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`

2) Salida del programa

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

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


'_A__secret')) ¿Qué imprime?
```

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.

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?

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?

6) Identifica el error

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

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

¿Qué ocurre y por qué?

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.

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?

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.

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.

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):
 ______

 @saldo.setter
 def saldo(self, value):
 # Validar no-negativo
 ______
```

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.

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
```

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
```

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`.

16) Elección de convención

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

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
```

Propón una corrección.

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
```

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.
```

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.

Taller de Python — Resolución

29 de September de 2025

Parte A. Conceptos y lectura de código

1) Atributos accesibles desde a: Existen A, B y D; C no. (Existen a.x, a._y y a._A__z; a.__z no).

2) Salida:

```
False True
```

3) Verdadero/Falso (+ explicación):
a) Falso. '_' es solo una convención; no bloquea acceso.
b) Falso. '__' aplica name mangling y puede accederse como _Clase__atrib.
c) Verdadero. El mangling usa el nombre de la clase donde se define.

4) Lectura de código: imprime 'abc'. _token es solo convención y la subclase hereda ese atributo sin restricción real.

5) Name mangling en herencia:

```
(2, 1)
# Sub.__v -> _Sub__v = 2 y Base.__v -> _Base__v = 1
```

6) Identifica el error: con __slots__ = ('x',) solo se permiten atributos listados; al hacer c.y = 20 lanza AttributeError.

7) Rellenar espacios: un “protegido por convención” empieza con '_'. Ejemplo:

```
self._dato = 99
```

8) Métodos “privados”: imprime:

```
True False True
# Existe _step; __tick no (mangleado); _M__tick sí.
```

9) Acceso a __data (solo para comprobar):

```
print(s._S__data)
```

10) dir y mangling: el nombre más probable es '_D__a', porque '__a' se manglea; 'a' no es un atributo.

Parte B. Encapsulación con @property y validación

11) `saldo` nunca 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):
if value < 0:
raise ValueError("El saldo no puede ser negativo")
self._saldo = value
```

12) Propiedad de solo lectura temperatura_f:

```
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` siempre str:

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

@property
def nombre(self):
return self._nombre

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

14) Vista de solo lectura de colección:

```
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) Motor con validación 0–200:

```
    class Motor:
def __init__(self, velocidad):
     self._velocidad = 0
  self.velocidad = velocidad

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

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

16) ¿Cuándo usar _atributo vs __atributo?

- Usa _atributo para marcar API interna/no pública que puede cambiar; es una convención.
- Usa __atributo cuando quieras evitar colisiones de nombres en herencia (mangling) o reforzar que algo no debe tocarse desde fuera.

17) Fuga de encapsulación y corrección:

```
    class Buffer:
 def __init__(self, data):
  self._data = list(data)
   def get_data(self):
return self._data  # devuelve la lista interna (se puede mutar desde fuera)
```

```
# Corrección (devolver copia o tupla):
        def get_data(self):
return tuple(self._data)   # o: return self._data.copy()
```

18) Herencia y mangling: fallará en B.get porque self.__x se manglea como _B__x, distinto de _A__x. Arreglos: acceder a self._A__x, o mejor, exponer un getter en A, o cambiar a _x.

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

 class B(A):
   def get(self):
 return self._get_x()
```

19) Composición y fachada:

```
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()

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

20) Mini-kata ContadorSeguro:

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

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

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

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

    # Uso básico
  c = ContadorSeguro()
c.inc()   # imprime: tick
c.inc()   # imprime: tick
    print(c.n)  # 2
```