Clase7_1_Intro_POO

April 29, 2022

1 Seminario de Lenguajes - Python

1.1 Cursada 2022

1.1.1 Introducción a POO

2 Pensemos en la siguiente situación:

- En el trabajo integrador tenemos que registrar los datos de quien juega.
- Podemos pensar en una entidad "Jugador" con datos asociados tales como: - nombre - nick - contraseña? - género auto percibido - edad
- También podríamos pensar en: - puntaje acumulado - vidas restantes - paleta de colores elegida

2.1 ¿Con qué estructura de datos pensaron implementar esto con lo visto hasta ahora?

3 Podríamos utilizar diccionarios

[ ]:
```
jugador = {'nombre': 'Tony', 'nick': 'Ironman', "genero": "masculino", "edad":␣
↪40, "puntos": 0}
```

3.1 ¿Podemos asociar funcionalidades específicas a este "jugador"?

Por ejemplo, **incrementar edad**, **cambiar paleta de colores**, **modificar puntaje** etc.

4 Podríamos definir funciones para definir la funcionalidad asociada

[ ]:
```
def incrementar_puntaje(jugador, puntos):
    """ Incrementa los puntos del jugador

    jugador:
    puntos:
    """
    jugador["puntos"] += puntos

def decrementar_puntaje(jugador, puntos):
```

```
    """ ... """
    jugador["puntos"] -= puntos
```

5 Pero...

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

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**.
- 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**.

9 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**).

10 El mundo de los objetos

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

11 Objetos y clases

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

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

12 Pensemos en la clase Jugador

- 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?

13 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 **espacio de nombres nuevo**.

13.0.1 ¿Cómo se crea una instancia de una clase?

```
objeto =  NombreClase()
```

14 La clase Jugador

[ ]:
```
class Jugador():
    """Define la entidad que representa a un jugador en el juego"""

    #Propiedades
    nombre = 'Tony Stark'
    nick = 'Ironman'
    puntos = 0

    #Métodos
    def incrementar_puntos(self, puntos):
        self.puntos += puntos
```

- ¿self?

15 Creamos las instancias

[ ]:
```
tony = Jugador()
tony.incrementar_puntos(10)
print(tony.nombre)
```

- **tony.incrementar_puntos()**
  - Atención a la cantidad de parámetros pasados.
- Cuando creamos otros objetos de clase **Jugador**, ¿qué particularidad tendrán?

[ ]:
```
otro_jugador = Jugador()
```

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

16 Podemos parametrizar la creación de objetos

[ ]:
```
class Jugador():
    """ Define la entidad que representa a un jugador en el juego"""

    def __init__(self, nom, nic):
        self.nombre = nom
        self.nick = nic
        self.puntos = 0
    #Métodos
    def incrementar_puntos(self, puntos):
        self.puntos += puntos
```

[ ]:
```
tony = Jugador('Tony Stark','Ironman')
tony.incrementar_puntos(10)
```

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

17 ¿Qué pasa si..?

[ ]:
```
otro_jugador = Jugador()
```

17.1 Podemos inicializar con valores por defecto

[ ]:
```
class Jugador():
    """ Define la entidad que representa a un jugador en el juego"""

    def __init__(self, nom="Tony Stark", nic="Ironman"):
        self.nombre = nom
        self.nick = nic
        self.puntos = 0
    #Métodos
```

```
    def incrementar_puntos(self, puntos):
        self.puntos += puntos
```

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

18 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?

19 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_nombre(self):
        return self.nombre

    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)
```

[ ]:
```
# 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)

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")
```

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

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

20 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?
- ¡¡Aunque esto no sería lo más indicado de hacer!!

21 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)
```

22 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

23 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**.

24 Privado por convención

[ ]:
```
class Jugador():
    "Define la entidad que representa a un jugador en el juego"
    def __init__(self, nom="Tony Stark", nic="Ironman"):
        self._nombre = nom
        self.nick = nic
        self.puntos = 0
    #Métodos
    def incrementar_puntos(self, puntos):
        self.puntos += puntos
```

```
tony = Jugador()
print(tony._nombre)
```

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

25 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**?

26 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"))
print(texto_secreto.__texto_plano)
```

- ¿Qué pasa si tenemos quiero imprimir desde fuera de la clase: **texto_secreto.___texto_plano**?

26.1 Entonces, ¿sí es privado?

27 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___identificador, por ejemplo: **_CodigoSecreto___texto_plano**.
- +Info

28 Entonces... respecto a lo público y privado

28.1 Respetaremos las convenciones

28.1.1 Todo identificador que comienza con **"_"** será considerado privado.

- **Tarea:** buscar qué dice la PEP 8 sobre esto.

29 Algunos métodos especiales

Mencionamos antes que los "___" son especales 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<=y invoca x.___le___(y),
- x==y invoca x.___eq___(y),
- x!=y invoca x.___ne___(y),
- x>y invoca x.___gt___(y),
- x>=y invoca x.___ge___(y).

[ ]:
```
class Jugador:
    """ .. """
    def __init__(self, nom="Tony Stark", nic="Ironman"):
        self._nombre = nom
        self.nick = nic
        self.puntos = 0
    def __lt__(self, otro):
        return (self._nombre < otro._nombre)
```

```
    def __eq__(self, otro):
        return (self.nick == otro.nick)
    def __ne__(self, otro):
        return (self._nombre != otro._nombre)

tony = Jugador()
bruce = Jugador("Bruce Wayne", "Batman")

if bruce < tony:
    print("Mmmm.... Algo anda mal..")
print("Son iguales" if tony == bruce else "Son distintos")
```

30 El método ___str___

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

[ ]:
```
class Jugador:
    """ .. """
    def __init__(self, nom="Tony Stark", nic="Ironman"):
        self._nombre = nom
        self.nick = nic
        self.puntos = 0
    def __str__(self):
        return (f"{self._nombre}, mejor conocido como {self.nick}")
    def __lt__(self, otro):
        return (self._nombre < otro._nombre)
    def __eq__(self, otro):
        return (self.nick == otro.nick)
    def __ne__(self, otro):
        return (self._nombre != otro._nombre)
tony = Jugador()
bruce = Jugador("Bruce Wayne", "Batman")

print(tony)
print(tony if tony == bruce else bruce)
```

-Info

31 Juegos electrónicos

- Un jugador de LOL **"es un"** Jugador.

32 Herencia

- Es uno de los conceptos más importantes de la POO.

- La herencia permite que una clase pueda *heredar* los atributos y métodos de otra clase, que se "agregan" a los propios.
- Este concepto permite "sumar", es decir "extender" una clase.
- La clase que hereda se denomina **clase derivada** y la clase de la cual se deriva se denomina **clase base**.
- Así, **Jugador es la clase base** y **JugadorDeLOL es la clase derivada**.

33 Escribamos el código

[ ]:
```
class Jugador:
    def __init__(self, nombre, juego="Tetris", tiene_equipo=False, equipo=None):
            self.nombre = nombre
            self.juego = juego
            self.tiene_equipo = tiene_equipo
            self.equipo = equipo
    def jugar(self):
            if self.tiene_equipo:
                    print (f"{self.nombre} juega en el equipo {self.equipo} al␣
 ↪{self.juego}")
            else:
                    print(f"{self.nombre} juega solo al {self.juego}")

class JugadorDeFIFA(Jugador):
    def __init__(self, nombre, equipo):
            Jugador.__init__(self, nombre, "FIFA", True, equipo)

class JugadorDeLOL(Jugador):
    def __init__(self, nombre, equipo):
            Jugador.__init__(self, nombre, "LOL")

nico = JugadorDeFIFA('Nico Villalba', "Guild Esports")
nico.jugar()
faker = JugadorDeLOL("Faker", "SK Telecom")
faker.jugar()
```

34 ¿e-sports?

- Un jugador de FIFA **"es un"** Jugador, pero también **"es un"** Deportista.

34.1 Python tiene herencia múltiple

35 ¿A qué jugamos?

[ ]:
```
class Jugador:
    def __init__(self, nombre, juego="No definido", tiene_equipo= False,␣
 ↪equipo=None):
        self.nombre = nombre
        self.juego = juego
        self.tiene_equipo = tiene_equipo
        self.equipo = equipo

    def jugar(self):
        if self.tiene_equipo:
            print (f"{self.nombre} juega en el equipo {self.equipo} al {self.
 ↪juego}")
        else:
            print(f"{self.nombre} juega solo al {self.juego}")

class Deportista:
    def __init__(self, nombre, equipo = None):
        self.nombre = nombre
        self.equipo = equipo

    def jugar(self):
        print (f"Mi equipo es {self.equipo}")
```

[ ]:
```
class JugadorDeFIFA(Jugador, Deportista):
    def __init__(self, nombre, equipo):
        Jugador.__init__(self, nombre, "PS4", True, equipo)
        Deportista.__init__(self,nombre, equipo)

class JugadorDeLOL(Deportista, Jugador):
    def __init__(self, nombre, equipo):
        Jugador.__init__(self, nombre, "LOL")
        Deportista.__init__(self, nombre, equipo)
```

[ ]:
```
nico = JugadorDeFIFA('Nico Villalba', "Guild Esports")
nico.jugar()
faker = JugadorDeLOL("Faker", "SK Telecom")
faker.jugar()
```

- Ambas clases bases tienen definido un método **jugar**.
  - En este caso, se toma el método de la clase más a la **izquierda** de la lista.
- Por lo tanto, es MUY importante el orden en que se especifican las clases bases.

36 MRO: Method Resolution Order

```
class JugadorDeFIFA(Jugador, Deportista):
    def __init__(self, nombre, equipo):
        Jugador.__init__(self, nombre, "PS4", True, equipo)
        Deportista.__init__(self,nombre, equipo)
```

[ ]:
```
JugadorDeFIFA.__mro__
```

37 Resumiendo...

37.1 Objetos y clases

- La **clase** define las propiedades y los métodos de los objetos.
- Los **objetos** son instancias de una clase.
- Cuando se crea un objeto, se ejecuta el método **___init()___** que permite inicializar el objeto.
- La definición de la clase especifica qué partes son privadas y cuáles públicas.

38 Mensajes y métodos

TODO el procesamiento en este modelo es activado por mensajes entre objetos.

- El **mensaje** es el modo de comunicación entre los objetos. Cuando se invoca una función de un objeto, lo que se está haciendo es **enviando un mensaje** a dicho objeto.
- El **método** es la función que está asociada a un objeto determinado y cuya ejecución sólo puede desencadenarse a través del envío de un mensaje recibido.

39 Seguimos la próxima ...