RESUMEN CLASES EN PYTHON

1. **En Python, el uso de uno o dos guiones bajos antes de un atributo (`_atributo` o `__atributo`) tiene significados diferentes relacionados con el nivel de protección y accesibilidad del atributo dentro de una clase. Los atributos en python pueden ser públicos, "privados o protegidos"**

1.1. Un solo guion bajo (`_atributo`)

- **Convención**: Un guion bajo al inicio de un nombre (como `_atributo`) es una **convención** para indicar que un atributo o método es "protegido" o "privado" y **no debe ser accedido directamente** fuera de la clase.
- **Función**: No impide el acceso directo, pero indica a otros desarrolladores que el atributo es para uso interno de la clase y no debería ser modificado externamente.
- **Acceso**: Sigue siendo accesible desde fuera de la clase, por lo que técnicamente no es privado.

Ejemplo:

```
class Ejemplo:
    def __init__(self):
        self._atributo = 10  # Atributo "protegido"

obj = Ejemplo()
print(obj._atributo)  # Aunque es accesible, la convención sugiere que no
lo usemos directamente.
```

Prefijo	Nivel de Protección	Acceso Directo desde Fuera	Uso
_atributo	Convención de "protegido"	Sí	Indica que es para uso interno
__atributo	Name Mangling / "privado"	No, pero accesible como _Clase__atributo	Oculta el atributo para evitar conflictos accidentales
atributo	público	si	Uso genérico

1.2. Dos guiones bajos (`__atributo`)

- **Name Mangling**: Cuando un atributo comienza con dos guiones bajos (`__atributo`), Python aplica una técnica llamada **name mangling** para "enmarañar" el nombre del atributo. El atributo se renombra internamente con un prefijo que incluye el nombre de la clase, lo que dificulta el acceso accidental desde fuera de la clase.
- **Privacidad**: Se usa para dar una mayor sensación de privacidad en comparación con un solo guion bajo.
- **Acceso**: Aunque todavía es accesible externamente, el nombre mangling hace que sea más complicado (y menos común) acceder a él.

Ejemplo:

```
class Ejemplo:
    def __init__(self):
        self.__atributo = 20  # Atributo "privado" debido a name mangling

obj = Ejemplo()
# Esto lanzará un error porque __atributo no es directamente accesible:
# print(obj.__atributo)

# Podemos acceder a él con name mangling si conocemos el nombre interno:
print(obj._Ejemplo__atributo)  # Resultado: 20
```

Conclusión:

- _atributo es solo una convención, mientras que __atributo utiliza name mangling para "ocultar" el nombre, aunque ambos siguen siendo accesibles externamente si se quiere.
- Es preferible usar _atributo para indicar que es para uso interno y __atributo cuando se desea evitar colisiones de nombre en clases heredadas o dar una mayor protección.

Ejemplo: En este caso, imaginemos que tenemos una clase `CuentaBancaria` que tiene un atributo privado `_saldo`. Aunque normalmente no deberíamos acceder a este atributo desde fuera de la clase, te muestro cómo hacerlo para entender el concepto:

```
class CuentaBancaria:
    def __init__(self, saldo_inicial):
        self._saldo = saldo_inicial  # atributo "privado"

    def depositar(self, monto):
        self._saldo += monto
        return self._saldo

    def retirar(self, monto):
        if monto <= self._saldo:
            self._saldo -= monto
            return self._saldo
        else:
            print("Fondos insuficientes")
            return self._saldo

# Creación de una instancia de la clase
mi_cuenta = CuentaBancaria(1000)


# Acceso normal a los métodos públicos
mi_cuenta.depositar(500)
print("Saldo después de depósito:", mi_cuenta.retirar(200))

# Acceso "no recomendado" al atributo privado
print("Accediendo directamente al saldo privado:", mi_cuenta._saldo
```

2. **Hay varias formas de establecer los métodos getter y setter en python . Se puede usar la forma tradicional, estableciendo los métodos, bien usando "métodos decoradores" o bien usando los métodos predefinidos por python setattr o getattr**

   Vamos a verlo con ejemplos:

   ### 2.1. **Usando "métodos decoradores"**

   Ejemplo usando la clase `CuentaBancaria`, pero esta vez usando `@property` para controlar el acceso al saldo:

```
class CuentaBancaria:
    def __init__(self, saldo_inicial):
        self._saldo = saldo_inicial  # Atributo privado

    @property
    def saldo(self):
        """Devuelve el saldo actual de la cuenta"""
        return self._saldo

    @saldo.setter
    def saldo(self, monto):
        """Permite establecer el saldo, solo si el monto es positivo"""
        if monto >= 0:
            self._saldo = monto
        else:
            raise ValueError("El saldo no puede ser negativo")

    def depositar(self, monto):
        self._saldo += monto
        return self._saldo

    def retirar(self, monto):
        if monto <= self._saldo:
            self._saldo -= monto
            return self._saldo
        else:
            print("Fondos insuficientes")
            return self._saldo

# Creación de una instancia de la clase
mi_cuenta = CuentaBancaria(1000)

# Acceso a la propiedad `saldo`
print("Saldo inicial:", mi_cuenta.saldo)

# Depósito y consulta del saldo
mi_cuenta.depositar(500)
print("Saldo después de depósito:", mi_cuenta.saldo)

# Intento de establecer un nuevo saldo

mi_cuenta.saldo = 300  # Uso de @saldo.setter
print("Saldo modificado manualmente:", mi_cuenta.saldo)

# Intento de asignar un saldo negativo (provoca una excepción)
try:
    mi_cuenta.saldo = -500
except ValueError as e:
    print(e)
```

Explicación

1. **Definición de propiedad `saldo`**: Usamos `@property` para definir el método `saldo`, que devuelve el valor de `_saldo`. Esto permite acceder a `_saldo` desde fuera de la clase usando `mi_cuenta.saldo` en lugar de `mi_cuenta._saldo`.
2. **Método `saldo.setter`**: Definimos un método `saldo` con `@saldo.setter` para permitir modificar el saldo de manera controlada. Solo se asigna un nuevo valor si es positivo; si no, lanza un `ValueError`.
3. **Ventaja**: Los usuarios de la clase acceden a `mi_cuenta.saldo` como si fuera un atributo normal, sin saber que hay lógica detrás. Esto ayuda a mantener el encapsulamiento y el control sobre el valor interno de `_saldo`.

Beneficios de usar `@property`

- **Encapsulamiento**: Controla cómo se leen y escriben los atributos.
- **Simplicidad de uso**: Permite acceder a métodos como si fueran atributos.
- **Validación**: Facilita la aplicación de reglas de validación para mantener los valores correctos.

2.2. Usando métodos tradicionales

```
class Producto:
    def __init__(self, nombre):
        self._nombre = nombre  # Usamos el guión bajo para indicar que es "privado"

    def get_nombre(self):
        return self._nombre

    def set_nombre(self, nombre):
        if not nombre:
            raise ValueError("El nombre no puede estar vacío")
        self._nombre = nombre



# Uso
producto = Producto("Laptop")
print(producto.get_nombre())  # Acceso
producto.set_nombre("Tablet")  # Modificación
print(producto.get_nombre())
```

2.3 Métodos getattr, setattr

En Python, `setattr` y `getattr` son funciones ya definidas que permiten manipular los atributos de un objeto de manera dinámica

→setattr(objeto, nombre_atributo, valor)

La función `setattr` se usa para asignar un valor a un atributo de un objeto. Si el atributo ya existe, `setattr` actualizará su valor; si no existe, `setattr` lo creará.

Sintaxis:

```
setattr(objeto, 'nombre_atributo', valor)
```

Ejemplo:

```
class Persona:

persona = Persona()

# Crear un nuevo atributo dinámicamente
setattr(persona, 'nombre', 'Juan')
print(persona.nombre)  # Salida: Juan

# Cambiar el valor del atributo
setattr(persona, 'nombre', 'Carlos')
print(persona.nombre)  # Salida: Carlos
```

→getattr(objeto, nombre_atributo[, valor_por_defecto])

La función `getattr` se usa para obtener el valor de un atributo de un objeto. Si el atributo no existe, `getattr` lanzará una excepción `AttributeError`, a menos que se proporcione un valor por defecto, el cual será devuelto en caso de que el atributo no esté presente.

Sintaxis:

```
getattr(objeto, 'nombre_atributo', valor_por_defecto)
```

Ejemplo:

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

persona = Persona("Ana")

# Obtener el valor de un atributo existente
print(getattr(persona, 'nombre'))  # Salida: Ana

# Intentar obtener un atributo inexistente con valor por defecto
print(getattr(persona, 'edad', 25))  # Salida: 25 (usa el valor por
defecto)
```

→delattr

Borra un atributo del objeto. El atributo debe existir, porque se produce un error si no se encuentra

p = Punto()

delattr(p, “nombre”)

→hasattr()

Devuelve un valor booleano indicando si un atributo existe.

p = Punto()
tiene_x= hasattr(p, “x”)
tiene_z=getattr(p,“z”)

Resumen:

- **`setattr`**: permite crear o modificar el valor de un atributo.
- **`getattr`**: permite obtener el valor de un atributo y evita errores si el atributo no existe (cuando se proporciona un valor por defecto).

Estas funciones son muy útiles para manejar objetos de manera dinámica, especialmente cuando los atributos a manipular se conocen en tiempo de ejecución.

3. **En Python, la herencia es un concepto de la programación orientada a objetos que permite que una clase (llamada "clase hija" o "subclase") herede atributos y métodos de otra clase (llamada "clase padre" o "superclase"). Esto es útil para reutilizar código y crear una jerarquía de clases relacionadas.**

3.1 Ejemplo básico de herencia

Supongamos que tenemos una clase llamada `Animal`, y queremos crear otras clases (`Perro`, `Gato`) que hereden de ella:

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

    def hacer_sonido(self):
        pass  # Método que será sobrescrito en las subclases

class Perro(Animal):
    def hacer_sonido(self):
        return "Guau!"

class Gato(Animal):
    def hacer_sonido(self):
        return "Miau!"
```

Aquí, `Perro` y `Gato` heredan de `Animal`. Ambas subclases tienen su propia implementación de `hacer_sonido`.

```
# Crear instancias de Perro y Gato
mi_perro = Perro("Firulais")
mi_gato = Gato("Misi")

# Llamar al método hacer_sonido en ambas instancias
print(f"{mi_perro.nombre} dice: {mi_perro.hacer_sonido()}")
print(f"{mi_gato.nombre} dice: {mi_gato.hacer_sonido()}")
```

Salida:

```
Firulais dice: Guau!
Misi dice: Miau!
```

3.2 Herencia múltiple

Python permite que una clase herede de múltiples clases. Esto se llama herencia múltiple.

```
class Volador:
    def volar(self):
        return "Estoy volando!"

class Pato(Animal, Volador):
    def hacer_sonido(self):
        return "Cuac!"
```

En este caso, `Pato` hereda tanto de `Animal` como de `Volador`, y puede acceder a los métodos de ambas:

```
mi_pato = Pato("Donald")
print(f"{mi_pato.nombre} dice: {mi_pato.hacer_sonido()}")
print(mi_pato.volar())
```

Salida:

```
Donald dice: Cuac!
Estoy volando!
```

Uso de `super()` para extender métodos

Cuando sobrescribimos un método de la clase padre, podemos usar `super()` para acceder a la versión original del método en la superclase.

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

    def saludar(self):
        return f"Hola, soy {self.nombre}"

class Estudiante(Persona):
    def __init__(self, nombre, universidad):
        super().__init__(nombre)  # Llamamos al constructor de la
superclase
        self.universidad = universidad

    def saludar(self):
        return super().saludar() + f" y estudio en {self.universidad}"
```

Aquí, `Estudiante` extiende `Persona` y usa `super()` para obtener el comportamiento de `saludar` de la clase padre.

```
estudiante = Estudiante("Carlos", "Universidad Nacional")
print(estudiante.saludar())
```

Salida:

```
Hola, soy Carlos y estudio en Universidad Nacional
```

La herencia permite organizar el código en estructuras jerárquicas y reutilizar funcionalidades, haciendo el código más limpio y fácil de mantener.