

Introducción a la Pseudo Orientación a Objetos en Python

Hoy vamos a aprender sobre un concepto muy importante llamado pseudo orientación a objetos. Este concepto nos ayudará a organizar nuestro código de una manera más clara y estructurada, permitiéndonos crear programas más potentes y fáciles de entender. 😊😊😊

La pseudo orientación a objetos es una aproximación a la programación orientada a objetos, adaptada para lenguajes que no tienen soporte nativo para este enfoque, como Python en su forma básica. Aunque no es una implementación completa de la programación orientada a objetos, la pseudo orientación a objetos nos permite simular algunos de sus principios fundamentales.

🤔 ¿Pero qué es la **programación orientada a objetos**? 🤔

Bueno, imagina que estás construyendo una ciudad en un videojuego. En lugar de tener que programar cada edificio y cada personaje por separado, la *programación orientada a objetos* nos permite crear "objetos" que representan entidades del mundo real, como autos, personas o casas. Cada objeto tiene un tipo, y objetos de mismo tipo tendrán características y comportamientos específicos, que podemos definir y utilizar en nuestro código.

Ahora, en la pseudo orientación a objetos en Python, usaremos una estructura llamada diccionario para representar nuestros objetos. Un diccionario es como una lista de pares clave-valor, donde cada clave es un atributo del objeto (una característica) y el valor asociado a cada clave es el valor que tiene el atributo. Por ejemplo, si queremos representar un coche, podríamos tener atributos como el color, la velocidad y la posición en la pista. Al usar diccionarios para agrupar los atributos de un objeto podemos agrupar de manera coherente la información relacionada dentro del objeto.

Además de los atributos, los objetos también pueden realizar acciones o tener comportamientos específicos. Estas acciones se implementan como **funciones**, que actúan sobre los datos del objeto. Por ejemplo, un coche puede acelerar, frenar o cambiar de dirección. Estas funciones se llaman **métodos** y nos permiten interactuar con los objetos y modificar sus **atributos**.

La pseudo orientación a objetos nos brinda una forma más estructurada y organizada de programar. Podemos crear diferentes objetos a partir de un mismo "molde" o clase, donde definimos los atributos y métodos que tendrán todos los objetos de esa clase. Luego, podemos utilizar estos objetos en nuestro código, llamando a sus métodos y accediendo a sus atributos.

Es importante destacar que la pseudo orientación a objetos en Python es una aproximación simplificada de la programación orientada a objetos completa, pero es un gran primer paso para entender los conceptos fundamentales de este enfoque. A medida que vayamos avanzando en nuestro aprendizaje, podremos profundizar en la programación orientada a objetos y explorar todas sus capacidades.

Ahora que conocemos los conceptos básicos de la pseudo orientación a objetos, ¡Veamos algunos ejemplos!

Ejemplo 1: constructorOrdenadores

En este ejemplo, vamos a ver como funciona la Pseudo Orientación a Objetos para crear objetos de tipo ordenador.

Antes de ponernos a hacer el ejemplo es importante tener claras tres definiciones técnicas de estructuras que vamos a usar al programar con Pseudo Orientación a Objetos, que son: **atributos**, **métodos** y **parámetros**. 🧐

Los **atributos** son las características que un objeto puede tener, por ejemplo, un objeto de tipo coche podría tener como atributos el número de puertas, la velocidad, el tipo de motor, el color, número de asientos, tamaño de ruedas, etc.

Los **métodos** son las estructuras que conocemos hasta ahora como **funciones**. No obstante, los métodos son funciones muy especiales, son responsables por crear objetos y también por cambiar y observar (obtener) los valores de los atributos de un objeto. Cada objeto **SIEMPRE** tendrá asociado un **método constructor**, el método constructor sirve para construir y definir un objeto con todas sus atributos iniciales. Los atributos de los objetos por su vez tendrán normalmente al menos dos métodos distintos asociados: un **método observador** (sirve para obtener el valor del atributo) y un **método modificador** (sirve para cambiar el valor del atributo). En los ejemplos veremos cada uno de los diferentes métodos con detalles.

Por último, los **parámetros**, estos también son usados en las funciones básicas, los parámetros son los valores pasados en los paréntesis de los métodos, sirven para introducir a los métodos valores que han sido generados fuera del método, pero que necesitan ser usados dentro para hacer las operaciones necesarias.

Dicho lo anterior, ahora podemos proseguir con la definición del código para generar ordenadores usando Pseudo Orientación a Objetos.

1. Métodos Constructores

Lo primero que necesitamos hacer para trabajar con Pseudo Orientación a Objetos es definir un **método constructor**. Como dicho antes, los métodos son funciones que usamos para manipular los objetos y los *métodos constructores* son los métodos responsables por definir y crear los objetos en el programa con todas sus características iniciales.

Antes de crear cualquier método constructor, tenemos que pensar qué atributos los objetos del tipo que queremos crear tienen en común y cuáles de estos son importantes para nuestro programa. En el caso de los ordenadores, hay una infinidad de atributos que podríamos establecer para crearlo, pero para este ejemplo lo haremos de forma más sencilla y nos quedaremos solo con algunas características principales de los ordenadores. Los atributos que vamos a usar son: RAM, procesador, tarjeta gráfica, almacenamiento, pantalla, teclado y ratón.

```
'''
Atributos de un ordenador:
ram
procesador
tarjeta gráfica
almacenamiento
pantalla
teclado
ratón
'''
```

Una vez hayamos definido los atributos necesarios para crear los objetos, podemos seguir con la definición del *método constructor*, usando los atributos del ordenador como parámetros para el método:

```
def constructorOrdenadores(ram, procesador, tarjeta_grafica, almacenamiento, pantalla, teclado, raton):
```

Este sería el encabezado del método constructor, mi recomendación para nombrar los métodos constructores es nombrarlos siempre por “**constructorObjeto**”, donde el “objeto” sería el tipo de cosa que el método construye, en nuestro caso, el método se llama “**constructorOrdenadores**”. En la definición del constructor proporcionamos por parámetros toda la información necesaria para crear el objeto, nosotros usaremos en ese caso todos los atributos que hemos definido. Sin embargo, vale destacar que no siempre usaremos todos los atributos de un objeto como parámetros del método constructor.

Si hay algún atributo cuyo valor se deriva de 2 otros parámetros o más, éste no será pasado como parámetro al método constructor. Por ejemplo, si el método constructor define una **persona**, y tiene como algunos de sus atributos una fecha de nacimiento y una edad, no sería necesario pasar la edad como parámetro al método constructor ya que ésta podría ser calculada y definida a partir de la fecha de nacimiento. De esa forma, dentro del constructor

haríamos los cálculos necesarios para determinar la edad de la persona a partir de la fecha de nacimiento proporcionada por parámetro.

Después de definir el encabezado del constructor, tenemos que definir el código que contendrá. En el caso de que el constructor tenga que hacer alguna operación con los parámetros pasados para obtener más atributos del objeto (como en el ejemplo del constructor de personas) hacemos el código de dichas operaciones en esta parte antes de crear el objeto definitivo. Si no, podemos saltar este paso y seguir con la creación del objeto.

Para este ejemplo, ¡estamos con suerte! No necesitamos hacer ninguna operación dentro del método constructor, ya que todo lo que nos hace falta para crear un ordenador es pasado como parámetro, con eso dispensamos la parte de sacar más atributos a través de los parámetros pasados. 😊😄🥳

De este modo, el método quedaría de la siguiente forma:

```
def constructorOrdenadores(ram, procesador, tarjeta_grafica, almacenamiento, pantalla, teclado, raton):  
  
    objeto = {  
        "tipo": "ordenador",  
        "ram": ram,  
        "procesador": procesador,  
        "tarjeta_grafica": tarjeta_grafica,  
        "almacenamiento": almacenamiento,  
        "pantalla": pantalla,  
        "teclado": teclado,  
        "raton": raton}  
  
    return objeto
```

Uy 😬😬... vamos por partes:

La función `constructorOrdenadores` toma varios parámetros como entrada, los cuales representan diferentes características de un ordenador, como la memoria RAM, el procesador, la tarjeta gráfica, el almacenamiento, la pantalla, el teclado y el ratón.

Dentro de la función, creamos un diccionario llamado `objeto`, que actúa como nuestro objeto de ordenador. Un diccionario es como un conjunto de compartimentos, donde podemos guardar información de forma estructurada. Cada compartimento tiene un nombre especial llamado clave y contiene un valor asociado.

Por ejemplo, el diccionario `objeto` tiene una clave `"tipo"` que tiene el valor `"ordenador"`, lo que indica que estamos creando un objeto de tipo ordenador. Los objetos **siempre** tienen la clave tipo dentro del diccionario, el valor asociado a esta clave nos servirá para indicar que podemos hacer con ese objeto cuando definamos los métodos modificadores y observadores. Después de definir el valor de la clave tipo, asignamos los valores de los parámetros de la función a las claves correspondientes del diccionario.

Es decir, el parámetro `ram` se asigna a la clave `"ram"`, el parámetro `procesador` se asigna a la clave `"procesador"`, y así sucesivamente para las demás características del ordenador.

Después de llenar el diccionario con todas las características del ordenador, utilizamos la instrucción `return objeto` para devolver el objeto completo.

Ahora, cuando llamamos a la función `constructorOrdenadores` y le pasamos valores específicos para cada parámetro, obtenemos un objeto que representa un ordenador con esas características. Por ejemplo, si llamamos a la función de la siguiente manera:

```
miOrdenador = constructorOrdenadores(ram = 8,
                                     procesador = "Intel i5",
                                     tarjeta_grafica = "NVIDIA GTX 1050",
                                     almacenamiento = "1TB HDD",
                                     pantalla = "15.6 pulgadas",
                                     teclado = "QWERTY",
                                     raton = "óptico")
```

Nota: Los parámetros podrían estar puestos todos en la misma línea, pero por cuestiones de organización del código, los puse uno por línea.

La función devolverá un objeto que representa un ordenador con las siguientes características:

- RAM: 8GB
- Procesador: Intel i5
- Tarjeta gráfica: NVIDIA GTX 1050
- Almacenamiento: 1TB HDD
- Pantalla: 15.6 pulgadas
- Teclado: QWERTY
- Ratón: Óptico

Tal y como se ve en el print:

```
34 print(miOrdenador)
35

Ejecutando: Ordenadores.py
{'tipo': 'ordenador', 'ram': 8, 'procesador': 'Intel i5',
'tarjeta_grafica': 'NVIDIA GTX 1050', 'almacenamiento': '1TB HDD',
'pantalla': '15.6 pulgadas', 'teclado': 'QWERTY', 'raton': 'óptico'}
>>>
```

¿Fácil, verdad? 🐼

Lo que acabamos de ver será muy útil cuando tengamos que definir varios objetos con características similares en nuestro proyecto. La herramienta explicada arriba nos proporciona un código mucho más ordenado y limpio para las situaciones que vamos a enfrentar de aquí en adelante.

2. Métodos Observadores

Los **métodos observadores**, también conocidos como **métodos de acceso** o **getters**, son funciones que nos permiten obtener información o acceder a los valores de los atributos de un objeto. Estos métodos proporcionan una forma segura de obtener información sin permitir modificaciones directas en los atributos subyacentes. Por lo general, los métodos observadores tienen nombres descriptivos que indican la información que se está obteniendo, en nuestro caso, definiremos los métodos observadores con el nombre **"getAtributo"**, donde "Atributo" es el atributo del objeto que queremos observar.

Por ejemplo, supongamos que tenemos un objeto de tipo "ordenador" con un atributo llamado "ram". Un método observador para obtener el valor de "ram" podría ser algo como:

```
def getRAM(ordenador):  
    return ordenador["ram"]
```

Este método nos permitiría obtener el valor de "ram" de un objeto de ordenador sin acceder directamente al atributo. Sin embargo, hay un pequeño problema con ese método. Miremos el programa abajo para que entendamos que es lo que va mal con el método **getRAM**:

```
25  
26 miOrdenador = constructorOrdenadores(ram = 8,  
27                                     procesador = "Intel i5",  
28                                     tarjeta_grafica = "NVIDIA GTX 1050",  
29                                     almacenamiento = "1TB HDD",  
30                                     pantalla = "15.6 pulgadas",  
31                                     teclado = "QWERTY",  
32                                     raton = "óptico")  
33  
34 ordenadorFake = {"tipo": "virus", "objetivo": "teviarobá 😈"}  
35  
36  
37 print("mi ordenador:", miOrdenador)  
38 print("ordenador fake:", ordenadorFake)  
39  
40 def getRAM(ordenador):  
41     return ordenador["ram"]  
42  
43 ram_miOrdenador = getRAM(miOrdenador)  
44 print("RAM de mi ordenador:", ram_miOrdenador)  
45 ram_ordenadorFake = getRAM(ordenadorFake)  
46 print("RAM del ordenador fake:", ram_ordenadorFake)  
47
```

Si sois muy buenos alumnos y estáis siempre atentos a lo que aprendemos en las clases (sois así, ¿a que sí? 🙄) creo que ya sabéis que error hay en el código. Si no, echadle un vistazo con calma al código e intentad encontrar que es lo que falla. Venga, os doy 10 segundos. 🕒

Y... ¡tiempo!

Si no habéis encontrado el error, os enseño qué pasa cuando ejecutamos el código enseñado arriba:


```
mi ordenador: {'tipo': 'ordenador', 'ram': 8, 'procesador': 'Intel i5',
'tarjeta_grafica': 'NVIDIA GTX 1050', 'almacenamiento': '1TB HDD',
'pantalla': '15.6 pulgadas', 'teclado': 'QWERTY', 'raton': 'óptico'}

ordenador fake: {'tipo': 'virus', 'objetivo': 'teviarobá 🤖'}

RAM de mi ordenador: 8

Traceback (most recent call last):
  File "d:\clovi\aulas\python-2-hilcode-alumnos-22-23\9 - pseudo
programación orientada a objetos\ordenadores.py", line 45, in <module>
    ram_ordenadorFake = getRAM(ordenadorFake)
  File "d:\clovi\aulas\python-2-hilcode-alumnos-22-23\9 - pseudo
programación orientada a objetos\ordenadores.py", line 41, in getRAM
    return ordenador["ram"]
KeyError: 'ram'
>>>
```

Como se puede ver, al intentar obtener la RAM de `ordenadorFake`, nos salta un error. En el método `getRAM`, la única línea de código que hay es `return ordenador["ram"]`. Por esto, si le pasamos a este método un objeto que no sea de tipo ordenador, ese objeto no tendrá la clave `"ram"`, que es lo que pasa con `ordenadorFake` que tiene solo la clave tipo y la clave objeto. Luego, al intentar acceder al atributo `"ram"` de `ordenadorFake`,

¡boom!  nos sale un error de tipo `KeyError` indicando que hubo un fallo al acceder a la clave `"ram"` del objeto pasado como parámetro al método `getRAM`, ya que esta clave no existe en el objeto. Un error con el que estamos medianamente familiarizados.

¿Y cómo lo arreglamos?

¡Es muy sencillo! Es algo que hemos hecho ya en otros proyectos. Mirémoslo:

```
def getRAM(ordenador):
    ram = None
    if type(ordenador) != dict:
        print("La variable no es un objeto. Tipo informado: ",type(ordenador))
    elif "tipo" not in ordenador:
        print("El objeto no tiene tipo definido.")
    elif ordenador["tipo"] != "ordenador":
        print("Objeto de tipo incompatible: ", ordenador["tipo"])
    else:
        ram = ordenador["ram"]
    return ram
```

¡Ahora sí! Este sería el método `getRAM` hecho de manera correcta, veamos cómo funciona:

1. Al llamar a la función `getRAM` y pasar un objeto de tipo ordenador como argumento, se crea una variable llamada `ram` y se le asigna inicialmente el valor `None`. Esta

variable será utilizada para almacenar la cantidad de memoria RAM del objeto de ordenador.

```
ram = None
```

2. A continuación, se realizan varias comprobaciones para asegurarnos de que el objeto pasado como parámetro cumpla con las condiciones esperadas de un objeto de tipo ordenador.
3. En la primera comprobación, se verifica si el tipo del objeto `ordenador` es un diccionario. Esto se hace utilizando el comando `type(ordenador)`. Si el *tipo de variable* informada como parámetro no es un diccionario, significa que el objeto no cumple con la estructura requerida y no puede ser un *objeto de tipo ordenador*. En este caso, se muestra un mensaje de error indicando que la variable no es un objeto válido, y se imprime el tipo de dato informado.

```
if type(ordenador) != dict:  
    print("La variable no es un objeto. Tipo informado: ", type(ordenador))
```

4. Si el objeto `ordenador` es un diccionario, se procede a la siguiente comprobación. Aquí se verifica si el diccionario `ordenador` tiene una clave llamada "tipo". Esta clave es necesaria para identificar el *tipo del objeto*. Si la clave "tipo" no está presente en el diccionario, significa que el objeto no tiene un tipo definido. En este caso, se muestra un mensaje de error indicando que el objeto no tiene un tipo definido.

```
elif "tipo" not in ordenador:  
    print("El objeto no tiene tipo definido.")
```

5. Si el diccionario `ordenador` tiene la clave "tipo", se realiza otra comprobación para verificar si el valor asociado a esa clave es igual a "ordenador". Esto asegura que el objeto sea de tipo ordenador y no otro tipo incompatible. Si el valor no coincide con "ordenador", se muestra un mensaje de error indicando que el objeto es de un tipo incompatible.

```
elif ordenador["tipo"] != "ordenador":  
    print("Objeto de tipo incompatible: ", ordenador["tipo"])
```

6. Si todas las comprobaciones anteriores se pasan correctamente y se determina que el objeto `ordenador` es válido y de tipo "ordenador", se procede a asignar el valor del atributo "ram" del diccionario `ordenador` a la variable `ram`. Esto se hace accediendo al diccionario mediante la clave "ram", es decir, `ordenador["ram"]`.

```
else:  
    ram = ordenador["ram"]
```

7. Finalmente, se devuelve el valor de la variable `ram`. Si todas las comprobaciones se pasan correctamente, el valor de `ram` será la cantidad de memoria RAM del objeto de ordenador. En caso contrario, si se produce algún error en las comprobaciones, el valor de `ram` permanecerá como `None`.

```
return ram
```


Ejecutando el código anterior con el nuevo método `getRAM` tenemos la siguiente salida:

```
mi ordenador: {'tipo': 'ordenador', 'ram': 8, 'procesador': 'Intel i5',
'tarjeta_grafica': 'NVIDIA GTX 1050', 'almacenamiento': '1TB HDD',
'pantalla': '15.6 pulgadas', 'teclado': 'QWERTY', 'raton': 'óptico'}

ordenador fake: {'tipo': 'virus', 'objetivo': 'teviarobá 🇪🇸'}

RAM de mi ordenador: 8

Objeto de tipo incompatible: virus
RAM del ordenador fake: None

>>> |
```

Cómo es posible observar, ya no saltan errores y el código puede seguir su ejecución como si nada hubiera pasado (y sin ser robado por un virus).

Tras explicar ese método, es importante aclarar dos cosas: **tipo de variable** y **tipo de objeto**.

Cuando nos referimos al **tipo de una variable**, nos referimos a las variables estándar de Python, que pueden ser de tipo `int`, `float`, `str`, `dict`, `list`, etc. Ya, hablando de **tipo de un objeto**, no nos referimos si es `int`, `float` o los demás tipos de variables, nos referimos al tipo que tiene asociado. Por ejemplo, los **objetos** de tipo ordenador son representados por una **variable** de tipo diccionario (`dict`), dentro de esta variable existe una clave “tipo” que contiene el valor “ordenador”, esta clave es la que define el tipo de objeto.

Los métodos observadores seguirán siempre el mismo modelo de código y tiene dos características que se cumplen **siempre**:

- 👉 reciben como parámetro un **objeto**;

- 👉 devuelven en el return un valor que puede ser `None` o el valor de uno de los atributos del objeto pasado por parámetro.

3. Métodos Modificadores

Ahora que ya vimos los métodos observadores, vamos a ver cómo funcionan los métodos modificadores.

Como dicho antes, los métodos modificadores, también conocidos como métodos de mutación o métodos setters, son funciones especiales en la programación orientada a objetos que nos permiten cambiar el valor de los atributos de un objeto. Estos métodos también son parte integral de la encapsulación, que es uno de los principios fundamentales de la programación orientada a objetos.

Un método modificador se utiliza para actualizar o modificar el estado interno de un objeto al cambiar el valor de uno o más de sus atributos. Estos métodos son definidos dentro de la clase a la que pertenece el objeto y tienen acceso directo a los atributos del objeto.

Así como los métodos observadores, los métodos modificadores tienen nombres descriptivos que indican la información que se está cambiando, en nuestro caso, definiremos los métodos modificadores con el nombre “`setAtributo`”, donde “Atributo” es el atributo del objeto que queremos modificar.

Veamos cómo sería entonces la base para el método que modifica el atributo ram de un objeto de tipo ordenador:

```
def setRAM(cantidad, ordenador):  
    ordenador["ram"] = cantidad  
    return ordenador
```

Como se puede observar en el código arriba, lo que hace el método modificador es simplemente cambiar el valor de un atributo por un valor nuevo que es pasado como parámetro al método. El uso de los modificadores es similar al uso de los observadores:

```
53 def setRAM(cantidad, ordenador):  
54     ordenador["ram"] = cantidad  
55     return ordenador  
56  
57 miOrdenador = setRAM(cantidad=16, ordenador=miOrdenador)  
58 print("miOrdenador después del setRAM:", miOrdenador)  
59 ordenadorFake = setRAM(cantidad=8, ordenador=ordenadorFake)  
60 print("ordenadorFake después del setRAM", ordenadorFake)
```

Vamos a fijarnos ahora en un detalle importante: en los métodos get (los observadores) el return de la función lo guardamos en una variable a parte, recordémoslo:

```
ram_miOrdenador = getRAM(miOrdenador)
print("RAM de mi ordenador:", ram_miOrdenador)
ram_ordenadorFake = getRAM(ordenadorFake)
print("RAM del ordenador fake:", ram_ordenadorFake)
```

Podemos ver que cada valor devuelto por los get, los guardamos en una variable a parte.

Ya para los métodos set (los modificadores) ¡el valor devuelto por el método es todo el objeto ya cambiado! Por eso, guardamos el valor en el mismo objeto que pasamos como parámetro al llamar el método:

```
57 miOrdenador = setRAM(cantidad=16, ordenador=miOrdenador)
58 print("miOrdenador después del setRAM:", miOrdenador)
59 ordenadorFake = setRAM(cantidad=8, ordenador=ordenadorFake)
60 print("ordenadorFake después del setRAM", ordenadorFake)
```

Fijemos también que los métodos set reciben dos parámetros: el nuevo valor para el atributo que se está cambiando y el objeto en el cuál queremos cambiar ese atributo. En el ejemplo vemos que en la línea 57, `setRAM` recibe como parámetros la nueva cantidad para la ram (que en este caso vale 16) y también recibe como parámetro qué objeto tendrá esa cantidad modificada (que es el objeto `miOrdenador`) y siempre, **SIEMPRE**, al llamar un método modificador, guardamos el valor que devuelve en el mismo objeto que ha sido pasado como parámetro, como se ve en la línea 57 que llamamos el método para cambiar la ram de `miOrdenador` y en la línea 59, que llamamos el método para cambiar la ram de `ordenadorFake`.

Dicho todo esto, lamento informaros, pero el código proporcionado para el método `setRAM` también está incompleto y creo que vosotros ya tenéis una idea del porqué. Miremos que pasa al ejecutar el código:

```
mi ordenador: {'tipo': 'ordenador', 'ram': 8, 'procesador': 'Intel i5',
'tarjeta_grafica': 'NVIDIA GTX 1050', 'almacenamiento': '1TB HDD',
'pantalla': '15.6 pulgadas', 'teclado': 'QWERTY', 'raton': 'óptico'}

ordenador fake: {'tipo': 'virus', 'objetivo': 'teviarobá 🤖'}

miOrdenador después del setRAM: {'tipo': 'ordenador', 'ram': 16,
'procesador': 'Intel i5', 'tarjeta_grafica': 'NVIDIA GTX 1050',
'almacenamiento': '1TB HDD', 'pantalla': '15.6 pulgadas', 'teclado':
'QWERTY', 'raton': 'óptico'}

ordenadorFake después del setRAM {'tipo': 'virus', 'objetivo':
'teviarobá 🤖', 'ram': 8}
```

¿Véis algo raro?

¡Sí! Después de la ejecución del método, `ordenadorFake` pasa a tener 8 de RAM 🤖, así, *by the face*. Obviamente, ¡esto no debería pasar!

Los métodos modificadores **NO** pueden crear nuevos atributos a los objetos, son responsables **APENAS** por cambiar el valor de los atributos ya existentes. Lo que pasa arriba es que el método recibe dos parámetros: la nueva cantidad de ram y el objeto, y lo que hace a continuación es asignar del tirón el nuevo valor de ram al atributo "ram" del objeto indicado. Como estamos trabajando con diccionarios, al hacer `ordenador["ram"] = cantidad`, si la clave "ram" no existe, el programa la crea al instante. Por eso, tras ejecutar el método `setRAM` para `ordenadorFake`, `ordenadorFake` sale con un atributo bonus, como buen virus que es. Pero no os preocupéis, lo hemos pillado antes que os lo robara vuestra cuenta de Fortnite con todas vuestras skins caras de Naruto y Spiderman 🕸️.

Bueeeeno, ¿cómo lo arreglamos entonces?

Pomusimple, al igual que hemos hecho con `getRAM`:

```
53 def setRAM(cantidad, ordenador):
54     if type(cantidad) != int and type(cantidad) != float:
55         print("La cantidad no es de tipo numérico. Tipo informado: ",type(cantidad))
56     elif cantidad < 0:
57         print("La cantidad tiene que ser un valor positivo y no nulo.")
58     elif type(ordenador) != dict:
59         print("La variable no es un objeto. Tipo informado: ",type(ordenador))
60     elif "tipo" not in ordenador:
61         print("El objeto no tiene dipo definido.")
62     elif ordenador["tipo"] != "ordenador":
63         print("Objeto de tipo incompatible: ", ordenador["tipo"])
64     else:
65         ordenador["ram"] = cantidad
66     return ordenador
```

Vamos a ver abajo con detalles que es lo que pasa en ese método:

1. El método `setRAM` recibe dos parámetros: `cantidad` y `ordenador`. `cantidad` representa la nueva cantidad de memoria RAM que se desea asignar, y `ordenador` es el objeto de tipo ordenador al que se le quiere cambiar la cantidad de RAM.
2. La primera comprobación que se realiza es verificar si `cantidad` no es un número entero ni un número de punto flotante (float). Si `cantidad` no es un tipo numérico válido, se muestra un mensaje de error indicando que la cantidad no es de tipo numérico, y se imprime el tipo de dato informado.

```
if type(cantidad) != int and type(cantidad) != float:
    print("La cantidad no es de tipo numérico. Tipo informado: ",type(cantidad))
```

3. A continuación, se verifica si la `cantidad` es un valor negativo. Obviamente un ordenador no puede tener -12GB de RAM, por ejemplo. Por eso se espera que los valores sean positivos y no nulos. Si la `cantidad` es menor que cero, se muestra un mensaje de error indicando que la cantidad debe ser un valor positivo y no nulo.

```
elif cantidad < 0:
    print("La cantidad tiene que ser un valor positivo y no nulo.")
```

4. Luego, se realiza una serie de comprobaciones sobre el parámetro `ordenador`. Se verifica si `ordenador` no es un diccionario. Si no es un diccionario, se muestra un mensaje de error indicando que la variable no es un objeto válido, y se imprime el tipo de dato informado.

```
elif type(ordenador) != dict:  
    print("La variable no es un objeto. Tipo informado: ",type(ordenador))
```

5. A continuación, se verifica si el diccionario `ordenador` tiene una clave llamada "tipo". Esta clave es necesaria para identificar el tipo del objeto. Si la clave "tipo" no está presente en el diccionario, se muestra un mensaje de error indicando que el objeto no tiene un tipo definido.

```
elif "tipo" not in ordenador:  
    print("El objeto no tiene tipo definido.")
```

6. Después, se verifica si el valor asociado a la clave "tipo" en el diccionario `ordenador` es igual a "ordenador". Esto asegura que el objeto sea de tipo ordenador y no otro tipo incompatible. Si el valor no coincide con "ordenador", se muestra un mensaje de error indicando que el objeto es de un tipo incompatible.

```
elif ordenador["tipo"] != "ordenador":  
    print("Objeto de tipo incompatible: ", ordenador["tipo"])
```

7. Si todas las comprobaciones anteriores se pasan correctamente y se determina que el objeto `ordenador` es válido y de tipo "ordenador", se procede a asignar la nueva cantidad de memoria RAM al atributo "ram" del diccionario `ordenador`. Esto se hace mediante la línea `ordenador["ram"] = cantidad`.

```
else:  
    ordenador["ram"] = cantidad
```

8. Por último, se devuelve el objeto `ordenador`, que ahora ha sido modificado con la nueva cantidad de RAM. Si alguna de las comprobaciones falla, el objeto se devuelve sin modificaciones así nos aseguramos de no crear nuevas claves en los objetos.

```
return ordenador
```

Una vez implementado el método `setRAM` como enseñado arriba, la ejecución del código queda así:

```
mi ordenador: {'tipo': 'ordenador', 'ram': 8, 'procesador': 'Intel i5',  
'tarjeta_grafica': 'NVIDIA GTX 1050', 'almacenamiento': '1TB HDD', 'pantalla':  
'15.6 pulgadas', 'teclado': 'QWERTY', 'raton': 'óptico'}  
  
ordenador fake: {'tipo': 'virus', 'objetivo': 'teviarobá 🤖'}  
  
miOrdenador después del setRAM: {'tipo': 'ordenador', 'ram': 16, 'procesador':  
'Intel i5', 'tarjeta_grafica': 'NVIDIA GTX 1050', 'almacenamiento': '1TB HDD',  
'pantalla': '15.6 pulgadas', 'teclado': 'QWERTY', 'raton': 'óptico'}  
  
Objeto de tipo incompatible: virus  
ordenadorFake después del setRAM {'tipo': 'virus', 'objetivo': 'teviarobá 🤖'}
```

Se ve que no hubo cambios en `ordenadorFake`, ya que no cumplía con los estándares para que el método `setRAM` cambiase alguno de sus atributos.

En resumen, el método `setRAM` nos permite asignar una nueva cantidad de memoria RAM a un objeto de tipo ordenador, realizando previamente varias comprobaciones para asegurarnos de que los parámetros sean válidos y cumplan con las condiciones esperadas. Esto nos ayuda a controlar los cambios en los atributos del objeto y garantizar la integridad de los datos.

Normalmente, los objetos tienen para cada atributo un método observador y a depender del atributo que estamos manipulando, tendrá o no métodos modificadores. Para nuestro ejemplo, haremos solo el `getRAM` y el `setRAM`, pero si haremos el proyecto completo, existiría también los métodos `getProcesador`, `setProcesador`, `getAlmacenamiento`, `setAlmacenamiento`, etc. Además, para algunos atributos como la RAM y almacenamiento, podríamos hacer otros métodos modificadores diferentes de los sets, como el `aumentaRAM`, `disminuyeRAM`, `aumentaAlmacenamiento` y `disminuyeAlmacenamiento`. Normalmente para atributos numéricos es común haber métodos responsables para incrementar y decrementar sus valores como los últimos mencionados.

Tras todo eso, aquí terminamos este ejemplo. ¿Vamos al siguiente ejemplo?

Ejemplo 2: constructorCine

Ahora vamos a explorar un nuevo ejemplo de aplicación de Pseudo Orientación a Objetos. En este caso, crearemos un programa que se encargará de construir salas de cine con diferentes propiedades.

Al igual que en el ejemplo anterior, lo primero que debemos hacer es definir el método constructor para los objetos. Pero antes de hacer eso, necesitamos identificar los atributos que tendrán los objetos creados por ese constructor. En este caso, cada sala de cine tendrá cuatro atributos: el número de filas, el número de columnas, los diferentes asientos y un nombre.

Sin embargo, a diferencia del ejemplo anterior, no todos los atributos serán pasados como parámetros al método constructor. En este caso, uno de los atributos se generará a partir de la interacción de otros dos. ¿Adivinas cuál es?

¡Exactamente! Los asientos. Los asientos serán generados a partir del número de filas y el número de columnas de cada sala. Por ejemplo, si una sala tiene 10 filas y 8 columnas, habrá un total de 80 asientos ($10 \times 8 = 80$). Si son 9 filas y 9 columnas, entonces habrá un total de 81 asientos ($9 \times 9 = 81$), y así sucesivamente.

Conociendo este detalle, ahora podemos partir para la definición del encabezado método constructor, que quedaría de la siguiente forma:

```
def constructorSalas(n_filas, n_columnas, nombre):
```

Ahora que tenemos una idea clara de lo que queremos lograr, podemos continuar con la definición del método constructor. A diferencia del ejemplo anterior, dentro del método constructor de la sala de cine tendremos que escribir algunas líneas de código para generar los asientos de la sala antes de definir y devolver el objeto.

La forma en que generamos los asientos sigue un estándar particular: las filas se codifican con letras de la A a la Z, y las columnas se representan con números. Por lo tanto, si una sala tiene 8 filas y 10 columnas, las filas se etiquetan desde la A hasta la H, y las columnas irán del 1 al 10. Cada asiento se identificará mediante la combinación de una letra y un número (A-1, A-2, C-4, D-7, etc.). Nuestro objetivo es escribir un código que defina todos los asientos de una sala en función del número de filas y columnas, y luego inicializar esos asientos como vacíos (utilizaremos el valor False para representar un asiento vacío y True para un asiento ocupado).

En cada sala de cine, las butacas se almacenarán en un diccionario. Cada clave del diccionario representará la ubicación de una butaca (por ejemplo, B-5, F-9, A-3), y el valor asociado a cada clave indicará si la butaca está ocupada o vacía. Inicialmente, todas las butacas estarán vacías, por lo que los valores serán False.

Veamos un ejemplo para comprender mejor. Supongamos que quiero construir una sala de cine con 5 filas y 5 columnas. En este caso, el constructor generaría las butacas de la siguiente manera:

```
{ 'A-1': False, 'A-2': False, 'A-3': False, 'A-4': False, 'A-5': False,
  'B-1': False, 'B-2': False, 'B-3': False, 'B-4': False, 'B-5': False,
  'C-1': False, 'C-2': False, 'C-3': False, 'C-4': False, 'C-5': False,
  'D-1': False, 'D-2': False, 'D-3': False, 'D-4': False, 'D-5': False,
  'E-1': False, 'E-2': False, 'E-3': False, 'E-4': False, 'E-5': False }
```

Si digo que quiero una sala con 8 filas y 10 columnas, el resultado sería el siguiente: Cada clave del diccionario representa la dirección de una butaca y su valor es False, indicando que la butaca está vacía.

Si en cambio deseamos construir una sala de cine con 8 filas y 10 columnas, el resultado sería el siguiente:

```
{ 'A-1': False, 'A-2': False, 'A-3': False, 'A-4': False, 'A-5': False,
  'A-6': False, 'A-7': False, 'A-8': False, 'A-9': False, 'A-10': False,
  'B-1': False, 'B-2': False, 'B-3': False, 'B-4': False, 'B-5': False,
  'B-6': False, 'B-7': False, 'B-8': False, 'B-9': False, 'B-10': False,
  'C-1': False, 'C-2': False, 'C-3': False, 'C-4': False, 'C-5': False,
  'C-6': False, 'C-7': False, 'C-8': False, 'C-9': False, 'C-10': False,
  'D-1': False, 'D-2': False, 'D-3': False, 'D-4': False, 'D-5': False,
  'D-6': False, 'D-7': False, 'D-8': False, 'D-9': False, 'D-10': False,
  'E-1': False, 'E-2': False, 'E-3': False, 'E-4': False, 'E-5': False,
  'E-6': False, 'E-7': False, 'E-8': False, 'E-9': False, 'E-10': False,
  'F-1': False, 'F-2': False, 'F-3': False, 'F-4': False, 'F-5': False,
  'F-6': False, 'F-7': False, 'F-8': False, 'F-9': False, 'F-10': False,
  'G-1': False, 'G-2': False, 'G-3': False, 'G-4': False, 'G-5': False,
  'G-6': False, 'G-7': False, 'G-8': False, 'G-9': False, 'G-10': False,
  'H-1': False, 'H-2': False, 'H-3': False, 'H-4': False, 'H-5': False,
  'H-6': False, 'H-7': False, 'H-8': False, 'H-9': False, 'H-10': False }
```

Te invito a intentar escribir el código por ti mismo para generar las butacas de forma dinámica, siguiendo el modelo que hemos explicado previamente. Puede parecer un poco difícil al principio, pero recuerda que la práctica es la clave para mejorar tus habilidades de programación. ¡Vamos, atrévete a enfrentar este desafío!

Como decía, abajo está el código del método constructor completo, incluyendo la parte responsable por generar las butacas de manera dinámica:

```
def constructorSalas(n_filas, n_columnas, nombre):
    letras = ["A", "B", "C", "D", "E", "F", "G", "H", "I", "J", "K", "L", "M",
              "N", "O", "P", "Q", "R", "S", "T", "U", "V", "W", "X", "Y", "Z"]
    filas = list() # es lo mismo que filas = [] (crea una lista vacía)
    asientos = dict() # es lo mismo que sala = {} (crea un diccionario vacío)

    # en la línea 11, todos los elementos de la lista 'letras'
    # a partir de la posición 0 hasta la posición 'n_filas' se guardan en 'filas'
    filas = letras[0:n_filas] # [elemento_inicial : elemento_final]

    for i in range(len(filas)):
        for j in range(1, n_columnas+1):
            asientos[filas[i] + "-" + str(j)] = False

    objeto = { "tipo": "sala",
               "nombre": nombre,
               "filas": n_filas,
               "columnas": n_columnas,
               "asientos": asientos }
    return objeto
```

¡Eh, no te asustes! El código no es tan aterrador como parece, es en verdad muy simpático. Miremos cómo funciona y ya te darás cuenta de lo cuqui que es:

1. Se define la función `constructorSalas` que toma tres parámetros: `n_filas` (número de filas de la sala), `n_columnas` (número de columnas de la sala) y `nombre` (nombre de la sala).

```
def constructorSalas(n_filas, n_columnas, nombre):
```

2. Se crea una lista llamada `letras` que contiene todas las letras del alfabeto en mayúsculas.

```
letras = ["A", "B", "C", "D", "E", "F", "G", "H", "I", "J", "K", "L", "M",
          "N", "O", "P", "Q", "R", "S", "T", "U", "V", "W", "X", "Y", "Z"]
```

3. Se inicializa una lista vacía llamada `filas` que se utilizará para almacenar las filas de la sala.

```
filas = list() # es lo mismo que filas = [] (crea una lista vacía)
```

4. Se crea un diccionario vacío llamado `asientos` que se utilizará para almacenar la información de ocupación de cada asiento de la sala.

```
asientos = dict() # es lo mismo que sala = {} (crea un diccionario vacío)
```

5. Se asigna a la lista `filas` los primeros `n_filas` elementos de la lista `letras`. Esto crea una lista con las letras necesarias para representar las filas de la sala. El código

usado para tal hecho es `letras[0:n_filas]`, este código toma un subconjunto de elementos de la lista `letras`, desde el elemento en la posición `0` hasta el elemento en la posición `n_filas-1`. La posición `0` corresponde a la primera letra de la lista `letras`.

Por ejemplo, si `n_filas` es igual a 3, la línea de código se vería así: `filas = letras[0:3]`. Esto significa que queremos tomar los primeros 3 elementos de la lista `letras`. Suponiendo que la lista `letras` es `['A', 'B', 'C', 'D', 'E', . . ., Y, Z]`, el resultado sería una nueva lista llamada `filas` que contendría `['A', 'B', 'C']` (los tres primeros elementos contenidos en “letras”).

```
# en la línea 11, todos los elementos de la lista 'letras'
# a partir de la posición 0 hasta la posición 'n_filas' se guardan en 'filas'
filas = letras[0:n_filas] # [elemento_inicial : elemento_final]
```

6. Se inicia un bucle `for` que itera sobre cada elemento de la lista `filas`. Imagina que cada elemento de esa lista es una fila de butacas en nuestra sala. ¡Tenemos muchas filas para acomodar a nuestros espectadores!

```
for i in range(len(filas)):
```

7. Dentro del primer bucle `for`, encontramos el segundo bucle `for`. Este bucle se encarga de iterar sobre el número de columnas que queremos en nuestra sala. Cada número representa una butaca en una fila determinada. ¡Aquí es donde los espectadores tomarán asiento y disfrutarán de la película!

```
for j in range(1,n_columnas+1):
```

8. En cada iteración de los bucles `for`, creamos una clave en nuestro diccionario de `asientos`. La clave está formada por la combinación de una letra de la lista de filas (representado por `filas[i]`) un guión (“-”) y un número de columna convertido a cadena (`str(j)`). Esto nos permite identificar cada butaca de manera única. Por ejemplo, podríamos tener butacas como “A-1”, “B-2”, “C-3” y así sucesivamente.

El valor asociado a cada clave en el diccionario de asientos es inicialmente establecido como `False`. Esto significa que las butacas están vacías y esperando a que los espectadores ocupen sus lugares.

```
for i in range(len(filas)):
    for j in range(1,n_columnas+1):
        asientos[filas[i] + "-" + str(j)] = False
```

9. Después de que los bucles se completan, se crea un diccionario llamado `objeto` que representa la sala de cine. Este diccionario tiene cuatro pares clave-valor: "tipo" con el valor "sala", "nombre" con el valor del parámetro `nombre`, "filas" con el valor de `n_filas`, "columnas" con el valor de `n_columnas` y "asientos" con el diccionario `asientos`, que ha sido rellenado con los asientos dentro de los bucles `for`.

```
objeto = { "tipo": "sala",  
           "nombre": nombre,  
           "filas": n_filas,  
           "columnas": n_columnas,  
           "asientos": asientos}
```

10. Finalmente, se devuelve el diccionario `objeto` como resultado de la función `constructorSalas`, nótese que el método constructor devuelve siempre el objeto creado, así como en el ejemplo anterior.

```
return objeto
```

En resumen, el código crea un objeto de sala de cine representado por un diccionario. La sala tiene un número específico de filas y columnas, y cada asiento se representa con una clave única en el diccionario `asientos`, inicializado como `False` para indicar que están vacíos.

Ahora tenemos que hacer los métodos observadores y modificadores para los objetos de la clase "sala" (objetos hechos por el método `constructorSalas`).

Los objetos de tipo sala tienen en total 5 atributos(estamos considerando el "tipo" como atributo), de los cuales, todos tendrán métodos observadores, pero solo 2 de los 5 atributos tendrán métodos modificadores, ya que hay atributos que deberían ser inmutables.

Los dos atributos que tendrán métodos modificadores son `asientos`, ya que queremos que los asientos muden de vacío a ocupado, y de ocupado a vacío, y también el `nombre`, ya que una sala puede cambiar de "sala 1" a "sala 5", por ejemplo. Los otros 3 (`tipo`, `n_filas` y `n_columnas`) no tienen métodos modificadores porque el tipo siempre será el mismo (sala), es decir, una sala no deja de ser "sala" para pasar a ser "ordenador". Los otros dos (`n_filas` y `n_columnas`) no tienen modificadores porque las salas, una vez creadas, tendrán siempre el mismo tamaño.

Vamos a definir entonces el primer método observador, que sería el `getNombre`. Como el propio nombre indica, este método sería responsable por devolver el valor del nombre de una sala.

Los métodos `get` y `set` son muy parecidos entre sí, aunque estemos lidiando con objetos de diferentes clases. Vamos a recordar como habíamos hecho el `getRAM` para los objetos de la clase “ordenador”:

```
def getRAM(ordenador):
    ram = None
    if type(ordenador) != dict:
        print("La variable no es un objeto. Tipo informado: ", type(ordenador))
    elif "tipo" not in ordenador:
        print("El objeto no tiene tipo definido.")
    elif ordenador["tipo"] != "ordenador":
        print("Objeto de tipo incompatible: ", ordenador["tipo"])
    else:
        ram = ordenador["ram"]
    return ram
```

Y ahora, éste sería el `getNombre`, para objetos de la clase “sala”:

```
def getNombre(sala):
    nombre = None
    if type(sala) != dict:
        print("La variable no es un objeto. Tipo informado:", type(sala))
    elif "tipo" not in sala:
        print("El objeto no tiene tipo definido.")
    elif sala["tipo"] != "sala":
        print("Objeto de tipo incompatible:", sala["tipo"])
    else:
        nombre = sala["nombre"]
    return nombre
```

¿Ves la semejanza? Aunque un poquito diferentes, son el mismo perro con diferente collar, ¿a que sí?

En este caso, voy a dispensar la explicación completa del método ya que sería exactamente la misma explicación del `getRAM`.

Sobre el `setNombre`, pues sería más de lo mismo. Recordemos cómo estaba hecho para los objetos de tipo ordenador en `setRAM`:

```
def setRAM(cantidad, ordenador):
    if type(cantidad) != int and type(cantidad) != float:
        print("La cantidad no es de tipo numérico. Tipo informado: ",type(cantidad))
    elif cantidad < 0:
        print("La cantidad tiene que ser un valor positivo y no nulo.")
    elif type(ordenador) != dict:
        print("La variable no es un objeto. Tipo informado: ",type(ordenador))
    elif "tipo" not in ordenador:
        print("El objeto no tiene tipo definido.")
    elif ordenador["tipo"] != "ordenador":
        print("Objeto de tipo incompatible: ", ordenador["tipo"])
    else:
        ordenador["ram"] = cantidad
    return ordenador
```

Y ahora el `setNombre`:

```
def setNombre(nombre, sala):
    if type(nombre) != str:
        print("El nombre debe ser una cadena de texto.")
    elif type(sala) != dict:
        print("La variable no es un objeto. Tipo informado:", type(sala))
    elif "tipo" not in sala:
        print("El objeto no tiene tipo definido.")
    elif sala["tipo"] != "sala":
        print("Objeto de tipo incompatible:", sala["tipo"])
    else:
        sala["nombre"] = nombre
    return sala
```

Son prácticamente gemelos, el cambio más significativo que hay es en el primer `if` en `setNombre` que compara el tipo de la variable `nombre` porque tiene que ser una `string`, y en `setRAM` la comparación es con la variable `cantidad` que tiene que ser un valor numérico `int` o `float`. Aparte de eso, ¡los métodos son prácticamente iguales!

Vamos a ver más un ejemplo. Miremos cómo quedaría el método `getEstadoAsiento` para obtener el valor de un asiento de la sala y saber si está vacío u ocupado:

```
def getEstadoAsiento(asiento, sala):
    estado = None
    if type(asiento) != str:
        print("El asiento debe ser especificado en formato de cadena de texto.")
    elif type(sala) != dict:
        print("La variable no es un objeto. Tipo informado:", type(sala))
    elif "tipo" not in sala:
        print("El objeto no tiene tipo definido.")
    elif sala["tipo"] != "sala":
        print("Objeto de tipo incompatible:", sala["tipo"])
    elif asiento not in sala["asientos"]:
        print("El asiento",asiento,"no existe en", sala["nombre"])
    else:
        estado = sala["asientos"][asiento]

    return estado
```

Muy parecido con los anteriores también, ¿verdad? El único cambio está en el último `elif` que es responsable por verificar si el asiento pasado como parámetro existe como clave en el diccionario `asientos` contenido dentro del objeto `sala` que también es pasado como parámetro al método. De resto, el método es prácticamente igual a los anteriores.

Ahora abajo dejo un ejemplo de código usando los métodos descritos arriba para los objetos de la clase `sala`:

```
# construyendo las salas
sala_1 = constructorSalas(n_filas=10, n_columnas=10, nombre="Sala 1")
salaFake = {"tipo": "no soy una sala jijiji", "n_filas": "-7 🤪", "n_columnas": "no tengo eso"}

print("Sala 1 recién construida: ", sala_1)
print("\n")
print("Sala Fake recién construida: ", salaFake)

print("\n\nUSANDO EL getNombre EN SALA 1")
nombre_sala_1 = getNombre(sala_1)
print("Nombre de la Sala 1: ", nombre_sala_1)
print("\n\nUSANDO EL getNombre EN SALA FAKE")
nombre_salaFake = getNombre(salaFake)
print("Nombre de la Sala Fake: ", nombre_salaFake)

print("\n\nUSANDO EL setNombre EN SALA 1")
sala_1 = setNombre(sala = sala_1, nombre="Sala Uno")
print("USANDO EL setNombre EN SALA FAKE")
salaFake = setNombre(sala = salaFake, nombre="Soy super fake")

print("\n")
print("Sala 1 después del setNombre: ", sala_1)
print("\n")
print("Sala Fake después del setNombre: ", salaFake)

estado_asientoG8 = getEstadoAsiento(sala = sala_1, asiento = "G-8")
print("Estado del asiento G-8 en Sala Uno: ", estado_asientoG8)
estado_asientoM15 = getEstadoAsiento(sala = sala_1, asiento = "M-15")
print("Estado del asiento M-15 en Sala Uno: ", estado_asientoM15)
estado_asientoA1 = getEstadoAsiento(sala = salaFake, asiento = "A-1")
print("Estado del asiento A-1 en Sala Fake: ", estado_asientoA1)
```

Y esta sería la salida generada por el código descrito arriba:

```
Sala 1 recién construida: {'tipo': 'sala', 'nombre': 'Sala 1', 'filas': 10,
'columnas': 10, 'asientos': {'A-1': False, 'A-2': False, 'A-3': False, 'A-4': False,
'A-5': False, 'A-6': False, 'A-7': False, 'A-8': False, 'A-9': False, 'A-10': False,
'B-1': False, 'B-2': False, 'B-3': False, 'B-4': False, 'B-5': False, 'B-6': False,
'B-7': False, 'B-8': False, 'B-9': False, 'B-10': False, 'C-1': False, 'C-2': False,
'C-3': False, 'C-4': False, 'C-5': False, 'C-6': False, 'C-7': False, 'C-8': False,
'C-9': False, 'C-10': False, 'D-1': False, 'D-2': False, 'D-3': False, 'D-4': False,
'D-5': False, 'D-6': False, 'D-7': False, 'D-8': False, 'D-9': False, 'D-10': False,
'E-1': False, 'E-2': False, 'E-3': False, 'E-4': False, 'E-5': False, 'E-6': False,
'E-7': False, 'E-8': False, 'E-9': False, 'E-10': False, 'F-1': False, 'F-2': False,
'F-3': False, 'F-4': False, 'F-5': False, 'F-6': False, 'F-7': False, 'F-8': False,
'F-9': False, 'F-10': False, 'G-1': False, 'G-2': False, 'G-3': False, 'G-4': False,
'G-5': False, 'G-6': False, 'G-7': False, 'G-8': False, 'G-9': False, 'G-10': False,
'H-1': False, 'H-2': False, 'H-3': False, 'H-4': False, 'H-5': False, 'H-6': False,
'H-7': False, 'H-8': False, 'H-9': False, 'H-10': False, 'I-1': False, 'I-2': False,
'I-3': False, 'I-4': False, 'I-5': False, 'I-6': False, 'I-7': False, 'I-8': False,
'I-9': False, 'I-10': False, 'J-1': False, 'J-2': False, 'J-3': False, 'J-4': False,
'J-5': False, 'J-6': False, 'J-7': False, 'J-8': False, 'J-9': False, 'J-10':
False}}

Sala Fake recién construida: {'tipo': 'no soy una sala jijiji', 'n_filas': '-7 🤪',
'n_columnas': 'no tengo eso'}
```

USANDO EL getNombre EN SALA 1

Nombre de la Sala 1: Sala 1

USANDO EL getNombre EN SALA FAKE

Objeto de tipo incompatible: no soy una sala jijiji

Nombre de la Sala Fake: None

USANDO EL setNombre EN SALA 1

USANDO EL setNombre EN SALA FAKE

Objeto de tipo incompatible: no soy una sala jijiji

```
Sala 1 después del setNombre: {'tipo': 'sala', 'nombre': 'Sala Uno', 'filas': 10,
'columnas': 10, 'asientos': {'A-1': False, 'A-2': False, 'A-3': False, 'A-4': False,
'A-5': False, 'A-6': False, 'A-7': False, 'A-8': False, 'A-9': False, 'A-10': False,
'B-1': False, 'B-2': False, 'B-3': False, 'B-4': False, 'B-5': False, 'B-6': False,
'B-7': False, 'B-8': False, 'B-9': False, 'B-10': False, 'C-1': False, 'C-2': False,
'C-3': False, 'C-4': False, 'C-5': False, 'C-6': False, 'C-7': False, 'C-8': False,
'C-9': False, 'C-10': False, 'D-1': False, 'D-2': False, 'D-3': False, 'D-4': False,
'D-5': False, 'D-6': False, 'D-7': False, 'D-8': False, 'D-9': False, 'D-10': False,
'E-1': False, 'E-2': False, 'E-3': False, 'E-4': False, 'E-5': False, 'E-6': False,
'E-7': False, 'E-8': False, 'E-9': False, 'E-10': False, 'F-1': False, 'F-2': False,
'F-3': False, 'F-4': False, 'F-5': False, 'F-6': False, 'F-7': False, 'F-8': False,
'F-9': False, 'F-10': False, 'G-1': False, 'G-2': False, 'G-3': False, 'G-4': False,
'G-5': False, 'G-6': False, 'G-7': False, 'G-8': False, 'G-9': False, 'G-10': False,
'H-1': False, 'H-2': False, 'H-3': False, 'H-4': False, 'H-5': False, 'H-6': False,
'H-7': False, 'H-8': False, 'H-9': False, 'H-10': False, 'I-1': False, 'I-2': False,
'I-3': False, 'I-4': False, 'I-5': False, 'I-6': False, 'I-7': False, 'I-8': False,
'I-9': False, 'I-10': False, 'J-1': False, 'J-2': False, 'J-3': False, 'J-4': False,
'J-5': False, 'J-6': False, 'J-7': False, 'J-8': False, 'J-9': False, 'J-10':
False}}
```

```
Sala Fake después del setNombre: {'tipo': 'no soy una sala jijiji',
'n_filas': '-7 😊', 'n_columnas': 'no tengo eso'}
Estado del asiento G-8 en Sala Uno: False
El asiento M-15 no existe en Sala Uno
Estado del asiento M-15 en Sala Uno: None
Objeto de tipo incompatible: no soy una sala jijiji
Estado del asiento A-1 en Sala Fake: None
>>> |
```

Se puede ver que cuándo intentamos aplicar cualquier método a la `salaFake`, siempre salta un aviso, además los valores contenidos en `salaFake` nunca cambian al aplicar algún método modificador, ya que, igual que en ejemplo anterior, los métodos modificadores solo pueden alterar objetos de una clase específica, pero no de otras.

CONCLUSIÓN

Para cerrar todo con broche de oro, podemos observar cómo se utilizan diccionarios y funciones para crear objetos y definir sus atributos y comportamientos a partir de los ejemplos que hemos hecho. Cada uno de los códigos muestra la construcción de un objeto diferente: un ordenador y una sala de cine, y algunas aplicaciones de sus métodos relacionados.

Además de los atributos, los objetos también pueden tener comportamientos asociados, que se implementan como funciones que actúan sobre los datos del objeto. En los ejemplos proporcionados, vemos algunos de los métodos (funciones asociadas a un objeto), pero se podría ampliar la implementación agregando todavía métodos a cada uno de los códigos.

En resumen, la pseudo orientación a objetos en estos ejemplos se basa en la idea de representar los objetos como diccionarios que contienen atributos relacionados y puede incluir comportamientos adicionales a través de métodos asociados. Aunque no se muestra en los códigos proporcionados, los objetos también pueden interactuar entre sí, pueden pasarse como argumentos a funciones y realizar otras operaciones comunes en la programación orientada a objetos. Detalles que veremos más adelante.

¡Lo dejaremos por aquí, pero no te quedes ahí! Te animo a seguir explorando este nuevo concepto, ¡te será de gran utilidad en futuros proyectos!