Clase8_POO

May 2, 2022

1 Seminario de Lenguajes - Python

1.1 Cursada 2022

1.1.1 Aspectos básicos de POO

2 Repasemos algunos conceptos vistos previamente

#

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

3 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**).

4 El mundo de los objetos

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

5 Objetos y clases

##

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

6 La clase SuperHeroe

[ ]:
```
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)
```

¿self?¿Cuáles son las variables de instancias?¿Y los métodos?¿Y villanos?

7 Creamos instancias de SuperHeroes

[ ]:
```
batman = SuperHeroe( "Bruce Wayne", "Batman")
ironman = SuperHeroe( "Tony Stark", "ironman")
```

Agregamos enemigos...

[ ]:
```
batman.agregar_enemigo("Joker")
batman.agregar_enemigo("Pinguino")
batman.agregar_enemigo("Gatubela")

ironman.agregar_enemigo("Whiplash")
ironman.agregar_enemigo("Thanos")
```

8 Mostramos los enemigos

Definimos una función,FUERA de la clase, que dado el nombre de un SuperHeroes, muestra en pantalla la lista de sus enemigos.

[ ]:
```
def imprimo_villanos(nombre, lista_de_villanos):
    "imprime  la lista de todos los villanos de nombre"
    print("*"*40)
    print(f"Los enemigos de {nombre}")
    print("*"*40)
    for malo in lista_de_villanos:
        print(malo)
```

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

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

8.1 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.

¿Cómo se especifica privado o público en Python?

9 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.
- 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?

#

¿e-sports?

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

##

Python tiene **herencia múltiple**

[ ]:
```
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}")
```

```python
class Deportista:
    def __init__(self, nombre, equipo = None):
        self.nombre = nombre
        self.equipo = equipo
    def jugar(self):
        print (f"Mi equipo es {self.equipo}")
```

[ ]:
```python
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)
```

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

10 ¿Qué términos asociamos con la programación orientada a objetos?

- Completar: https://www.menti.com/c49iykk8fw

¿

Resultados

11 Destacados ...

- Encapsulamiento
  - **class**, métodos privados y públicos, propiedades.
- Herencia
  - Clases bases y derivadas.
  - Herencia múltiple.
- **¿Polimorfismo?**

12 Polimorfismo

- Capacidad de los objetos de distintas clases de responder a mensajes con el mismo nombre.
- Ejemplo: + entre enteros y cadenas.
- En Python al no ser necesario especificar explícitamente el tipo de los parámetros que recibe una función, las funciones son naturalmente polimórficas.

[ ]:
```
print("hola " + "que tal.")
print(3 + 4)
```

13 ¿Podemos sumar dos jugadores?

- Podemos definir el método__add__

[ ]:
```
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}")
    def __add__(self, otro):
         return (f"Ambos jugadores son {self._nombre} y {otro._nombre}")
```

[ ]:
```
nico = Jugador('Nico Villalba', "Guild Esports")
faker = Jugador("Faker", "SK Telecom")
print(nico + faker)
```

14 En el ejemplo de e-sports de la clase pasada

[ ]:
```
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 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")
faker = JugadorDeLOL("Faker", "SK Telecom")

nico.jugar()
faker.jugar()
```

15 Observemos la modificación realizada

[ ]:
```
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.nombre}")

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

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

- ¿super()?

[ ]:
```
nico = JugadorDeFIFA('Nico Villalba', "Basilea")
faker = JugadorDeLOL("Faker", "SK Telecom")

nico.jugar()
faker.jugar()
```

16 super y herencia múltiple

- ¿Qué imprime?

[ ]:
```
class A():
    def __init__(self):
        print("Soy A")

class B():
    def __init__(self):
        print("Soy B")

class C(A, B):
    def __init__(self):
        print("Soy C")
        super().__init__()
```

[ ]:
```
obj = C()
```

17 Recordemos: MRO "Method Resolution Order"

[ ]:
```
C.__mro__
```

[ ]:
```
class A():

    def funcion1(self):
        return ("Hola")

class B():

    def funcion1(self):
        return ("Chau")

class C(A, B):

    def funcion1(self):
        return ("Hasta la vista!")
```

```
    def saludo(self):
        #x = self.funcion1()
        x = super().funcion1()
        print(x)
```

[ ]:
```
obj = C()
obj.saludo()
```

- Entonces, super() a qué clase base hace referencia?
- +Info

18 También podemos chequear..

[ ]:
```
"Es de clase A" if isinstance(obj, A) else "NO es A"
```

[ ]:
```
"Es subclase de A" if issubclass(C, A) else "NO es subclase de A"
```

19 Probamos en casa

¿Qué podemos decir de las variables de instancias cuyo nombre comienza con ___?

[ ]:
```
class A:
    def __init__(self, x, y, z):
        self.varX = x
        self._varY = y
        self.__varZ = z

    def demo(self):
        return f"ESTOY en A: x: {self.varX} -- y:{self._varY} --- z:{self.
 ↪__varZ}"

class B(A):
    def __init__(self):
        super().__init__("x", "y", "z")

    def demo(self):
        #return(super().demo())
        return f"ESTOY en B: x: {self.varX} -- y:{self._varY} --- z:{self.
 ↪__varZ}"

objB = B()
print(objB.demo())
```

20 Avancemos un poco más

21 getters y setters

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

    def getx(self):
        return self._x

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

    def delx(self):
        del self._x
```

- ¿Cuántas variables de instancia?
- Por cada variable de instancia no pública tenemos un método **get** y un método **set**.
- ¿Y **del**?
- TAREA: ¿qué dice la PEP 8 sobre esto?

22 Propiedades

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

[ ]:
```
class Demo:
    def __init__(self):
        self._x = 0
    def getx(self):
        print("estoy en get")
        return self._x
    def setx(self, value):
        print("estoy en set")
        self._x = value
    def delx(self):
        print("estoy en del")
        del self._x

    x = property(getx, setx, delx, "x es una propiedad")
```

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

23 La función property()

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

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

x = property(getx, setx, delx, "x es una propiedad")
```

- +Info

24 Más sobre property()

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

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

    def getx(self):
        return self._x

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

    def delx(self):
        del self._x

    x = property(getx)
```

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

25 ¿Y esto?

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

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

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

- @property es un **decorador**.

26 ¿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.

26.0.1 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.

27 Observemos el siguiente código

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

def decimos_chau(nombre):
    return f"Chau {nombre}!"

def saludo_a_Clau(saludo):
    return saludo("Clau")
```

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

28 ¿Qué podemos decir de este ejemplo?

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

[ ]:
```
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)
```

- ¿De qué tipo es saludo?

```
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?

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

29 Otra forma de escribir esto en Python:

[ ]:
```
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!")
```

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

30 Es equivalente a:

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

- +Ínfo
- +Info en español

31 Dijimos que @property es un decorador

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

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

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

- ATENCIÓN: x no es un método, es una propiedad.
- +Info

32 El ejemplo completo

[ ]:
```
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
    @x.deleter
    def x(self):
        del self._x
```

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

33 Herencia y propiedades

- Observemos este código: ¿qué imprime?, ¿qué significa?

[ ]:
```
class A:
    def __init__(self):
        self._x = 0
    @property
    def x(self):
        return self._x
    @x.setter
    def x(self, value):
        self._x = value

class B(A):
    def __init__(self):
        super().__init__()
```

```
obj = B()
obj.x = 10
print(obj.x)
```

34 INVESTIGAR: ¿qué son los métodos de clase?

- Uso del decorador **@classmethod**

35 Seguimos la próxima ...