```python
"""
En un clase "tradicional" todo atributo tendrá, para cada atributo:
    - uno o varios métodos para recuperar su valor. Estos métodos se llaman métodos Getter del atributo.
    - uno o varios métodos para cambiar su valor. Estos métodos se llaman métodos Setter del atributo.

Por ejemplo, para el atributo `luz: bool` de una casa se puede tener 1 método GET que se puede llamar `luz_encendida()`
para saber si la luz está encendida y puede tener 3 métodos SET para cambiar su estado: `encender()`, `apagar()`,
`cambiar_estado_luces(nuevo_valor)`.

Lo más usual es que exista uno o ningún método Setter, y uno o ningún método Getter.

En los ejemplos que vienen a continuación supondremos que existe un solo método SET y un solo método GET
"""

class MiClaseTradicional:
    """
    En esta clase se definen los métodos Get/Set de la forma tradicional, la forma estándar de hacerlo en cualquier
    lenguaje de programación.
        - Para recuperar el valor del atributo habrá que escribir `objeto.get_atributo()`
        - Para cambiar el valor del atributo habrá que escribir `objeto.set_atributo(valor)`
    Lo tradicional es que el nombre de los métodos se llamen igual que el del atributo.
    Si el atributo se llamara `x` entonces los métodos deberían llamarse `get_x()` y `set_x(valor)`.
    Siempre puedes cambiar el nombre del método, pero lo más normal es indicarlo como se indica.

    Nota: Los métodos Get/Set nunca deben usar print(). Aquí solo se usa por motivos docentes.
    """
    def __init__(self, un_x: str):
        self._x: str = un_x

    def get_x(self) -> str:
        print(f"Invocado el método GET de la clase {self.__class__}")
        return self._x

    def set_x(self, nuevo_x: str):
        print(f"Invocado el método SET de la clase {self.__class__}")
        self._x = nuevo_x


class MiClaseFuncionProperty:
    """
    En esta clase se definen los métodos Get/Set de la forma tradicional, la forma estándar de hacerlo en cualquier
    lenguaje de programación.
        - Para recuperar el valor del atributo habrá que escribir `objeto.get_atributo()`
        - Para cambiar el valor del atributo habrá que escribir `objeto.set_atributo(valor)`

    Ahora añadimos la instrucción `propiedad = property(fget=..,fset=..)`. Con esto estamos añadiendo una propiedad.
    Una propiedad es un miembro de la clase (como lo son los atributos, métodos, clases internas, ...)
    Una propiedad es un mecanismo con el que se puede leer, escribir o calcular el valor de un campo privado.
    Por tanto, una propiedad, si se define, sirve para gestionar el acceso a un atributo privado (y todos los
    atributos deben ser privados).
    Una propiedad es un miembro público que se usa como si fuera un campo (atributo o variable) pero realmente es un
    método especial que sirve para gestionar un atributo.

    Definida la propiedad, su uso es el siguiente:

        * Para recuperar el valor del atributo habrá que escribir `objeto.propiedad`.
            - `objeto.propiedad` invocará al método que se definió en el parámetro fget=.. de property()
            - Si se definió `propiedad = property(fget=funcionA, ...)`, entonces:
                    `objeto.propiedad` es equivalente a `objeto.funcionA()`

        * Para cambiar el valor del atributo habrá que escribir `objeto.propiedad = valor`.
            - `objeto.propiedad=valor` invocará al método que se definió en el parámetro fset=.. de property()
            - Si se definió `propiedad = property(fset=funcionB, ...)`, entonces:
                    `objeto.propiedad=valor` es equivalente a `objeto.funcionB(valor)`

        * Además, como el objetivo de una propiedad es servir de acceso a un campo privado, las funciones `funcionA`
        y `funcionB` deben de ser algún método GET y SET de dicho campo.

    En la práctica, si un campo se llama `_atributo` (para indicar que está oculto) su correspondiente propiedad,
    si se define, se suele llamar `atributo` (para indicar que hace referencia a una campo que se llama igual que él,
    pero que tiene un guion bajo).

    Si el atributo se llamara `_x` lo normal es que los métodos get/set se llamen get_x() y set_x(valor).
    Si el atributo se llamara `_x` lo normal es que su propiedad se llame x=property(...)
    Si juntamos las dos cosas, lo normal es escribir x = property(fset=set_x, fget=set_y)

    Nota: Los métodos Get/Set nunca deben usar print(). Aquí solo se usa por motivos docentes.
```

```
78      """
79      def __init__(self, un_x: str):
80          self._x: str = un_x
81
82      def get_x(self) -> str:
83          print(f"Invocado el método GET de la clase {self.__class__}")
84          return self._x
85
86      def set_x(self, nuevo_x: str):
87          print(f"Invocado el método SET de la clase {self.__class__}")
88          self._x = nuevo_x
89
90      x = property(fset=set_x, fget=get_x)
91
92
93  class MiClassDecoracionProperty:
94      """
95      En esta clase se muestra una forma alternativa al uso de `propiedad = property(fset=.., fget=..)`.
96      Ahora lo que hacemos es usar una decoración que se llama @property.
97      Cuando se antepone esta decoración a un método estamos indicando que dicho método pasa a ser una
    propiedad.
98      Y si es una propiedad su función es servir de acceso a un campo (privado).
99      En concreto, @property indica que el acceso es del tipo GET.
100     Así, su uso debe ser el siguiente si se tiene en cuenta que el nombre de la propiedad debe ser igual
    que el del
101     campo al que accede (pero sin el primer guión bajo):
102
103         @property
104         def atributo(self):
105             return self._atributo
106
107     Para decorar el método que actuará de propiedad SET se usa la decoración @atributo.setter y su uso es:
108
109         @atributo.setter
110         def atributo(self, valor):
111             self._atributo = valor
112
113     Esta es la única forma en la que habrás visto en Python la posibilidad de hacer sobrecarga. Aparace el
    mismo
114     método `atributo()` definido dos veces con una cantidad diferente de parámetros de entrada. Con (self)
    y con
115     (self, valor)
116     """
117     def __init__(self, un_x: str):
118         self._x: str = un_x
119
120     @property
121     def x(self) -> str:
122         print(f"Invocando a la propiedad -get- de {self.__class__}")
123         return self._x
124
125     @x.setter
126     def x(self, nuevo_x: str):
127         print(f"Invocando a la propiedad -set- de {self.__class__}")
128         self._x = nuevo_x
129
130
131
132 def uso_tradicional():
133     cosa: MiClaseTradicional = MiClaseTradicional("Tradicional")
134     print(f"Valor inicial = {cosa.get_x()}")
135     cosa.set_x("Tradicional modificado")
136     print(f"Nuevo valor = {cosa.get_x()}")
137
138
139 def uso_property():
140     cosa: MiClaseFuncionProperty = MiClaseFuncionProperty("Property")
141     print(f"Valor inicial = {cosa.x}")
142     cosa.x = "Property modificado"
143     print(f"Nuevo valor = {cosa.x}")
144
145
146 def uso_decoracion():
147     cosa: MiClassDecoracionProperty = MiClassDecoracionProperty("Decoración")
148     print(f"Valor inicial = {cosa.x}")
149     cosa.x = "Decoración modificado"
150     print(f"Nuevo valor = {cosa.x}")
151
152
153 if __name__ == '__main__':
154     metodos = [uso_tradicional, uso_property, uso_decoracion]
155     for m in metodos:
156         print('='*30)
157         m()
158         print('=' * 30)
159
```