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.