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: Las opciones A, B y D son correctas.
Opción A: Es correcta porque el atributo x es publico
Opción B: Es correcta porque el atributo _y esta protegido, pero esto no restringe su acceso
Opción C: No es correcta porque el atributo __z es privado y provocaría un AtributeError
Opción D: Es correcta porque, aunque el atributo __z es privado, con el name mangling se puede acceder sin problemas.

Sergio Gabriel Chaves Mosquera – sechaves@unal.edu.co
Estudiante de Ingeniería de Sistemas y Computación

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: Imprime False True, porque en la primera pregunta, no contiene '__secret' porque ya se establece anteriormente que a=A(), entonces el atributo queda '_A__secret' lo que hace que en la segunda pregunta, la respuesta sea True

3) Verdadero/Falso (explica por qué)

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

Respuesta: Falso, porque él _ lo único que hace es la convención de público a protegido, pero esto no hace ninguna restricción, es una convención para la organización del programador

b) El prefijo __ hace imposible acceder al atributo.

Respuesta: Falso. Aunque "dificulta" el acceso a como lo seria si el atributo tuviera _ o solamente no tuviera ningún prefijo, aun se puede acceder mediante la estructura _clase__atributo

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

Respuesta: Verdadero. El name mangling lo que permite es acceder a atributos añadiendo como prefijo el nombre de la clase, manteniendo esta estructura: _nombredelaclase__nombredelatributo

Sergio Gabriel Chaves Mosquera – sechaves@unal.edu.co
Estudiante de Ingeniería de Sistemas y Computación

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: "abc". No hay error de acceso debido a la herencia, porque class Sub(Base): tiene herencia con la clase Base, entonces hereda tambien el metodo self._token = "abc"

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". Sub sobreescribe los datos de la clase Base con su metodo show(), debido a la herecia, al cual le da el valor de 2 al atributo __v, y luego con super() se crea el nuevo atributo __v = 1

Sergio Gabriel Chaves Mosquera – sechaves@unal.edu.co
Estudiante de Ingeniería de Sistemas y Computación

6) Identifica el error

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

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

¿Qué ocurre y por qué?

Respuesta: Genera un error, porque el atributo "y" no esta declarado ni en el metodo __slots__ ni se esta llamando de manera correcta desde c

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: self._nombredeunatributo=99

Cualquier nombre que comience con _

8) Lectura de métodos "privados"

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

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

Sergio Gabriel Chaves Mosquera – sechaves@unal.edu.co
Estudiante de Ingeniería de Sistemas y Computación

```
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: Imprime "True False True"
Porque si existe el metodo step() en M y es protegido, False pq esta llamando al metodo __tick como si fuera de la misma clase m, y no se puede, porque es privado, por lo que la ultima es True, pq aqui si lo llama de manera correcta.

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)

Sergio Gabriel Chaves Mosquera – sechaves@unal.edu.co
Estudiante de Ingeniería de Sistemas y Computación

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`
Porque pues `__a` no se puede porque estamos llamando antes d=D() entonces toca hacer el name mangling para poder acceder a este atributo privado y a no se puede porque es privado, entonces si o si, tiene que tener la nomenclatura antes de prefijo __
La unica que cumple con la condicion seria `_D__a`

Sergio Gabriel Chaves Mosquera – sechaves@unal.edu.co
Estudiante de Ingeniería de Sistemas y Computación

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")
            else:
                  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.

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

Sergio Gabriel Chaves Mosquera – sechaves@unal.edu.co
Estudiante de Ingeniería de Sistemas y Computación

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 tipo string")
		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
	@property
	def items(self):
	  return tuple(self._items)
```

Sergio Gabriel Chaves Mosquera – sechaves@unal.edu.co
Estudiante de Ingeniería de Sistemas y Computación

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`.

```
@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: Las convenciones _atributo, las usaria para todo lo que tenga que estar relacionado entre clases, como por ejemplo en objetos como Estudiante, en las que tambien esten relacionadas las clases Pregrado y Posgrado, las usaria para los atributos que tengan en comun se relacionen entre ellas, sin ser publicos con todas las demas clases, como _matricula, _avance, etc. Las convenciones __atributo, las usaria para todo lo que sea especifico de la clase y por privacidad o simplemente comodidad, no deba ser compartido con otras clases, como por ejemplo en la clase Estudiante, atributos como __documentoid, __pbm, entre otras.

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.

Respuesta: El problema es que el método get_data() devuelve directa la lista interna _data. No devuelve una copia, sino la lista original.

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

   def get_data(self):
      # Devolvemos una tupla, que es una copia inmutable.
      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: Este codigo fallara en la ultima linea porque “return self.__x” esta escrito de manera incorrecta, ya que para mencionar un atributo privado, aunque se tenga herencia, se debe hacer name mangling con el prefijo _nombredelaclase.

```
      return self._A__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.
```

```
    #Aqui iria el codigo brindado anteriormente, class _Repositorio, class Servicio...
    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
    def __log(self):
        print("tick")
    def inc(self):
        self._n += 1
        self.__log()
    @property
    def n(self):
        return self._n
contador = ContadorSeguro()
contador.inc()
contador.inc()
print(contador.n)
```

Sergio Gabriel Chaves Mosquera – sechaves@unal.edu.co
Estudiante de Ingeniería de Sistemas y Computación