Programador de Videojuegos

CLASE 20/08/25

Metodologías de programación

Programación Lineal

- El código se ejecuta de forma secuencial, instrucción tras instrucción, sin modularidad clara.
- Todo está en un único bloque (un programa largo).
- No hay reutilización de código (cada vez que se necesita algo, se vuelve a escribir).

Ventajas: Simple de entender en programas muy pequeños.
Desventajas: Difícil de mantener, leer y escalar.

Programación estructurada

- Evolución de la programación lineal.
- Propone organizar el código en bloques (funciones, procedimientos) para hacerlo más claro y mantenible.

Ventajas: Más ordenada y mantenible que la lineal.
Desventajas: Aunque modulariza, no representa directamente objetos del mundo real.

Programación Orientada a Objetos

- Organiza el programa en clases y objetos que representan entidades del mundo real.
- Cada entidad representa elementos del problema a resolver y tienen atributos y comportamiento.
- Facilita la reutilización y escalabilidad del código.

Ventajas: Ideal para proyectos grandes, permite modelar problemas complejos de manera natural.
Desventajas: Puede ser más complejo de aprender.

Programación Orientada a Objetos

Programación Orientada a Objetos

La POO es un paradigma de programación que organiza el código en torno a objetos que representan cosas del mundo real, con atributos (datos) y métodos (funciones que los manipulan).

Ventajas de la POO:

- Organización clara del código.
- Reutilización mediante herencia.
- Facilita el mantenimiento y escalabilidad.
- Representación natural del problema.

Conceptos fundamentales:

➔ Clase: Molde o plantilla para crear objetos.
➔ Objeto: Instancia de una clase.
➔ Atributos: Características que pertenecen a un objeto.
➔ Métodos: Funciones que pertenecen a una clase y actúan sobre los objetos.
➔ Constructor: Método especial para crear e inicializar un objeto (__init__).
➔ self: Referencia al objeto actual dentro de la clase.
➔ Encapsulamiento: Oculta datos internos y solo permite acceso mediante métodos.
➔ Herencia: Permite que una clase herede atributos y métodos de otra.
➔ Polimorfismo: Permite usar el mismo método con diferentes comportamientos.

Clase y Objeto

- **¿Qué es una clase?**

Es una plantilla o molde que define atributos y métodos comunes a un tipo de objeto. Entonces, definen de manera genérica cómo van a ser los objetos de determinado tipo, especificando qué atributos (datos) tendrá y qué métodos (funciones) podrá usar.

Sintaxis:
```
class NombreClase:
    # cuerpo de la clase
```
Los nombres de las clases se escriben camelCase.

Ejemplo:
```
class Persona:
```

- **¿Qué es un objeto?**

Es una instancia concreta de una clase, es decir, se crea a partir de la plantilla que es la clase. Este nuevo objeto existe en tiempo de ejecución, por consiguiente ocupa memoria y se puede utilizar en el programa.
Cada objeto tiene sus propios valores, aunque usen la misma clase.

Sintaxis:
```
objeto = NombreClase()
```

Ejemplo:
```
# Creamos un objeto de la clase Persona
persona1 = Persona()
```

Atributos

Son características o propiedades de los objetos.

Tipos de atributos:

- De instancia: Es particular de cada objeto.
- De clase: Pertenecen a la clase en general (compartidos por todos los objetos).

```python
class Persona:
    activo = True   # Atributo de clase (presente en todos los objetos perteneciente a esta clase)

    def __init__(self, nombre, edad):
        self.nombre = nombre   # Atributo de instancia
        self.edad = edad       # Atributo de instancia

# Instanciamos objetos de la clase Persona
p1 = Persona("Ana", 25)
p2 = Persona("Juan", 30)

#Mostramos los atributos de las instancias (objetos)
print(p1.nombre, p1.edad)   # Ana 25
print(p2.nombre, p2.edad)   # Juan 30

# Mostramos los atributos de la clase
print(p1.activo)  # True
print(p2.activo)  # True

#También podemos crear directamente atributos de instancia,
#mediante la notación clase.atributo = valor
p1.apellido = "Perez"
print(p1.apellido)
```

Atributos

Visibilidad de los atributos: Nivel de acceso que tiene un atributo (dato o propiedad) de una clase dentro y fuera de ella. Es decir, determina quién puede ver y modificar ese atributo: sólo la propia clase, la clase y sus hijas (herencia) o cualquier parte del programa.

Tipos de visibilidad:

Público

- Accesibles desde cualquier parte del programa.
- Es la visibilidad por defecto.
- En Python, se definen sin guiones bajos.

Protegidos

- Se suelen usar cuando el atributo es para uso interno pero aún accesible por herencia.
- Pueden ser accedidos y modificados por la clase y sus clases hijas.
- En Python, los atributos protegidos se escriben iniciando con un guion bajo: _atributo.

Privados

- Solo pueden ser accedidos y modificados por la clase, mediante métodos get y set.
- En Python, el nombre del atributo se define iniciando con doble guion bajo: __atributo.

```python
class Persona:
    #Constructor
    def __init__(self, nombre, edad, apellido):
        # Atributo público (el nombre inicia sin guión bajo): self.nombreAtributoPublico
        self.nombre = nombre
        # Atributo protegido (el nombre inicia con un guión bajo): self._nombreAtributoProtegido
        self._edad = edad
        # Atributo privado (el nombre inicia con dos guiones bajo): self.__nombreAtributoPrivado
        self.__apellido = apellido


# Objeto
p1 = Persona("Ana", 25, "Perez")
print(p1.nombre) #Accesible
print(p1._edad)  #Se puede acceder pero no es recomendable
print(p1._Persona__apellido)  # Se puede acceder usando "name mangling", pero no es recomendable
```

Programación Orientada a Objetos: Métodos

- **Métodos**

Funciones definidas dentro de una clase para operar sobre objetos.

Sintaxis:

```
def nombre_metodo(self, otros_parametros):
    # instrucciones
```

```python
class Jugador:
    #Constructor
    def __init__(self, nombre):
        self.nombre = nombre
        self.__vida = 100 #atributo privado

    #Get: obtener el valor del atributo privado
    def get_vida(self):
        return self.__vida
    #Set : modificar el valor del atributo privado
    def set_vida(self, nueva_vida):
        if nueva_vida >= 0:
            self.__vida = nueva_vida

    #Métodos adicionales
    def decrementar_vida(self, cantidad):
        self.__vida -= cantidad
        print(f"El jugador recibió {cantidad} de daño. Vida restante: {self.__vida}")
```

Programación Orientada a Objetos: Métodos

Métodos generalmente usados en una clase:

- **Constructor:** Método que se ejecuta automáticamente cuando creamos un objeto, sirve para inicializar un objeto.
- **Getters y setters:** Sirven para acceder (getter) o modificar (setter) atributos privados.
- **To String:** Método que devuelve la representación en texto de un objeto.

Constructor

- **Constructor __init__**

Se ejecuta al crear un objeto e inicializa los atributos.

Sintaxis:

```
def __init__(self, parametro1, parametro2):
    self.atributo1 = parametro1
    self.atributo2 = parametro2
```

Explicación:

- __init__ es la definición del constructor.
- self es la referencia al objeto actual. Este parámetro es obligatorio.
- parametro1, parametro2, parametroN son los datos que se almacenarán como atributos del objeto

Ejemplo:

```
class Persona:
    """
    Método constructor __init__
    Permite instanciar y asignar valores a los atributos. Como parámetros tiene:
    - self: hace referencia a la instancia perteneciente a la clase. Este parámetro es obligatorio.
    - atributos de la clase
    """
    def __init__(self, nombre, edad):
        self.nombre = nombre   # Atributo de instancia
        self.edad = edad       # Atributo de instancia

# Instanciamos objeto
# Al crear un objeto de la clase Persona se debe pasar los valores de nombre y edad
p1 = Persona("Ana", 25)
```

Constructor

En Python no existe la sobrecarga de constructores, es decir, no podemos tener un constructor vacío y uno que reciba parámetros.

Pero si se puede, crear un constructor con parámetros opcionales.

```python
class Persona:
    """
    En Python NO existe la sobrecarga de constructores de manera directa,
    porque solo puede haber un __init__.
    """
    def __init__(self):  # primer constructor
        print("Constructor vacío")

    def __init__(self, nombre):  # segundo constructor: pisa al anterior
        self.nombre = nombre

p = Persona("Ana")   # funciona
# p = Persona()      # error, porque ya no existe el vacío

print(p.nombre)
```

```python
class Persona:
    #Constructor con parámetros opcionales
    # teniendo None como valores por defecto
    def __init__(self, nombre=None, edad=None):
        self.nombre = nombre
        self.edad = edad

# Crear con constructor vacío
p1 = Persona()
print(p1.nombre, p1.edad)  # None None

# Crear con atributos
p2 = Persona("Ana", 25)
print(p2.nombre, p2.edad)  # Ana 25
```

Getters y Setters

Atributos privados:
Recordemos que se definen con doble guion bajo __.
Sintaxis dentro del constructor:

self.__atributo = valor

Ejemplo:

```
class Cuenta:
    def __init__(self, saldo):
        self.__saldo = saldo
```

¿Cómo se acceden o modifican los atributos privados?
Los getters y setters son métodos especiales que nos permiten acceder y modificar atributos privados de un objeto, respetando el principio de encapsulamiento.
¿Para qué se usan?

- Para proteger los datos internos del objeto.
- Para validar los cambios antes de permitirlos.
- Para tener un control centralizado del acceso a los datos

Sintaxis:

```
def get_atributo(self):
    return self.__atributo

def set_atributo(self, valor):
    self.__atributo = valor
```

```python
class Jugador:
    #Constructor
    def __init__(self, nombre):
        self.nombre = nombre
        self.__vida = 100 #atributo privado

    #Get: obtener el valor del atributo privado
    def get_vida(self):
        return self.__vida
    #Set : modificar el valor del atributo privado
    def set_vida(self, nueva_vida):
        if nueva_vida >= 0:
            self.__vida = nueva_vida
```

```python
# Utilización
jugador1 = Jugador("Rocío")
print(jugador1.nombre)  # Rocío
print(jugador1.get_vida())      # 100
jugador1.set_vida(80)           # cambia la vida a 80
print(jugador1.get_vida())      # 80
```

Programación Orientada a Objetos: Atributos

```python
class CuentaBancaria:
    #Constructor
    def __init__(self, titular, saldo):
        self.titular = titular
        self.__saldo = saldo  # atributo privado (doble guion bajo)

    #Get y set
    def get_saldo(self):
        return self.__saldo

    def set_saldo(self, nuevo_saldo):
        if nuevo_saldo >= 0:
            self.__saldo = nuevo_saldo

    #Métodos adicionales
    def mostrar_saldo(self):
        print(f"Saldo: ${self.__saldo}")

    def depositar(self, monto):
        self.__saldo += monto

    def retirar(self, monto):
        if monto <= self.__saldo:
            self.__saldo -= monto
        else:
            print("Fondos insuficientes")
```

```python
cuenta = CuentaBancaria("Rocío", 1000)
cuenta.mostrar_saldo()  # Saldo: $1000
cuenta.depositar(500)
cuenta.retirar(300)
cuenta.mostrar_saldo()  # Saldo: $1200
```

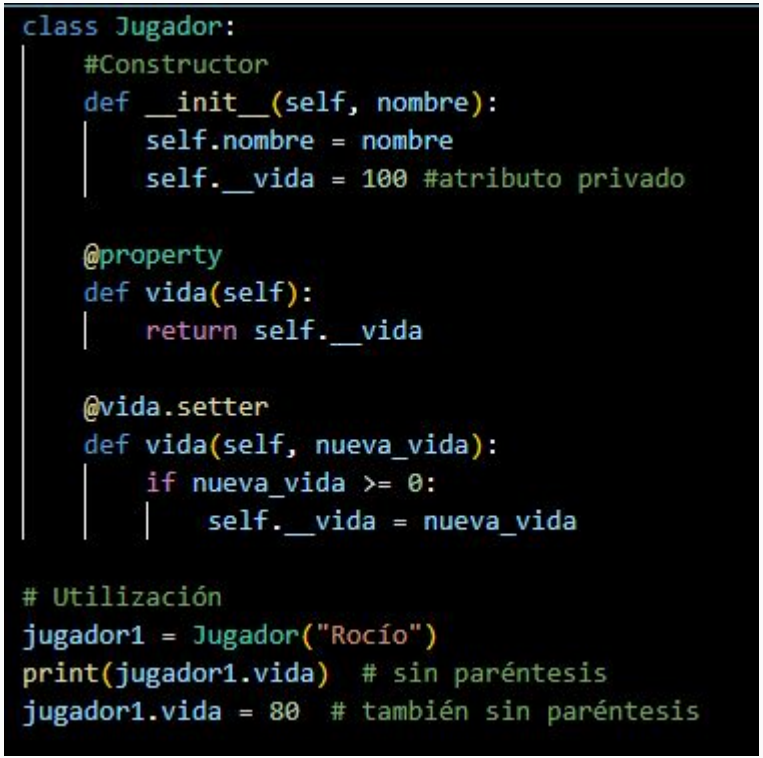
Atributos privados - Uso de propiedades (@property y @<atributo>.setter)

¿Qué es @property?
Es un decorador en Python que convierte un método en un atributo de solo lectura. Nos permite acceder a métodos como si fueran atributos, sin necesidad de escribir paréntesis ().

- Ventajas de usar @property
  - Hace el código más limpio y legible.
  - Permite controlar el acceso a atributos privados sin cambiar la forma de accederlos.
  - Ideal para cuando querés validar, calcular o formatear datos al ser accedidos o modificados.

```
class Jugador:
    def __init__(self, nombre):
        self.nombre = nombre
        self.__vida = 100

    #Getter con property
    @property
    def vida(self):
        return self.__vida
    #Setter con @vida.setter
    @vida.setter
    def vida(self, nueva_vida):
        if nueva_vida >= 0:
            self.__vida = nueva_vida
```



```python
class Jugador:
    #Constructor
    def __init__(self, nombre):
        self.nombre = nombre
        self.__vida = 100 #atributo privado

    @property
    def vida(self):
        return self.__vida

    @vida.setter
    def vida(self, nueva_vida):
        if nueva_vida >= 0:
            self.__vida = nueva_vida

# Utilización
jugador1 = Jugador("Rocío")
print(jugador1.vida)  # sin paréntesis
jugador1.vida = 80  # también sin paréntesis
```

To String

__str__ Método que devuelve la representación de un objeto en texto.

```
class Persona:
    def __init__(self, nombre, edad):
        self.nombre = nombre
        self.edad = edad

    def __str__(self):
        return f"{self.nombre} ({self.edad} años)"

p = Persona("Ana", 25)
print(p)
# Si no estuviera el método __str__ se mostraría de la siguiente forma el objeto
# <__main__.Persona object at 0x...>  (no hay __str__)
```

Ejercitación

1) Crear una clase Personaje con los siguientes atributos:

nombre (atributo público de la instancia, obligatorio)
nivel (atributo público de instancia, opcional, default 1)
vida (atributo privado de instancia, opcional, default 100)
experiencia (atributo privado de instancia, opcional, default 0)
es_humano (atributo de clase, True)

Implementar métodos:

- constructor: Tener en cuenta que se pueda crear objetos solo con el nombre o con todos los parámetros.
- getters y setters en caso de corresponder. Para el atributo vida, en caso de adicionarle el método set, que modifique la vida pero el valor de la misma no puede ser negativa.
- to string que retorne: "Nombre (Nivel X) - Vida: Y"
- Adicionar el método:
  - subir_nivel() el cual incrementa nivel en 1 y suma 10 a experiencia
  - estado() que devuelva "vivo" si la vida es mayor a 0 o "derrotado" en caso contrario.
  - recibir_danio(valor) el cual resta el valor de la vida y devuelve el estado actual.
- Crear objetos para probar todos los métodos.

2) Generar la misma clase pero que los atributos sean de tipo privado y agregar propiedades con @property y @setter en vez de métodos get/set tradicionales.