ORIENTACIÓN A OBJETOS

3º Entrega: Visibilidad, Métodos de Clase, Métodos Estáticos

Visibilidad

Uno de los aspectos principales en los que difiere la orientación a objetos de Python con respecto a otros lenguajes es en el concepto de visibilidad de componentes de una clase.
En Python no contamos con modificadores como public o private y simplemente, todos los métodos y atributos pueden considerarse como publicos.
Es decir, realmente no se puede impedir el acceso a un objeto o atributo desde la instancia de una clase concreta.

Ahora bien: Si es posible de alguna forma indicar que un atributo o método solo debe ser invocado desde la misma clase!!
Esto correspondería a utilizar el modificador private en otro lenguaje.
Los programadores de Python utilizan una sencilla convención:

- si un atributo debe ser privado, basta con anteponer un único guion bajo (underscore).
- sin embargo, para Python, esto no significa nada, simplemente es una convenio entre programadores.

Lo anterior sobre la visibilidad es aplicable, tanto a variables como a métodos.
Entonces, si utilizamos el **underscore** para declarar un método como privado, veremos como eso no impide su acceso.

```
Ej: declaramos una clase y su correspondiente método

class Test():
    def _Privado(self):
        print("Metodo privado")
```

Ahora crearemos una instancia y observaremos como la llamada al método en cuestión es valida:

```
>>> t=Test()
>>> t._Privado()
Metodo privado
>>>
```

Doble Guion Bajo Para Un Atributo Privado
hemos visto que el doble guion bajo (__) ha sido empleado para declarar un atributo privado de instancia.
Realmente, si el atributo ha sido declarado de esta forma, Python previene el acceso desde la instancia.

Ej:
Observemos la siguiente clase que declara un atributo de esta forma y comprobemos como Python no nos deja acceder al mismo:

```
class Privado():
    def __init__(self):
          sel.__atributo=1

>>>p=Privado
>>>p.__atributo
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: type object 'Privado' has no attribute '__atributo'
>>>
```

Sin embargo, para acceder a este atributo y dado que Python en realidad no entiende el concepto de visibilidad como tal, el lenguaje emplea un mecanismo técnicamente denominado "name mangling".
-Este consiste en convertir cualquier atributo declarado con doble guion en simple guion seguido del nombre de la clase y luego doble guion bajo para acceso directo al atributo.
- Así pues, y continuando con nuestra clase Privada, la forma de acceder a nuestro atributo seria la siguiente:

```
class Privado():
    def __init__(self):
          self.__atributo=1

>>> p=Privado()
>>> p_atributo
>>> p._Privado__atributo
1
>>>
```

Algunos prefieren no emplear la denominación privado para referirse a este tipo de atributos. En su lugar, prefieren llamarlos pseudoprivados.

La técnica del name mangling se diseño en Python para asegurar que una clase hijo no sobrescribe accidentalmente los métodos y/o atributos de su clase padre.

Esta es la verdadera razón de su existencia, no la de poder utilizar un modificador de visibilidad.

Acabamos de introducir el concepto de hijo y padre para referirnos a dos tipos de clases diferentes.
- Estos conceptos se engloban dentro del ámbito de la herencia

Ej:
El código muestra como evitar la sobrescritura del atributo test:

```
class Padre():
    def __init__(self):
          self.__test="Padre"

class Hijo():
    def __init__(self):
          self.__test="Hijo"

>>> p=Padre()
>>> h=Hijo()
>>> p._Padre__test
'Padre'
>>> h._Hijo__test
'Hijo'
>>>
```

Sin aplicar la técnica del name mangling, en el ejemplo anterior, el atributo test de la clase padre seria sobrescrito por el de la clase hija y otra clase que heredara de la pare se vería afectada.

Métodos De Clase

Ya hemos aprendido sobre atributos y métodos de instancia. Pero Python también permite crear métodos de clase.
Los métodos de clase:

- Se caracterizan porque pueden ser invocados directamente sobre la clase, sin necesidad de car ninguna instancia.
- Dado que no vamos a utilizar un objeto, no es necesario emplear el argumento self como referencia en los métodos de clase.
- En su lugar, si que debemos contar con otro parámetro similar que referenciará a la clase en cuestión.

Nota:
Por convención, al igual que se emplea self para los métodos de instancia, cls es el nombre que suele ser empleado como referencia en los métodos de clase.

- Ademas, esto implica que el programador no tiene que pasar como parámetro la mencionada referencia, ya que Python lo hace automáticamente.
- Eso si, es responsabilidad del programador utilizar el parámetro formal, para dicha referencia, en la definición del método.
- Esto es valido tanto para self como para cls

Por otro lado y al igual que en el caso de los métodos de instancia, se pueden utilizar parámetros adicionales para el Método de clase.

- Lo que si que es importante que nunca olvidemos la referencia cls en su definición.
- El resto de parámetros son opciones

Los métodos de clase requieren el uso de un decorador definido por Python, su nombres es "**classmethod**" y funciona de forma similar a como lo hace el comentado **property**.

- Gracias al decorator, solo debemos definir el Metodo con el argumento referencial cls y escribir su funcionalidad.

Ej:

```
class Test:
    def __init__(self):
          self.x=8

 class Test:
    def __init__(self):
          self.x=8
    @classmethod
    def método._clase(cls, param1):
          print("Parametro: {0}".format(param1))
```

invocar directamente al método de clase:

```
>>> Test.metodo_clase(6)
Parametro: 6
>>>
```

Invocar el método de instancia, previamente creando un objeto de la clase

```
>>> t=Test()
>>> print(t.x)
8
```

Importante:
Es interesante saber que los métodos de clase también pueden ser invocados a través de una instancia de la misma. Se los invoca como si de un método de instancia se tratase.

La siguiente sentencia es valida para el ejemplo anterior y produce el siguiente resultado:

```
>>> t.metodo_clase(5)
Parametro: 5
>>>
```

Lógicamente, si un método. de clase puede se invocado desde una instancia, también podemos crear un objeto e invocar directamente al método. en cuestión. en la misma linea de código:

```
>>> Test().método._clase(11)
Parametro: 11
>>>
```

En el comportamiento anteriormente explicado tiene sentido, ya que, en realidad, Python no tiene modificadores de visibilidad, entonces un método de clase es también un método de instancia.
---La diferencia es que un método de instancia no puede ser creado si esta no existe.---

Nota:
1) Lenguaje como Java y C++ emplean la palabra clave static para declarar métodos de clase. Ademas, en estos lenguajes, no es posible utilizar la referencia a la instancia, llamada en ambos casos this..
--Sin embargo, estos no pueden ser invocados desde una instancia, a diferencia de Python--
2) Otros lenguajes como Ruby y Smalltalk no tienen métodos estáticos, pero si de clase. Esto implica que estos métodos si que tienen acceso a los datos de la instancia.
---Python cuenta tanto con métodos de clase como con métodos estáticos---

Métodos Estáticos

Otro de los tipos de métodos que puede contener una clase en Python son los llamados estáticos.

- La principal diferencia con respecto a los métodos de clase es que los estáticos no necesitan ningún argumento como referencia, ni a la instancia, ni a la clase.
- All no tener esta referencia, un método estático no puede acceder a ningún atributo de la clase.
- De la misma forma que los métodos de clase en Python usan el decorador **classmethod**, para los estáticos contamos con otro decorador llamado **staticmethod**.
- La definición de un método estático requiere del uso de este decorador, que debe aparecer antes de la definición del mismo.

Ej: para comprobar como funcionan los métodos estáticos, añadamos el siguiente método a nuestra clase Test:

```
class Test:
    def __init__(self):
            self.x=8
    @classmethod
    def método._clase(cls,param1):
            print("Parametro: {0}".format(param1))
    @staticmethod
    def método._estático(valor):
            print("Valor: {0}".format(valor))
```

invocando directamente desde una instancia determinada de la clase

```
>>> t=Test()
>>> t.metodo_estático(32)
Valor: 32
>>>
```

Nota:
1- Podemos apreciar que la diferencia de un método estático y otro de instancia es, ademas del uso del decorado en cuestión, que como primer parámetro no estamos usando self para referenciar a la instancia.
2- Pero este hecho tiene otra implicación y es que no es posible acceder desde el mismo a un atributo de clase.

Entonces si cambiamos el código del método anterior por el siguiente, se producirá una excepción en tiempo de ejecución:

```
>>> t=Test()
>>> t.metodo_estático(1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 6, in método._estático
NameError: name 'self' is not defined
>>>
```

"el error en cuestión, producido por la ejecución del método estático desde una instancia, seria: NameError: global name self is not defined"

Detalle:
- Es lógico que se produzca el error, porque self es solo una convención para referenciar a la instancia de clase. Dado que el método es estático y no existe tal referencia al mismo, no es posible utilizar ningún atributo de instancia en su interior.
-**NameError** es una excepción predefinida por Python y es lanzada como consecuencia de intentar acceder al atributo.

Nota:
Algunos programadores prefieren crear una función en lugar de un método estático. Se puede llamar a la función y utilizar su resultado interactuando posteriormente con la instancia de la clase.
Sin embargo, otros prefieren emplear los métodos estáticos, argumentando que, si la funcionalidad esta estrechamente relacionada con la clase, se mantiene y cumple el principio de abstracción, comúnmente utilizado en la programación orientada a objetos.