Taller: POO y modificadores de acceso en Python

Instrucciones

• Lee cada fragmento, ejecuta mentalmente el código y responde lo que se pide. • Recuerda: en Python no hay “modificadores” como en Java/C++; se usan convenciones:

- Público: `nombre`
- Protegido (convención): `_nombre`
- Privado (name mangling): `__nombre` se convierte a `_<Clase>__nombre` • No edites el código salvo que la pregunta lo solicite.

Parte A. Conceptos y lectura de código

1) Selección múltiple

Dada la clase:

```
class A:
 x = 1
 _y = 2
 __z = 3

a = A()
```

¿Cuáles de los siguientes nombres existen como atributos accesibles directamente desde `a`?

A) `a.x`
B) `a._y`
C) `a.__z`
D) `a._A__z`

Respuesta: A, B y D

2) Salida del programa

```
class A:
 def __init__(self):
 self.__secret = 42

a = A()
print(hasattr(a, '__secret'), hasattr (a,'_A__secret'))
```

¿Qué imprime?

Respuesta: `False True` ya que la función `hasattr()` revisa si el objeto tiene o no un atributo con ese nombre y el nombre del atributo se vuelve el segundo por name mangling.

3) Verdadero/Falso (explica por qué)

a) El prefijo _ impide el acceso desde fuera de la clase.

b) El prefijo __ hace imposible acceder al atributo.

c) El name mangling depende del nombre de la clase.

Respuesta:

a.) Falso: el prefijo "_" solo es una convención dentro de python que no impide el acceso.

b.)Falso: el prefijo "__" activa el name mangling, cambiando el nombre del atributo para hacerlo más difícil de acceder pero no imposible.

c.)Verdadero: Ya que como el name mangling utiliza el nombre de la clase antes del atributo, cambia dependiendo del nombre de la clase.

4) Lectura de código

```
class Base:
 def __init__(self):
 self._token = "abc"

class Sub(Base):
 def reveal(self):
```

```
 return self._token

print(Sub().reveal())
```

¿Qué se imprime y por qué no hay error de acceso?
Respuesta: se imprime “abc” ya que el prefijo _ en el atributo `self._token` solo es una convención y no impide el acceso.

5) Name mangling en herencia

```
class Base:
 def __init__(self):
 self.__v = 1

class Sub(Base):
 def __init__(self):
 super().__init__()
 self.__v = 2
 def show(self):
 return (self.__v, self._Base__v)

print(Sub().show())
```

¿Cuál es la salida?
Respuesta: `(2, 1)`

6) Identifica el error

```
class Caja:
 __slots__ = ('x',)

c = Caja()
c.x = 10
c.y = 20
```

¿Qué ocurre y por qué?

Respuesta: y nunca fue definido en `__slots__` de la clase `Caja`

7) Rellenar espacios

Completa para que `b` tenga un atributo “protegido por convención”.

```
class B:
 def __init__(self):
 self ______ = 99
```

Escribe el nombre correcto del atributo.

Respuesta: El nombre correcto del atributo es `self._numero`

8) Lectura de métodos “privados”

```
class M:
 def __init__(self):
 self._state = 0

 def _step(self):
 self._state += 1
 return self._state

 def __tick(self):
 return self._step()

m = M()
print(hasattr(m, '_step'), hasattr(m, '__tick'), hasattr(m,
 '_M__tick'))
```

¿Qué imprime y por qué?

Respuesta: `True, False, True` siguiendo la lógica del punto 2, `_step` si se puede acceder ya que solo está protegido, mientras que `__tick` al ser privado pasa por name mangling, por lo que se convierte en `_M__tick`

9) Acceso a atributos privados

```
class S:
 def __init__(self):
 self.__data = [1, 2]
 def size(self):
 return len(self.__data)

s = S()
# Accede a __data (solo para comprobar), sin modificar el código de la
clase:
# Escribe una línea que obtenga la lista usando name mangling y la
imprima.
```

Escribe la línea solicitada.
Respuesta: `print('s._S__data')`

10) Comprensión de `dir` y mangling

```
class D:
 def __init__(self):
 self.__a = 1
 self._b = 2
 self.c = 3

d = D()
names = [n for n in dir(d) if 'a' in n]
print(names)
```

¿Cuál de estos nombres es más probable que aparezca en la lista: `__a`, `_D__a` o `a`? Explica.

Respuesta: `_D__a` es el más probable ya que `__a` pasa por name mangling, por lo cual no se va a encontrar y el atributo a es privado

Parte B. Encapsulación con `@property` y validación

11) Completar propiedad con validación

Completa para que `saldo` nunca sea negativo.

```
class Cuenta:
 def __init__(self, saldo):
 self._saldo = 0
 self.saldo = saldo

 @property
 def saldo(self):
      return self._saldo

 @saldo.setter
 def saldo(self, value):
 # Validar no-negativo
      if value < 0:
           raise ValueError("El saldo no puede ser negativo")
      self._saldo = value
```

12) Propiedad de solo lectura

Convierte `temperatura_f` en un atributo de solo lectura que se calcula desde `temperatura_c`.

```
class Termometro:
 def __init__(self, temperatura_c):
 self._c = float(temperatura_c)

 # Define aquí la propiedad temperatura_f: F = C * 9/5 + 32
```

Escribe la propiedad.

Respuesta:

```
@property
def temperatura_f(self):
    return self._c * 9/5 + 32
```

13) Invariante con tipo

Haz que `nombre` sea siempre `str`. Si asignan algo que no sea `str`, lanza `TypeError`.

```
class Usuario:
def __init__(self, nombre):
self.nombre = nombre

# Implementa property para nombre
@property
def nombre(self):
    return self._nombre

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

14) Encapsulación de colección

Expón una vista de solo lectura de una lista interna.

```
class Registro:
def __init__(self):
self.__items = []

def add(self, x):
self.__items.append(x)

# Crea una propiedad 'items' que retorne una tupla inmutable con el
```

contenido

Respuesta:

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

Parte C. Diseño y refactor

15) Refactor a encapsulación

Refactoriza para evitar acceso directo al atributo y validar que `velocidad` sea entre 0 y 200.

```
class Motor:
 def __init__(self, velocidad):
 self.velocidad = velocidad # refactor aquí
```

Escribe la versión con `@property`.

Respuesta:

```
@property
def velocidad(self):
    return self._velocidad

@velocidad.setter
def velocidad(self, value):
    if not (0 <= value <= 200):
        raise ValueError("La velocidad debe estar entre 0 y 200")
    self._velocidad = value
```

16) Elección de convención

Explica con tus palabras cuándo usarías `_atributo` frente a `__atributo` en una API pública de una librería.

Respuesta: `_atributo` lo utilizaría para definir que el atributo es interno, pero para no esconderlo completamente, y permitiendo que sea utilizado por cualquiera bajo su propio riesgo. En cambio `__atributo` lo usaria para reducir el riesgo de que otra subclase defina un atributo con el mismo nombre y se dañe el código sin querer.

17) Detección de fuga de encapsulación

¿Qué problema hay aquí?

```
class Buffer:
 def __init__(self, data):
 self._data = list(data)
 def get_data(self):
 return self._data
```

Respuesta: El método `get_data` devuelve la lista interna directamente. Eso significa que desde fuera alguien puede modificarla y alterar el estado interno del objeto sin pasar por ningún control

Propón una corrección.

Respuesta:

```
class Buffer:
    def __init__(self, data):
        self._data = list(data)

    def get_data(self):
        return tuple(self._data)
```

18) Diseño con herencia y mangling

¿Dónde fallará esto y cómo lo arreglas?

```
class A:
 def __init__(self):
 self.__x = 1
class B(A):
 def get(self):
 return self.__x
```

Respuesta: `self.__x = 1` se guarda como `self._A__x`, y en `B.get`, al pedir `self.__x` python aplica el name mangling para `B`, y busca self._B__x que no existe.

Para arreglarlo, como se piensa heredar el atributo, es mejor usar la convención `self._x` para proteger el atributo, pero que pueda ser heredado.

19) Composición y fachada

Completa para exponer solo un método seguro de un objeto interno.

```
class _Repositorio:
 def __init__(self):
      self._datos = {}
 def guardar(self, k, v):
      self._datos[k] = v
 def _dump(self):
      return dict(self._datos)

class Servicio:
 def __init__(self):
      self.__repo = _Repositorio()

 # Expón un método 'guardar' que delegue en el repositorio,
# pero NO expongas _dump ni __repo.
```

Respuesta:

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

20) Mini-kata

Escribe una clase `ContadorSeguro` con:

- atributo “protegido” `_n`
- método `inc()` que suma 1
- propiedad `n` de solo lectura
- método “privado” `__log()` que imprima `"tick"` cuando se incrementa Muestra un uso básico con dos incrementos y la lectura final.

```
class ContadorSeguro:
    def __init__(self):
        self._n = 0   #atributo “protegido”
    def inc(self):
        self._n += 1  #suma 1
        self.__log()

    @property         #propiedad de solo lectura
    def n(self):
        return self._n

    def __log(self):  #metodo “privado”
        print("tick")


# Uso básico
c = ContadorSeguro()
c.inc()   # tick
c.inc()   # tick

print("Valor final:", c.n)
```