

Clase 8_POO

May 1, 2023

1 Seminario de Lenguajes - Python

1.1 Cursada 2023

1.1.1 Clase 8: conceptos de la POO

2 Pensemos en la siguiente situación:

- En el trabajo integrador tenemos que registrar los datos de quienes interactúan con la aplicación.
- Podemos pensar en una entidad “Usuario” con datos asociados tales como: - nombre - nick - género - edad
- También podríamos pensar en: - avatar - ultimo_acceso - paleta de colores elegida ## Con lo visto hasta el momento, ¿qué estructura de datos podemos elegir para representar a un usuario?

3 Podríamos utilizar diccionarios

```
[ ]: usuario = {'nombre': 'Tony', 'nick': 'Ironman', 'genero': "masculino", "edad": 40, "avatar": "ironman.png"}
```

3.1 ¿Podemos asociar funcionalidades específicas a este “usuario”?

Por ejemplo, `cambiar_nombre`, `registrar_actividad`, `ver_avatar`, etc.

4 Podríamos definir funciones para definir la funcionalidad asociada

```
[ ]: def cambiar_nombre(usuario, nuevo_nombre):  
    """ Modifica el nombre del usuario  
  
    usuario: representa al usuario con el que queremos operar  
    nuevo_nombre: un str con el nuevo nombre  
    """  
    usuario["nombre"] = nuevo_nombre
```

5 Pero...

- ¿Podemos modificar a “nuestro usuario” sin utilizar estas funciones?
- ¿Podemos decir que “nuestro usuario” es una **entidad** que **encapsula** tanto su estructura como la funcionalidad para manipularlo?

Si...y no...

6 Hablemos de objetos ...

7 Un objeto es una colección de datos con un comportamiento asociado en una única entidad

8 Objetos

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

9 Objetos

- Ya vimos que en Python **todos** los elementos con los que trabajamos son objetos.

```
cadena = "Hola"  
archivo = open("archi.txt")
```

```
cadena.upper()  
archivo.close()
```

- **cadena** y **archivo** referencian a **objetos**.
- **upper** y **close** forman parte del comportamiento de estos objetos: son **métodos**.

10 POO: conceptos básicos

- En POO un programa puede verse como un **conjunto de objetos** que interactúan entre ellos **enviándose mensajes**.
- Estos mensajes están asociados al **comportamiento** del objeto (conjunto de **métodos**).

11 El mundo de los objetos

- ¿Qué representa cada objeto?
- ¿Qué podemos decir de cada grupo de objetos?

12 Objetos y clases

- No todos los objetos son iguales, ni tienen el mismo comportamiento.
- Así agrupamos a los objetos de acuerdo a características comunes.

Una clase describe las propiedades o atributos de objetos y las acciones o métodos que pueden hacer o ejecutar dichos objetos.

13 Pensemos en la clase Usuario

- 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.
- La **interfaz pública** del objeto está formada por las propiedades y métodos que otros objetos pueden usar para interactuar con él.
- ¿Qué pasa si todas las propiedades y métodos son privadas? ¿Y si son todas públicas?

14 Clases en Python

```
class NombreClase:  
    sentencias  
    ...
```

- La [PEP 8](#) sugieren usar CamelCase en el caso del nombre de las clases.
- Al igual que las funciones, las clases **deben** estar definidas antes de que se utilicen.
- Con la definición de una nueva clase se crea un nuevo **espacio de nombres**.

14.0.1 ¿Cómo se crea una instancia de una clase?

```
objeto = NombreClase()
```

15 La clase Usuario

```
[ ]: class Usuario():  
    """Define la entidad que representa a un usuario en UNLPImage"""  
  
    #Propiedades  
    nombre = 'Tony Stark'  
    nick = 'Ironman'  
    avatar = None  
  
    #Métodos  
    def cambiar_nombre(self, nombre):  
        self.nombre = nombre
```

- ¿self?
- ¿Qué quiere decir que Usuario tiene su propio espacio de nombres?

16 Creamos las instancias

```
[ ]: tony = Usuario()
      print(tony.nombre)
      tony.cambiar_nombre("Tony")
      print(tony.nombre)
```

- Observemos la línea 3: `tony.cambiar_nombre("Tony")`
 - Atención a la cantidad de parámetros pasados.
- Cuando creamos otros objetos de clase **Usuario**, ¿qué particularidad tendrán?

```
[ ]: otro_usuario = Usuario()
```

```
[ ]: print(otro_usuario.nombre)
```

17 Podemos parametrizar la creación de objetos

```
[ ]: class Usuario():
      """ Define la entidad que representa a un usuario en UNLPImage """

      def __init__(self, nom, alias):
          self.nombre = nom
          self.nick = alias
          self.avatar = None
      #Métodos
      def cambiar_nombre(self, nombre):
          self.nombre = nombre
```

```
[ ]: tony = Usuario('Tony Stark', 'Ironman')
      tony.cambiar_nombre("Tony")
```

- El método `init()` se invoca automáticamente al crear el objeto.

18 ¿Qué pasa si..?

```
[ ]: otro_usuario = Usuario()
```

18.1 Podemos inicializar con valores por defecto

```
[ ]: class Usuario():
      """ Define la entidad que representa a un usuario en UNLPImage """

      def __init__(self, nom="Tony Stark", alias="Ironman"):
          self.nombre = nom
          self.nick = alias
          self.avatar = None
```

```
#Métodos
def cambiar_nombre(self, nombre):
    self.nombre = nombre
```

```
[ ]: tony = Usuario()
      bruce = Usuario("Bruce Wayne", "Batman")
      print(tony.nombre)
      print(bruce.nombre)
```

19 Desafío

Estamos armando un curso y queremos modelar con clases los distintos recursos con los que vamos a trabajar. Cada recurso tiene un nombre, una URL donde está publicado, un tipo (para indicar si se encuentra en formato PDF, jupyter o video) y la fecha de su última modificación.

Crear la clase para trabajar con estos datos.

20 Tarea para el hogar ...

```
[ ]: class Recurso:
      ...
```

- ¿Qué debemos pensar?
 - ¿Qué propiedades tiene un recurso?
 - ¿Cuál es el comportamiento? ¿Cuáles son los métodos asociados?

21 Observemos este código: ¿qué diferencia hay entre villanos y enemigos?

```
[ ]: class SuperHeroe():
      villanos = []

      def __init__(self, nombre, alias):
          self.nombre = nombre
          self.enemigos = []
```

- **villanos** es una **variable de clase** mientras que **enemigos** es una **variable de instancia**.
- ¿Qué significa esto?

22 Variables de instancia vs. de clase

Una **variable de instancia** es exclusiva de cada instancia u objeto.

Una **variable de clase** es única y es **compartida por todas las instancias** de la clase.

23 Veamos el ejemplo completo:

```
[ ]: class SuperHeroe():  
    """ Esta clase define a un superheroe  
    villanos: representa a los enemigos de todos los superhéroes  
    """  
    villanos = []  
  
    def __init__(self, nombre, alias):  
        self.nombre = nombre  
        self.enemigos = []  
  
    def get_enemigos(self):  
        return self.enemigos  
  
    def agregar_enemigo(self, otro_enemigo):  
        "Agrega un enemigo a los enemigos del superhéroe"  
  
        self.enemigos.append(otro_enemigo)  
        SuperHeroe.villanos.append(otro_enemigo)
```

```
[ ]: batman = SuperHeroe( "Bruce Wayne", "Batman")  
    ironman = SuperHeroe( "Tony Stark", "ironman")  
  
    batman.agregar_enemigo("Joker")  
    batman.agregar_enemigo("Pinguino")  
    batman.agregar_enemigo("Gatubela")  
  
    ironman.agregar_enemigo("Whiplash")  
    ironman.agregar_enemigo("Thanos")
```

```
[ ]: # OJO que esta función está FUERA de la clase  
def imprimo_villanos(nombre, lista_de_villanos):  
    "imprime la lista de todos los villanos de nombre"  
    print("\n"+"*" * 40)  
    print(f"Los enemigos de {nombre}")  
    print("*" * 40)  
    for malo in lista_de_villanos:  
        print(malo)
```

```
[ ]: imprimo_villanos(batman.nombre, batman.get_enemigos())  
    imprimo_villanos(ironman.nombre, ironman.get_enemigos())
```

```
[ ]: imprimo_villanos("todos los superhéroes", SuperHeroe.villanos)
```

24 Python me permite cosas como éstas:

```
[ ]: class SuperHeroe:
    pass

tony = SuperHeroe()
tony.nombre = "Tony Stark"
tony.alias = "Ironman"
tony.soy_Ironman = lambda : True if tony.alias == "Ironman" else False

tony.soy_Ironman()
#tony.nombre
```

```
[ ]: del tony.nombre
tony.nombre
```

- ¿Qué significa esto?
- ¡¡Aunque esto no sería lo más indicado de hacer!! ¿Por qué?

25 Volvamos a este código: ¿no hay algo que parece incorrecto?

```
[ ]: class SuperHeroe():
    villanos = []

    def __init__(self, nombre, alias):
        self.nombre = nombre
        self.enemigos = []
```

```
[ ]: batman = SuperHeroe("Bruce", "Batman")
print(batman.nombre)
```

26 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>

27 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 (propiedad o método) que comienza con `"_"` se considera no público.
- Pero esto no impide que se acceda. **Simplemente es una convención.**

28 Privado por convención

```
[ ]: class Usuario():
    "Define la entidad que representa a un usuario en UNLPImage"
    def __init__(self, nom="Sara Connor", alias="mama_de_John"):
        self._nombre = nom
        self.nick = alias
        self.avatar = None
    #Métodos
    def cambiar_nombre(self, nuevo_nombre):
        self._nombre = nuevo_nombre

obj = Usuario()
print(obj._nombre)
```

- Hay otra forma de indicar que algo no es “tan” público: agregando a los nombres de la variables o funciones, dos guiones `__(__)` delante.

29 Veamos este ejemplo: códigos secretos

```
[ ]: class CodigoSecreto:
    '''¿¿¿Textos con clave???'''

    def __init__(self, texto_plano, clave_secreta):
        self.__texto_plano = texto_plano
        self.__clave_secreta = clave_secreta

    def desencriptar(self, clave_secreta):
        '''Solo se muestra el texto si la clave es correcta'''

        if clave_secreta == self.__clave_secreta:
            return self.__texto_plano
        else:
            return ''
```

- ¿Cuáles son las propiedades? ¿Públicas o privadas?

- ¿Y los métodos? ¿Públicos o privados?
- ¿Cómo creo un objeto `CodigoSecreto`?

30 Codificamos textos

```
class CodigoSecreto:
    '''¿¿¿Textos con clave????'''

    def __init__(self, texto_plano, clave_secreta):
        self.__texto_plano = texto_plano
        self.__clave_secreta = clave_secreta

    def desencriptar(self, clave_secreta):
        '''Solo se muestra el texto si la clave es correcta'''
        if clave_secreta == self.__clave_secreta:
            return self.__texto_plano
        else:
            return ''
```

```
[ ]: texto_secreto = CodigoSecreto("Seminario Python", "stark")
```

```
[ ]: print(texto_secreto.desencriptar("stark"))
```

31 ¿Qué pasa si quiero imprimir desde fuera de la clase: `**texto_secreto.__texto_plano**`?

```
[ ]: print(texto_secreto.__texto_plano)
```

31.1 Entonces, ¿sí es privado?

32 Códigos no tan secretos

- Ahora, probemos esto:

```
[ ]: print(texto_secreto._CodigoSecreto__texto_plano)
```

- Todo identificador que comienza con `**"__"`, por ejemplo `__texto_plano`, es reemplazado textualmente por `__NombreClase__`, por ejemplo: `__CodigoSecreto__texto_plano`.
- [+Info](#)

33 Entonces... respecto a lo público y privado

33.1 Respetaremos las convenciones

- Si el nombre de una propiedad comienza con `__` será considerada privada. Por lo tanto no podrá utilizarse directamente desde fuera de la clase.
- Aquellas propiedades que consideramos públicas, las usaremos como tal. Es decir, que pueden utilizarse fuera de la clase.

```
[ ]: class Usuario():
    "Define la entidad que representa a un usuario en UNLPImage"
    def __init__(self, nom="Sara Connor", alias="mama_de_John"):
        self._nombre = nom
        self.nick = alias
        self._avatar = None
    #Métodos
    def cambiar_nombre(self, nuevo_nombre):
        self._nombre = nuevo_nombre

obj = Usuario()
print(obj.nick)
```

34 getters y setters

```
[ ]: 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 más adelante: **propiedades**.

35 Algunos métodos especiales

Mencionamos antes que los `__` son especiales en Python. Por ejemplo, podemos definir métodos con estos nombres:

- `__lt__`, `__gt__`, `__le`, `__ge__`
- `__eq__`, `__ne__`

En estos casos, estos métodos nos permiten comparar dos objetos con los símbolos correspondientes:

- $x < y$ invoca `x.__lt__(y)`,
- $x \leq y$ invoca `x.__le__(y)`,
- $x == y$ invoca `x.__eq__(y)`,
- $x \neq y$ invoca `x.__ne__(y)`,
- $x > y$ invoca `x.__gt__(y)`,
- $x \geq y$ invoca `x.__ge__(y)`.

```
[ ]: class Banda():
    """ Define la entidad que representa a una banda .. """

    def __init__(self, nombre, genero="rock"):
        self.nombre = nombre
        self.genero = genero
        self._integrantes = []

    def agregar_integrante(self, nuevo_integrante):
        self._integrantes.append(nuevo_integrante)

    def __lt__(self, otra):
        return len(self._integrantes) < len(otra._integrantes)
```

- ¿Qué implementa el método `__lt__`?
- ¿Cuándo una banda es “menor” que otra?

```
[ ]: soda = Banda("Soda Stereo")
soda.agregar_integrante("Gustavo Ceratti")
soda.agregar_integrante("Zeta Bosio")
soda.agregar_integrante("Charly Alberti")

seru = Banda("Seru Giran")
seru.agregar_integrante("Charly García")
seru.agregar_integrante("David Lebon")
seru.agregar_integrante("Oscar Moro")
seru.agregar_integrante("Pedro Aznar")
```

```
[ ]: menor = soda.nombre if soda < seru else seru.nombre
menor
```

36 El método `__str__`

Retorna una cadena de caracteres (**str**) con la representación que querramos mostrar del objeto.

```
[ ]: class Banda():
    """ Define la entidad que representa a una banda .. """

    def __init__(self, nombre, genero="rock"):
        self.nombre = nombre
```

```
self.genero = genero
self._integrantes = []

def agregar_integrante(self, nuevo_integrante):
    self._integrantes.append(nuevo_integrante)

def __str__(self):
    return (f"{self.nombre} está integrada por {self._integrantes}")
```

```
[ ]: soda = Banda("Soda Stereo")
soda.agregar_integrante("Gustavo Ceratti")
soda.agregar_integrante("Zeta Bosio")
soda.agregar_integrante("Charly Alberti")

print(soda)
```

-Info

37 Seguimos la próxima ...