres de métodos o atributos causados por herencia. Siempre y cuando estemos hablando de programas que recurren a muchas clases, es probable que los veas utilizar.

De lo contrario, resulta, en realidad, innecesario.

Atributos privados en Python ("__").

En el caso de un atributo privado (__) estamos indicando que este sólo podrá ser accedido o modificado si se especifica la clase precedida por un guion bajo seguida del atributo precedido por doble guion bajo.

```
1.  #!/usr/bin/env python
2.  # -*- coding: utf-8 -*-
3.  #
4.  #  encaps.py
5.  #
6.  #  Copyright 2020 Pyr0 Maniac <Pyro@Pyro>
7.
8.  class usuario (object):
9.      def __init__(self, nombre, clave):
10.         self.nombre = nombre
11.         self.__clave = clave
12.
13. Usuario1 = usuario ("Roberto", "qwerty")
14.
15. print (Usuario1.nombre, Usuario1.__clave)
```

Resultado:

AttributeError: 'usuario' object has no attribute '__clave'

Cómo se puede observar, si hacemos esto obtendremos como salida que no existe el atributo clave que estamos intentando imprimir ya que como dije anteriormente, la forma correcta de acceder a él sería especificando primero la clase a la que pertenece de la siguiente manera:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
#
#  encaps.py
#
#  Copyright 2020 Pyr0 Maniac <Pyro@Pyro>
#
class usuario (object):
    def __init__(self, nombre, clave):
        self.nombre = nombre
        self.__clave = clave

Usuario1 = usuario ("Roberto", "qwerty")

print (Usuario1.nombre, Usuario1._usuario__clave)
```

Resultado:

('Roberto', 'qwerty')

Como ves podemos acceder igualmente a un atributo por más que sea privado y modificarlo de la misma manera. Pero no es lo que se "considera correcto". Por lo que, para ello, si deseamos implementar métodos que nos permitan modificar estos atributos de la forma que se suele hacer en otros lenguajes dónde se aplica el "encapsulamiento" podemos hacerlo utilizando Getter, Setter, Deleter mediante el uso del decorador @Property.

Propiedades de atributos de clase en Python: Getter, Setter y Deleter.

Siguiendo con las propiedades en Python tratemos de entenderlo rápidamente.

Atención

En python dentro de las clases podríamos decir que todo son atributos, incluso los métodos podríamos definirlos como "atributos llamables" y las propiedades **"atributos personalizables"**. Por ende ahora:

Las propiedades son atributos que "manejamos" mediante Getter, Setter y Deleter.

"Atributos manejables" que nos permiten invocar código personalizado al ser creados, modificados o eliminados.

Las propiedades nos permiten por ejemplo llamar código personalizado cuando un atributo, método, variable es mostrado/a, modificado/a, borrado/a.

Otro caso de ejemplo sería, automatizar la modificación de una variable "B" cuando se modifique "A" o guardar un historial de modificación de un atributo, entre otros muchos usos...

@Property en Python:

La función integrada **property()** nos permitirá interceptar la escritura, lectura o borrado de los atributos y además nos permite incorporar documentación sobre los mismos. La sintaxis para invocarla es la siguiente:

@property

Si, es un decorador.

Si nosotros no pasamos alguno de los parámetros su valor por defecto sera None.

***Getter*:** Se encargará de interceptar la lectura del atributo. (get = obtener).

***Setter*:** Se encargará de interceptar cuando se escriba. (set = definir o escribir).

***Deleter* :** Se encargará de interceptar cuando es borrado. (delete = borrar).

doc : Recibirá una cadena para documentar el atributo. (doc = documentación).

```python
#!/usr/bin/env python
# -*- coding: utf-8 -*-
# pythones.net

class Perros(object): #Declaramos la clase principal Perros
    def __init__(self, nombre, peso): #Definimos los parámetros
        self.__nombre = nombre #Declaramos los atributos (privados ocultos)
        self.__peso = peso
    @property
    def nombre(self): #Definimos el método para obtener el nombre
        "Documentación del método nombre bla bla" # Doc del método
        return self.__nombre #Aquí simplemente estamos retornando el atributo privado oculto



#Hasta aquí definimos los métodos para obtener los atributos ocultos o privados getter.
#Ahora vamos a utilizar setter y deleter para modificarlos

    @nombre.setter #Propiedad SETTER
    def nombre(self, nuevo):
        print ("Modificando nombre..")
        self.__nombre = nuevo
        print ("El nombre se ha modificado por")
        print (self.__nombre) #Aquí vuelvo a pedir que retorne el atributo para confirmar
    @nombre.deleter #Propiedad DELETER
    def nombre(self):
        print("Borrando nombre..")
        del self.__nombre

        #Hasta aquí property#

    def peso(self):    #Definimos el método para obtener el peso
        return self.__peso #Aquí simplemente estamos retornando el atributo privado

#Instanciamos
Tomas = Perros('Tom', 27)
print (Tomas.nombre) #Imprimimos el nombre de Tomas. Se hace a través de getter
#Que en este caso como esta luego de property lo toma como el primer método..

Tomas.nombre = 'Tomasito' #Cambiamos el atributo nombre que se hace a través de setter
del Tomas.nombre #Borramos el nombre utilizando deleter
```

Resultado:

Tom

Modificando nombre...

El nombre se ha modificado por

Tomasito

Borrando nombre...

Se define primero @property y justo después el método mediante el cual devolvemos el nombre (get), en este caso al ser el primer método después de @property hace que lo tome automáticamente como Getter (linea 10). A continuación, especificamos el (set) lo que nos permite lanzar un print al modificar el atributo privado nombre. Y luego el (deleter) que nos permite lanzar otro print al borrarlo.

Y eso es todo, sencillo y sin más vueltas.

En el ejemplo anterior las propiedades no aplican al atributo peso.
Cuando llamemos, modifiquemos o eliminemos el atributo privado "__nombre", se aplicarán dichas modificaciones según la acción que se realiza con él. ¿Me explico? Las propiedades te permiten variar el resultado según la acción que se realiza con el atributo, si lo modificas sucede algo. Pero si lo borras sucede otra cosa.

Recordamos que después de @property debes especificar el nombre del método seguido del punto y la propiedad de la que se trate (setter, getter, deleter).

Ahora vamos a agregar el peso, para posteriormente modificarlo u borrarlo. En ese caso no podemos colocarlo dentro del mismo decorador, por lo tanto, se debe crear otro decorador para dicho método.

```python
#!/usr/bin/env python
# -*- coding: utf-8 -*-
# pythones.net

class Perros(object): #Declaramos la clase principal Perros
    def __init__(self, nombre, peso): #Definimos los parámetros
        self.__nombre = nombre #Declaramos los atributos
        self.__peso = peso
    @property
    def nombre(self): #Definimos el método para obtener el nombre
        "Documentación del método nombre bla bla" # Doc del método
        return self.__nombre #Aquí simplemente estamos retornando el atributo privado


#Hasta aquí definimos los métodos para obtener los atributos ocultos o privados getter.
#Ahora vamos a utilizar setter y deleter para modificarlos

    @nombre.setter #Propiedad SETTER
    def nombre(self, nuevo):
        print ("Modificando nombre..")
        self.__nombre = nuevo
        print ("El nombre se ha modificado por")
        print (self.__nombre) #Aquí vuelvo a pedir que retorne el atributo para confirmar
    @nombre.deleter #Propiedad DELETER
    def nombre(self):
        print("Borrando nombre..")
        del self.__nombre

    @property
    def peso(self):    #Definimos el método para obtener el peso #Automáticamente GETTER
        return self.__peso #Aquí simplemente estamos retornando el atributo privado

    @peso.setter
    def peso(self, nuevopeso):
        self.__peso = nuevopeso
        print ("El peso ahora es")
        print (self.__peso)
    @peso.deleter #Propiedad DELETER
    def peso(self):
        print("Borrando peso..")
        del self.__peso
#Instanciamos
Tomas = Perros('Tom', 27)
print (Tomas.nombre) #Imprimimos el nombre de Tomas. Se hace a través de getter
#Que en este caso como esta luego de property lo toma como el primer método..

Tomas.nombre = 'Tomasito' #Cambiamos el atributo nombre que se hace a través de setter
print (Tomas.nombre) #Volvemos a imprimir
Tomas.peso = 28
del Tomas.nombre #Borramos el nombre utilizando deleter
```

Resultado:

Tom

Modificando nombre..

El nombre se ha modificado por

Tomasito

El peso ahora es

28

Borrando nombre..