Taller de Repaso POO

Modificadores de acceso en Python

Mauricio Cepeda Villanueva

Parte A. **Conceptos y lectura de código**

1) A) a.x, B) a._y, D) a._A__z

2) Imprime: False True

3) a) Falso, El guion bajo simple (_atributo) es solo convención: indica “esto es interno”, pero en realidad sí puedo acceder desde fuera
b) Falso, Con(__atributo) ocurre el name mangling: Python lo transforma internamente a _<Clase>__atributo. Eso lo vuelve menos accesible de manera accidental, pero no imposible; aún se puede acceder usando el nombre cambiado.
c) Verdadero, El atributo (__x) en una clase A se guarda como (_A__x), mientras que en una clase B sería (_B__x). El nombre de la clase forma parte del atributo mangled.

4) Imprime: abc y no hay error de acceso porque en Python el prefijo con un solo guion bajo (_token) es solo convención de “uso interno”. El lenguaje no bloquea el acceso desde fuera de la clase ni desde una subclase. Por eso, Sub puede usar sin problema el atributo (_token) definido en Base.

5) Imprime: (2, 1)

6) Error: AttributeError, porque __slots__ solo permite 'x'.

7) self._b = 99

8) Imprime True False True, porque

- (_step) existe tal cual, porque con un solo guion bajo es solo convención
- (__tick) no existe con ese nombre, porque Python lo transformó con name mangling
- (_M__tick) sí existe, porque así se renombró internamente el método(__tick) en la clase M

9) print(s._S__data)

10) Aparece: _D__a, porque cuando se declara un atributo con (__a), Python aplica name mangling, transformándolo en(_D__a) (porque la clase se llama D). Por eso, si se filtra dir(d) buscando 'a', el que se encuentra es el nombre transformado (_D__a)
Ni (__a) ni a aparecen en la lista: __a porque fue renombrado, y a porque nunca existió en teoría

Parte B. **Encapsulación con @property y validación**

11) @saldo.setter

```
  def saldo(self, value):
    if value < 0:
      raise ValueError("Saldo no puede ser negativo")
    self._saldo = value
```

12) class Termometro:

```
  def __init__(self, temperatura_c):
    self._c = float(temperatura_c)
  @property
  def temperatura_f(self):
    return self._c * 9/5 + 32
```

13) @nombre.setter

```
  def nombre(self, value):
    if not isinstance(value, str):
      raise TypeError("nombre debe ser str")
    self._nombre = value
```

14) class Registro:

```
  def __init__(self):
    self.__items = []
  def add(self, x):
```

```
        self.__items.append(x)
    @property
    def items(self):
        return tuple(self.__items)
```

Parte C. **Diseño y refactor**

15)
```
class Motor:
    def __init__(self, velocidad):
        self.velocidad = velocidad
    @property
    def velocidad(self):
        return self._velocidad
    @velocidad.setter
    def velocidad(self, v):
        if not (0 <= v <= 200):
            raise ValueError("Velocidad debe estar entre 0 y 200")
        self._velocidad = v
```

16) _atributo: Es como decir “esto es privado, pero si alguien lo necesita, lo puede usar”. No está hecho para mostrarse, pero tampoco es como que este escondido del todo.

__atributo: Aquí sí es como que esto no se puede usar. Python como que le cambia el nombre internamente para que no sea fácil de encontrar. Es como esconder algo y que nadie lo toque sin permiso.

17) En lugar de devolver la lista original, se debe devolver como una copia de la lista o una vista inmutable (por ejemplo, una tupla). Así, alguien que use este objeto podrá consultar los datos, pero no es como que podrá modificar la estructura interna de manera tan directa.

18) Falla en la llamada al self.__x dentro del método get() esa línea intenta acceder (a __x) desde B, pero (__x) en la clase base A fue renombrado como (_A__x), mientras que self.__x dentro de B se vuelve (_B__x) entonces B().get() arrojará AttributeError al no encontrar _B__x y pues básicamente se podría solucionar no usando directamente (__x) en la base, creo que podría usar usar(_x)

19) class _Repositorio:

```
    def guardar(self, k, v):
        return self.__repo.guardar(k, v)
```

20) class ContadorSeguro:

```
  def __init__(self):
    self._n = 0

  def inc(self):
    self._n += 1
    self.__log()

  @property
  def n(self):
    return self._n

  def __log(self):
    print("tick")
```