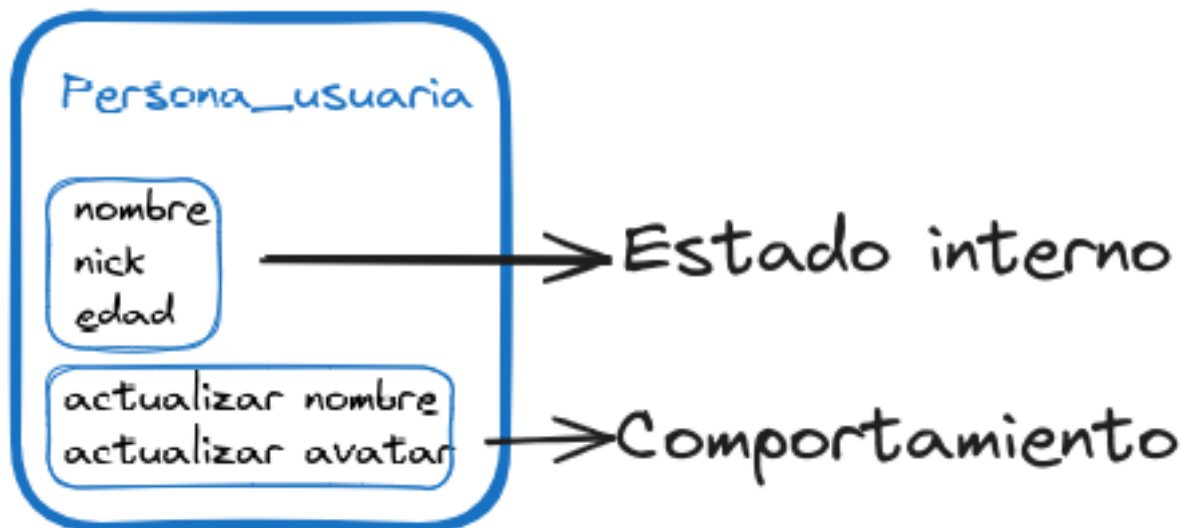


# Seminario de Lenguajes - Python

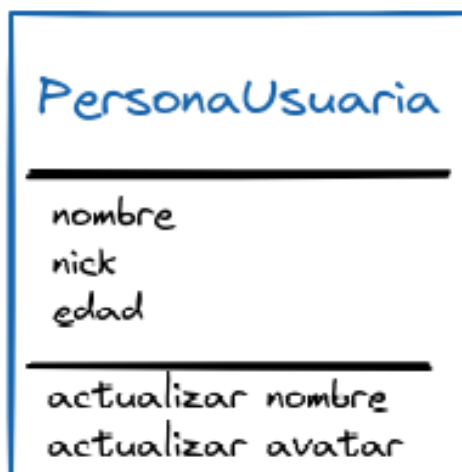
## Repaso de POO. Propiedades

### Sobre los objetos

- Son los elementos fundamentales de la POO.
- Son entidades que poseen **estado interno** y **comportamiento**.



### Pensemos en la clase `PersonaUsuaría`



- Cuando creamos un objeto, creamos una **instancia de la clase**.
- Una **instancia** es un **objeto individualizado** por los valores que tomen sus atributos o propiedades.



Observemos este código: ¿qué diferencia hay entre `name` y `_members`?

```
In [ ]: class Band():
    """ Define la entidad que representa a una banda .. """
    all_genres = set()

    def __init__(self, name, genres="rock"):
        self.name = name
        self.genres = genres
        self._members = []
        Band.all_genres.add(genres)

    def add_member(self, new_member):
        self._members.append(new_member)
```

## Público y privado

- Antes de empezar a hablar de esto ....

“Private” instance variables that cannot be accessed except from inside an object don’t exist in Python.”

- De nuevo.. en español..

“Las variables «privadas» de instancia, que no pueden accederse excepto desde dentro de un objeto, no existen en Python”

- ¿Y entonces?
- Más info: <https://docs.python.org/3/tutorial/classes.html#private-variables>

## Hay una convención ..

Es posible **definir el acceso** a determinados métodos y atributos de los objetos, quedando claro qué cosas se pueden y no se pueden utilizar desde **fuera de la clase**.

- **Por convención**, todo atributo (variable de instancia o método) que comienza con "\_" se considera no público.
- Pero esto no impide que se acceda. **Simplemente es una convención.**

```
In [ ]: class User():
        """Define la entidad que representa a un usuario en PyTrivia"""

        def __init__(self, name="Sara Connor", nick="mamá_de_John"):
            self._name = name
            self.nick = nick
            self.avatar = None
        #Métodos
        def set_name(self, new_name):
            self._name = new_name
```

```
In [ ]: obj = User()
        print(obj._name)
```

## getters y setters

```
In [ ]: class Demo:
        def __init__(self):
            self._x = 0
            self.y = 10

        def get_x(self):
            return self._x

        def set_x(self, value):
            self._x = value
```

- ¿Cuántas **variables de instancia**?
- Por cada variable de instancia **no pública** tenemos un método **get** y un método **set**. O, como veremos a continuación: **propiedades**.

## Propiedades

- Podemos definir a x como una **propiedad** de la clase. ¿Qué significa esto? ¿Cuál es la ventaja?

```
In [ ]: class Demo:
        def __init__(self):
            self._x = 0

        def get_x(self):
            print("estoy en get")
            return self._x

        def set_x(self, value):
            print("estoy en set")
            self._x = value

        x = property(get_x, set_x)
```

```
In [ ]: obj = Demo()
        print(obj.x)
```

```
In [ ]: obj.x = 10
        print(obj.x)
```

## La función property()

- **property()** crea una **propiedad** de la clase.
- Forma general:

```
property(fget=None, fset=None, fdel=None, doc=None)
```

- [+Info](#)

## El ejemplo completo

```
In [ ]: class Demo:
        def __init__(self):
            self._x = 0
        def get_x(self):
            print("estoy en get")
            return self._x
        def set_x(self, value):
            print("estoy en set")
            self._x = value
        def del_x(self):
            print("estoy en del")
            del self._x

        x = property(get_x, set_x, del_x, "x es una propiedad")
```

```
In [ ]: obj = Demo()
        obj.x = 10
        print(obj.x)
        del obj.x
```

## Más sobre property()

- ¿Qué pasa con el siguiente código si la **propiedad x** se define de la siguiente manera?:

```
In [ ]: class Demo:
        def __init__(self):
            self._x = 0

        def get_x(self):
            return self._x

        def set_x(self, value):
            self._x = value

        x = property(get_x)
```

```
In [ ]: obj = Demo()  
obj.x = 10
```

## ¿Y esto?

```
In [ ]: class Demo:  
    def __init__(self):  
        self._x = 0  
  
    @property  
    def x(self):  
        #print("estoy en get")  
        return self._x
```

```
In [ ]: obj = Demo()  
print(obj.x)
```

- @property es un **decorador**.

## ¿Qué es un decorador?

Un **decorador es una función** que recibe una función como argumento y **extiende** el comportamiento de esta última función sin modificarla explícitamente.

### RECORDAMOS: las funciones son objetos de primera clase

- **¿Qué significa esto?** Pueden ser asignadas a variables, almacenadas en estructuras de datos, pasadas como argumentos a otras funciones e incluso retornadas como valores de otras funciones.

## Observemos el siguiente código

```
In [ ]: def decimos_hola(nombre):  
    return f"Hola {nombre}!"  
  
def decimos_chau(nombre):  
    return f"Chau {nombre}!"
```

```
In [ ]: def saludo_a_Clau(saludo):  
    return saludo("Clau")
```

```
In [ ]: saludo_a_Clau(decimos_hola)  
#saludo_a_Clau(decimos_chau)
```

## ¿Qué podemos decir de este ejemplo?

- Ejemplo sacado de <https://realpython.com/primer-on-python-decorators/>

```
In [ ]: def decorador(funcion):  
    def funcion_interna():  
        print("Antes de invocar a la función.")  
        funcion()
```

```

        print("Después de invocar a la función.")

    return funcion_interna

def decimos_hola():
    print("Hola!")

```

```
In [ ]: saludo = decorador(decimos_hola)
```

- ¿De qué tipo es saludo?

```
In [ ]: def decorador(funcion):
        def funcion_interna():
            print("Antes de invocar a la función.")
            funcion()
            print("Después de invocar a la función.")
        return funcion_interna

def decimos_hola():
    print("Hola!")

saludo = decorador(decimos_hola)

```

- ¿A qué función hace referencia saludo?

```
In [ ]: saludo()
```

## Otra forma de escribir esto en Python:

```
In [ ]: def decorador(funcion):
        def funcion_interna():
            print("Antes de invocar a la función.")
            funcion()
            print("Después de invocar a la función.")
        return funcion_interna

@decorador
def decimos_hola():
    print("Hola!")

```

```
In [ ]: decimos_hola()
```

## Es equivalente a:

```
decimos_hola = decorador(decimos_hola)
```

- [+Ínfó](#)
- [+Info en español](#)

## Dijimos que @property es un decorador

```
In [ ]: class Demo:
        def __init__(self):
            self._x = 0

```

```
@property
def x(self):
    return self._x
```

```
In [ ]: obj = Demo()
obj.x = 10 # Esto dará error: ¿por qué?
print(obj.x)
```

- ATENCIÓN: x no es un método, es una propiedad.
- [+Info](#)

## El ejemplo completo

```
In [ ]: class Demo:
    def __init__(self):
        self._x = 0
    @property
    def x(self):
        print("Estoy en get")
        return self._x

    @x.setter
    def x(self, value):
        print("Estoy en set")
        self._x = value
```

```
In [ ]: obj = Demo()
obj.x = 10
print(obj.x)
#del obj.x
```

## Algo más para leer

<https://realpython.com/python-getter-setter/>

