**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: o Público: nombre o Protegido (convención): \_nombre

o 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) a.x

1. a.\_y
2. a.\_\_z
3. a.\_A\_\_z

(a.x) es un atributo de tipo publico, existe como atributo accesible.  
(a.\_y) es un atributo de tipo protegido, también es accesible dentro de la clase o de subclases.  
En el caso del uso de doble guion bajo (\_\_atributo) no crea atributos verdaderamente privados, sino que activa el name mangling, es decir el cambio automático de nombre a la forma (\_<Clase>\_\_atributo).  
Este mecanismo no impide el acceso externo, pero **reduce los conflictos en la herencia** y protege los atributos de ser sobrescritos accidentalmente.

**2) Salida del programa** class A: def \_\_init\_\_(self): self.\_\_secret = 42

a = A()

print(hasattr(a, '\_\_secret'), hasattr(a, '\_A\_\_secret'))

¿Qué imprime?

Imprime “False True” El atributo \_\_secret no se encuentra porque Python lo renombra internamente a \_A\_\_secret. Por eso, el primer hasattr da False y el segundo da True.

**3) Verdadero/Falso (explica por qué)**

1. El prefijo \_ impide el acceso desde fuera de la clase.
2. El prefijo \_\_ hace imposible acceder al atributo.
3. El name mangling depende del nombre de la clase.

RESPUESTA//   
C. El atributo \_\_x se transforma en \_ClassName\_\_x — por eso el nombre de la clase es parte del nombre final. (Regla: el prefijo de la clase en el mangling se basa en el nombre de la clase donde se definió el atributo).

1. **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?

\_token está declarado con un único guion bajo (\_token), que es convención de “protegido”, no una restricción. La subclase hereda el atributo y puede acceder a él sin error. El guion bajo no cambia el nombre del atributo, solo sugiere un uso interno.

IMPRIME: abc

1. **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?

(self.\_\_v) en **Base** se cambia a (self. Base\_\_v) (valor 1).

(self.\_\_v) en **sub** se cambia a (self. sub\_\_v) (valor 2).

Son **dos atributos distintos** en la misma instancia porque el cambio incluye el nombre de la clase donde se definió. En show() (self.\_\_v) → (\_Sub\_\_v (2)) y (self.\_Base\_\_v) → (1).

SALIDA: (2, 1)

1. **Identifica el error**

class Caja:

\_\_slots\_\_ = ('x',)

c = Caja() c.x = 10

c.y = 20

¿Qué ocurre y por qué?

RTA//  
 Al ejecutar c.y = 20 se lanza el error: (AttributeError: 'Caja' object has no attribute 'y')  
\_\_slots\_\_ limita los atributos que pueden añadirse dinámicamente a las instancias (aquí sólo 'x'). Intentar asignar y no permitido provoca AttributeError. \_\_slots\_\_ se usa para ahorrar memoria y controlar la API de atributos. (Si se quiere permitir nuevos atributos habría que incluir '\_\_dict\_\_' en \_\_slots\_\_ o no usar \_\_slots\_\_.)

1. **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.

self.\_valor = 99

un atributo “protegido por convención” se nombra con **un guion bajo inicial** (\_nombre). Cualquier identificador con un solo \_ al inicio cumple esa convención

1. **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?

IMPRIME: True False True

Porque \_step es público/convención “protegido”: existe → hasattr(m, '\_step') es True. (el primer True), \_\_tick fue definido con doble guion bajo, por lo que se cambia a \_M\_\_tick internamente; por eso hasattr(m, '\_\_tick') es False (el segundo False), (no existe literal \_\_tick) y hasattr(m, '\_M\_\_tick') es True (el tercer y utlimo true). Así se demuestra que métodos con \_\_ están accesibles mediante su nombre mangled.

**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.

RTA// print(s.\_S\_\_data)

(\_\_data) se transforma internamente a (\_S\_\_data) (clase S), por eso podemos acceder con (s.\_S\_\_data.) Esto demuestra que el doble guion bajo no elimina el atributo, solo lo renombra para evitar choques accidentales.

**10) Comprensión de dir y mangling** class D: def \_\_init\_\_(self): self.\_\_a = 1 self.\_b = 2

self.c = 3

d = D()

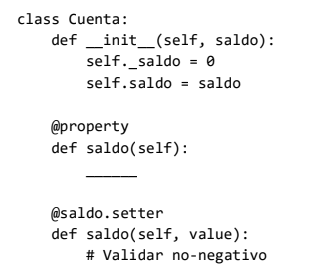
names = [n for n in dir(d) if 'a' in n] print(names)

¿Cuál de estos nombres es más probable que aparezca en la lista: \_\_a, \_D\_\_a o a? Explica.

RTA// \_D\_\_a es la más probable que aparezca.

\_\_a fue cambiada por Python a \_D\_\_a (incorporando el nombre de la clase). **dir(d)** lista los nombres reales del objeto, así que verá la forma **mangled** \_D\_\_a. No aparecerá un nombre literal \_\_a (porque ya fue reemplazado) ni a. Por eso la entrada con 'a' en dir(d) será algo como '\_D\_\_a'.

1. **Saldo nunca negativo**
   1. CÓDIGO BASE:



* 1. CÓDIGO RESUELTO:

class Cuenta:

def \_\_init\_\_(self, saldo):

self.\_saldo = 0

self.saldo = saldo

@property

def saldo(self):

return self.\_saldo

@saldo.setter

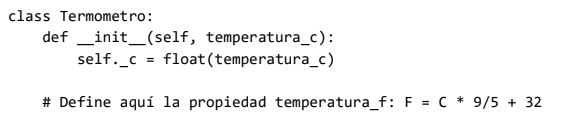
def saldo(self, value):

if value < 0:

raise ValueError("El saldo no puede ser negativo")

self.\_saldo = value

1. **Convierte temperatura\_f en un atributo de solo lectura que se calcula desde temperatura\_c.**
   1. CÓDIGO BASE:



* 1. CÓDIGO RESUELTO:

class Termometro:

def \_\_init\_\_(self, temperatura\_c):

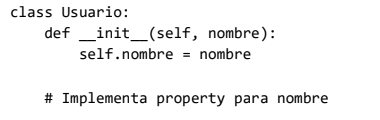
self.\_c = float(temperatura\_c)

@property

def temperatura\_f(self):

return self.\_c \* 9/5 + 32

1. **Haz que nombre sea siempre str. Si asignan algo que no sea str, lanza TypeError.**
   1. CÓDIGO BASE:



* 1. CÓDIGO RESUELTO:

class Usuario:

def \_\_init\_\_(self, nombre):

self.nombre = nombre

@property

def nombre(self):

return self.\_nombre

@nombre.setter

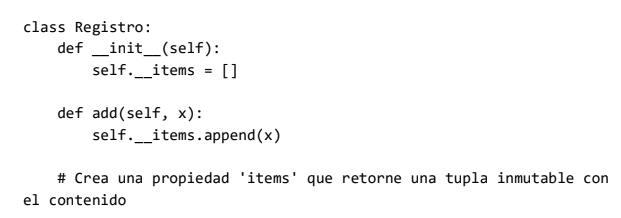
def nombre(self, value):

if not isinstance(value, str):

raise TypeError("El nombre debe ser una cadena (str)")

self.\_nombre = value

1. **Expón una vista de solo lectura de una lista interna.**
   1. CÓDIGO BASE:



* 1. CÓDIGO RESUELTO:

class Registro:

def \_\_init\_\_(self):

self.\_\_items = []

def add(self, x):

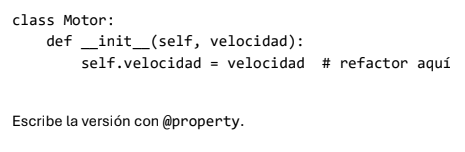
self.\_\_items.append(x)

@property

def items(self):

return tuple(self.\_\_items)

1. **Refactoriza para evitar acceso directo al atributo y validar que velocidad sea entre 0 y 200.**
   1. CÓDIGO BASE:



* 1. CÓDIGO RESUELTO:

class Motor:

def \_\_init\_\_(self, velocidad):

self.\_velocidad = 0

self.velocidad = velocidad

@property

def velocidad(self):

return self.\_velocidad

@velocidad.setter

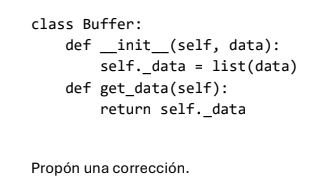
def velocidad(self, value):

if not (0 <= value <= 200):

raise ValueError("La velocidad debe estar entre 0 y 200")

self.\_velocidad = value

1. **Explica con tus palabras cuándo usarías \_atributo frente a \_\_atributo en una API pública de una librería.**
   1. RESPUESTA: En Python cuando uso un atributo con un solo guion bajo (\_atributo), lo hago como una forma de advertencia, es decir estoy diciendo que ese dato es interno de la clase y que aunque se pueda acceder, no debería usarse directamente, en cambio, si le pongo doble guion bajo (\_\_atributo) ya no es solo una convención, porque Python cambia el nombre internamente y se vuelve más difícil de acceder desde afuera, además de que así evito choques de nombres cuando trabajo con herencia. En pocas palabras, uso \_ para señalar que algo no es parte de la API pública y \_\_ cuando quiero protegerlo de verdad.
2. **¿Qué problema hay aquí?**
   1. CODIGO BASE:



* 1. CÓDIGO RESUELTO:
* Problema: El método get\_data() devuelve la lista interna \_data directamente

class Buffer:

def \_\_init\_\_(self, data):

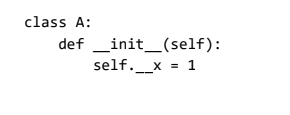
self.\_data = list(data)

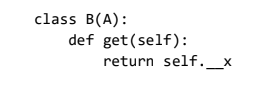
def get\_data(self):

return tuple(self.\_data)

* Se devuelve una tupla inmutable solo para que se pueda leer y no modificar

1. **¿Dónde fallará esto y cómo lo arreglas?**
   1. CÓDIGO BASE:





En la clase A, el atributo \_\_x se transforma por name mangling en \_A\_\_x.

En la clase B, cuando se escribe self.\_\_x, NO es el mismo atributo, ahi se transforma en \_B\_\_x

* 1. CÓDIGO RESUELTO:

class A:

def \_\_init\_\_(self):

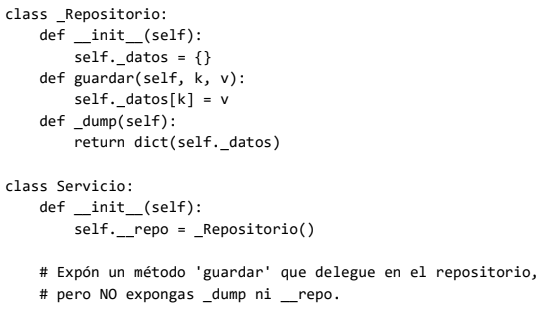
self.\_x = 1

class B(A):

def get(self):

return self.\_x

1. **Completa para exponer solo un método seguro de un objeto interno.**
   1. CÓDIGO BASE



* 1. SOLUCIÓN:

class Servicio:

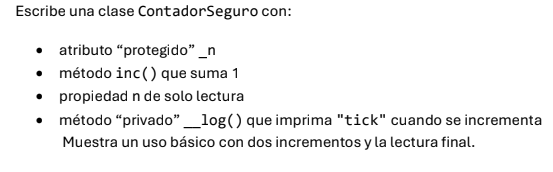
def \_\_init\_\_(self):

self.\_\_repo = \_Repositorio() # objeto interno

def guardar(self, k, v):

self.\_\_repo.guardar(k, v) # delega en el repositorio

1. **Escribe una clase ContadorSeguro con**
   1. REQUERIMIENTOS:



* 1. SOLUCIÓN:

class ContadorSeguro:

def \_\_init\_\_(self):

self.\_n = 0 # atributo protegido

def inc(self):

self.\_n += 1

self.\_\_log() # se llama al método privado cada vez que se incrementa

@property

def n(self):

return self.\_n # solo lectura

def \_\_log(self):

print("tick")