Diego Armando Pérez Solano – Ingeniería de sistemas

Programación orientada a objetos

**Taller: POO y modificadores de acceso en**

**Python**

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

solución:

De las opciones dadas, todas menos la opción C son accesibles desde a, a.x no tiene ningún tipo de seguridad, a.\_y se cuenta como protegida, aunque esto no hace que sea imposible acceder a su información y x.\_A\_\_z es la opción de name mangling con la cual se puede acceder sin problema a la información de \_\_z que normalmente daría error por estar como privada

2) Salida del programa

class A:

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

self.\_\_secret = 42

a = A()

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

¿Qué imprime?

solución:

Imprime dos valores, un “false” y un “true”, ¿Por qué? Esto es debido a el “hasattr” lo que hace esta función es comprobar si un atributo existe dentro de un objeto, en esta situación se están comprobando dos cosas e imprimiendo el resultado en valor booleano dentro del objeto “a” que es un objeto perteneciente a la clase A, en el primer hasattr se esta comprobando la existencia del atributo \_\_secret dentro de a, lo cual existe como tal pero no es el nombre con el cual Python ahora reconoce este atributo porque es privado por eso imprime el resultado booleano “False” porque como no reconoce al atributo con ese nombre, no existe para Python, por el otro lado \_A\_\_secret es la forma correcta de escribir el atributo \_\_secret después del name mangling de Python por lo que ahora este si reconoce al atributo e imprime el valor booleano con “True”



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.

solución:

1. Falso, Es falso porque el prefijo \_ aunque indique el atributo o modulo como protegido este no impide su acceso desde fuera de la clase
2. Falso, Es falso porque el prefijo \_\_ impide el uso directo de los atributos o módulos por fuera de las clases únicamente, dentro de las clases aun es usable normalmente
3. Verdadero, el name mangling se escribe como “\_Nombre de clase\_\_atributo” por lo que cada name mangling cambia con respecto 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?

solución:

Al final del código se va a imprimir el atributo self.\_token, primero que nada no se va a crear un error de acceso porque \_token solamente marca al atributo como protegido pero no protegido por lo que no tiene name mangling y por ende aun es accesible desde cualquier parte, ahora por la parte de lo que se imprime, va en partes, primero se crea la clase Base en la cual se tiene un inicializador con el atributo \_token después de esto se crea otra subclase llamada Sub la que hereda completamente la clase Base, esto hace que cuente con el inicializador y atributos de Base. Sub inicia un método llamado revelar el cual devuelve \_token y al final por fuera de la clase se imprime “sub().reveal()” el Sub() crea un objeto que pertenezca a la clase Sub y hereda todas las características de Base y .reveal inicia el método de Sub, así imprimiendo el valor que retorna el cual es \_token, por esto al final se imprime el mismo \_token el cual es “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?

solución:



¿Por qué?

Se crea una super clase llamada Base, esta cuenta con un atributo protegido .\_\_v, después de esto se crea una subclase la cual hereda a la superclase Base, Sub, después se usa el inicializador de la clase y con la función “supper()” traemos el inicializador de la superclase Base y con ello creamos él .\_\_v de Base, continuamos ahora con el atributo .\_\_v de Sub y para esto fue usada la función super() para poder tener los atributos .\_\_v de ambas clases sin que se sobrescriban, se termina el código con un método de la subclase Sub el cual devuelve dos valores “self.\_\_v” que es el .\_\_v de Sub y “self.\_Base\_v” que por ser el .\_\_v de Base paso por name mangling y ahora se reconoce como .\_Base\_\_v Con todo el código listo finalizamos con la impresión del método de Sub creando un objeto de sub y aplicándole el método para obtener el valor (2, 1), la impresión no da error porque el .\_\_v de Sub se maneja dentro de la clase y el .\_\_ v de Base se trabaja con su respectivo name mangling al estar en otra clase

6) Identifica el error

class Caja:

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

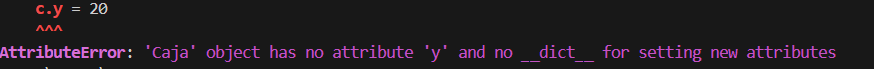
c = Caja()

c.x = 10

c.y = 20

¿Qué ocurre y por qué?

solución:

El error recae en c.y, ¿por qué? Porque y no está definido a diferencia de x por esto cae el error en c.y y al ejecutar le programa se muestra 

Pero entonces ¿por qué c.x si está definido? Por el método \_\_slots\_\_ , el método slots hace que la clase a la que se la aplique Utilice únicamente los atributos que se le sean definidos “x” en este caso

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.

solución:

Para que el atributo de la clase B sea protegido, se usa el prefijo \_ un ejemplo seria

self.\_vista = 99

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?

solución:



Muestra “True”, “False”, “True”, esto lo hace porque primero, la función hasattr solo devuelve valores Booleanos al ser una función que comprueba algo, segundo, devuelve valores porque en Python los métodos también cuentan como atributos, simplemente que atributos de función, y tercero, devuelve True por \_step porque al estar solo protegida conserva su nombre, False porque \_\_tick ya no tiene su nombre original al ser privado, paso por un name mangling y Python ya no reconoce a \_\_tick como un atributo existente, True porque \_M\_\_tick es el nuevo nombre con el que \_\_tick existe al pasar por el name mangling

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

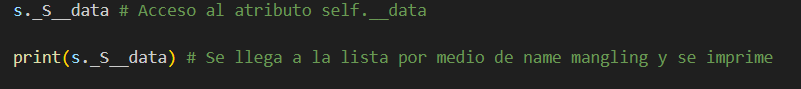
clase:

# Escribe una línea que obtenga la lista usando name mangling y la

imprima.

Escribe la línea solicitada.

solución:





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.

solución:

Es mas probable que aparezca \_D\_\_a, en la línea de names lo que se hace es un ciclo for, en el cual por cada item en dir(d) (esta función la cual contiene todos los objetos del objeto d), que contengan la letra “a” en el objeto d se añadirán a una lista, en esta lista como “a” esta privada por su prefijo \_\_ se le ha modificado el nombre por \_D\_\_a en lugar de \_\_a por lo que a la lista se le añadirá \_D\_\_a porque

1. \_\_a ya no existe como atributo que Python reconozca
2. a en primer lugar no existía

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):

\_\_\_\_\_\_

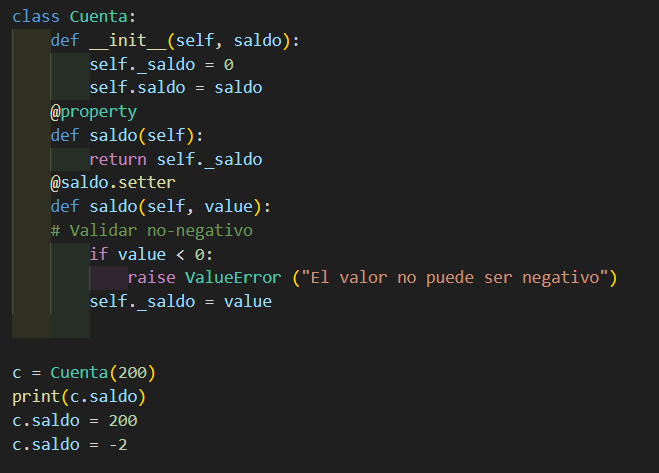
@saldo.setter

def saldo(self, value):

# Validar no-negativo

\_\_\_\_\_\_

solución:





Texto

El contenido generado por IA puede ser incorrecto.

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

solución:

Escribe la propiedad.

Texto

El contenido generado por IA puede ser incorrecto.



13) Invariante con tipo

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

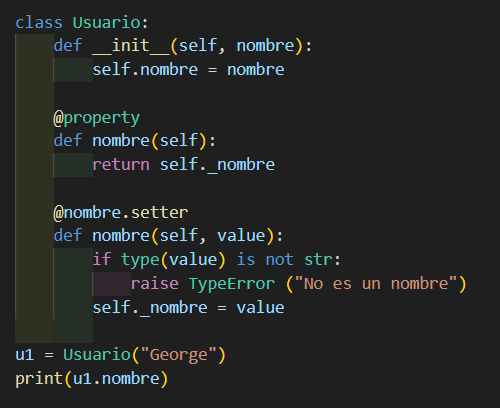
class Usuario:

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

self.nombre = nombre

# Implementa property para nombre

solución:





Captura de pantalla de un celular

El contenido generado por IA puede ser incorrecto.

14) Encapsulación de colección

Expón una vista de solo lectura de una lista interna.

class Registre:

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

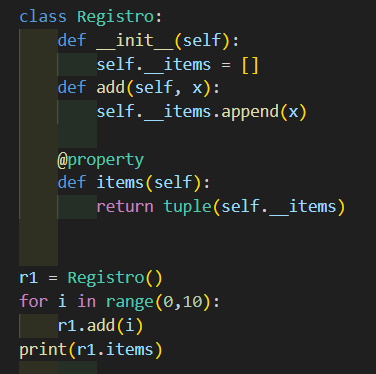
self.\_\_items = []

def add(self, x):

self.\_\_items.append(x)

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

solución:





15) Refactor a encapsulación

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

200.

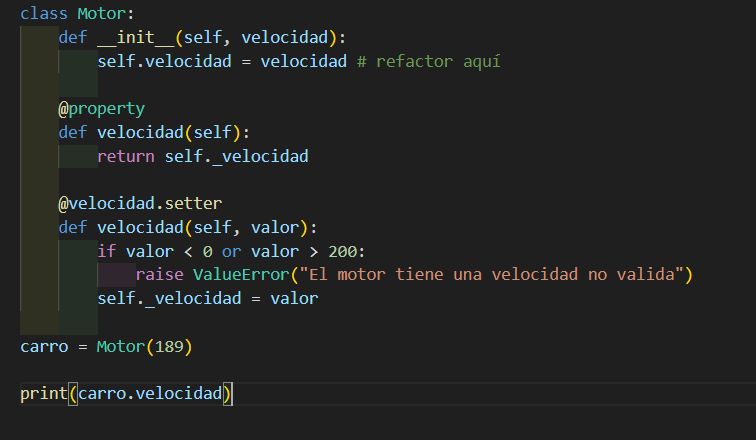
class Motor:

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

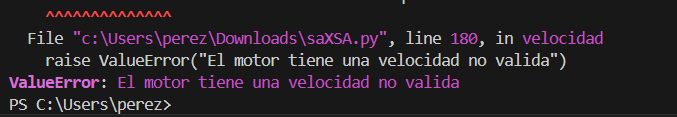
self.velocidad = velocidad # refactor aquí

Escribe la versión con @property.

solución:







16) Elección de convención

Explica con tus palabras cuándo usarías \_atributo frente a \_\_atributo en una API

pública de una librería.

solución:

\_atributo es un atributo protegido y \_\_atributo es un atributo privado por lo que en una API publica usaría \_atributo frente a \_\_atributo en valores que funcionen dentro de clases, subclases pero que no necesita de name mangling, ósea atributos que no necesiten del máximo nivel de seguridad que tiene Python “los privados”

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.

solución:

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

solución:

El código falla en la línea “return self.\_\_x”, pero ¿Por qué? Muy fácil, esto es porque por mas que se haya heredado la clase A el atributo se encuentra privado, ósea que cuenta con name mangling, el programa no está retornando nada porque self.\_\_x ya no se llama así ahora se llama self.\_A\_\_x haciendo referencia a la clase A con su atributo privado

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.

solución:

20) Mini-kata

Escribe una clase ContadorSeguro con:

• atributo “protegido” \_n

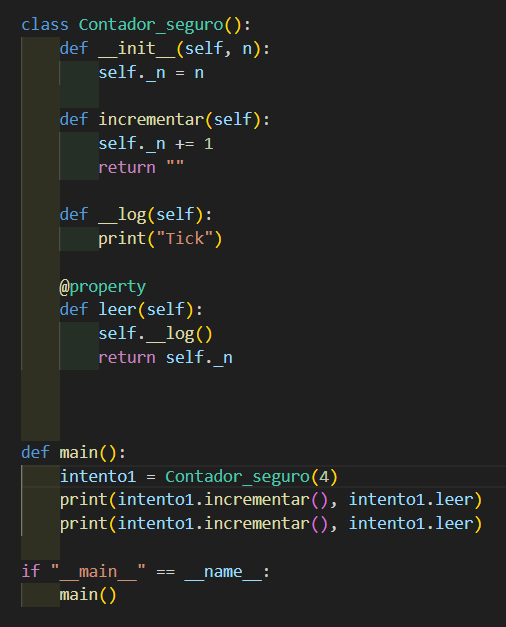
• método inc() que suma 1

• 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.

solución:



Interfaz de usuario gráfica, Texto

El contenido generado por IA puede ser incorrecto.

Texto

El contenido generado por IA puede ser incorrecto.