Resumen de clases en python

1. Clases y objetos

En ambos lenguajes, una **clase** es un molde o plantilla que define los **atributos** (datos) y **métodos** (funciones) que tendrán los objetos.

En Java:

```java
public class Persona {
    private String nombre;
    private int edad;

    public Persona(String nombre, int edad) {
        this.nombre = nombre;
        this.edad = edad;
    }

    public void saludar() {
        System.out.println("Hola, soy " + nombre);
    }
}
```

En Python:

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

    def saludar(self):
        print("Hola, soy", self.__nombre)
```

Equivalencia:

- __init__ en Python = constructor (Persona(...))
- self en Python = this en Java
- __nombre = atributo “privado” (convención)
- Los métodos públicos se definen igual, sin decorador especial.

2. Encapsulación y visibilidad

En Java tenemos modificadores (public, private, protected), pero **Python no los aplica de forma estricta**.
Sin embargo, **podemos seguir la misma disciplina** con las siguientes convenciones:

Nivel de acceso	Java	Python (convención)	Descripción
Público	public	nombre	Accesible desde cualquier parte
Protegido	protected	_nombre	Solo debe usarse dentro de la clase o subclases
Privado	private	__nombre	No accesible directamente (usa *name mangling*)

Ejemplo en Python siguiendo la estructura de Java:

```python
class CuentaBancaria:
    def __init__(self, saldo):
        self.__saldo = saldo   # privado

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

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

    def get_saldo(self):
        return self.__saldo

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

2.1 Name Mangling en Python

En Python, el "name mangling" es un mecanismo que modifica internamente el nombre de los atributos que comienzan con dos guiones bajos (__). Su objetivo principal es **evitar colisiones de nombres** en clases con herencia y simular cierto nivel de encapsulación.

¿Qué hace exactamente?

Cuando declaras un atributo así:

self.__oculto = 10

Python cambia internamente su nombre a:

self._NombreDeLaClase__oculto

¿Por qué existe?

Para que si una subclase define otro atributo llamado __oculto, **no sobrescriba accidentalmente** el de la clase padre.

Ejemplo completo

```
class Ejemplo:
    def __init__(self):
        self.__oculto = 10

    def mostrar(self):
        print(self.__oculto)

obj = Ejemplo()
obj.mostrar()      # ✓ Funciona
print(obj.__oculto)  # ✗ Error: el atributo ha sido renombrado

# Pero esto sí funciona (porque es el nombre "mangleado")
print(obj._Ejemplo__oculto)  # ✓ Imprime 10
```

3. Métodos y su tipo

Python distingue tres tipos de métodos, que podemos relacionar con Java:

Tipo de método	Decorador	Primer parámetro	Qué representa	Equivalente en Java
De instancia	*(sin decorador)*	`self`	la **instancia** concreta del objeto	método normal
De clase	`@classmethod`	`cls`	la **clase**, no el objeto	método `static` que accede a atributos de clase
Estático	`@staticmethod`	*(ninguno)*	no accede ni a la clase ni al objeto	método `static`

3.1. Método de instancia, el normal y el que se usará en un 90% de las veces

No lleva ningún decorador.

- No lleva decorador.
- Su primer parámetro es **self**.
- Self representa **el objeto concreto** (como this en Java).

Se usa para trabajar con los **atributos del objeto**. Equivale a los métodos no static en Java.

```python
class Persona:
    def __init__(self, nombre):
        self.nombre = nombre  # atributo de instancia

    def saludar(self):  # método de instancia
        print(f"Hola, soy {self.nombre}")

# Uso
p = Persona("Lucía")
p.saludar()  # ✅ "Hola, soy Lucía"
```

No necesita decorador.
Python entiende automáticamente que `saludar()` pertenece a la instancia.

3.2 Método de clase (@classmethod)

Lleva el decorador @classmethod porque **no trabaja con una instancia concreta**, sino con la **clase en sí** (su estructura y atributos comunes).

Es un método que pertenece a **la clase**, no a cada objeto.

- Lleva el decorador **@classmethod**.
- Su primer parámetro es **cls** (class).
- Sirve para acceder a **atributos de clase** (compartidos entre objetos).

```python
class Persona:
    total = 0  # atributo de clase

    def __init__(self, nombre):
        self.nombre = nombre
        Persona.total += 1

    @classmethod
    def numero_de_personas(cls):
        print(f"Se han creado {cls.total} personas")
```

Sin el @classmethod, el primer parámetro sería self, y no podrías acceder a los atributos de clase (cls.total).

3.3.Método estático (@staticmethod)

Lleva @staticmethod cuando **no necesitas ni la instancia ni la clase**, es decir, es una función auxiliar que simplemente está *dentro de la clase por organización*.

Es simplemente **una función metida dentro de la clase** por organización.

- Lleva **@staticmethod**.
- No recibe ni self ni cls.
- No usa nada del objeto ni de la clase.

```
class Matematicas:
    @staticmethod
    def sumar(a, b):
        return a + b

print(Matematicas.sumar(3, 5))  # ✅ 8
```

En Java es igual a un **método static que no usa nada de la clase**.

Algunos ejemplos:

```
class Matematicas:
    @staticmethod
    def sumar (a,b):
        return a+b

m=Matematicas()
print (m.sumar(3,4))  #SI FUNCIONARÍA
-------------------------------------
class Matematicas:
    #@staticmethod
    def sumar (a,b):
        return a+b

m=Matematicas()

print (m.sumar(3,4))  #NO FUNCIONARIA AL COMENTAR EL DECORADOR, PUES
LLAMAMOS AL MÉTOD DESDE UNA INSTANCIA
```

¿Por qué?
Porque al llamar así:

m.sumar(3,4)

Python pasa implícitamente:

self = m
a = 3
b = 4

→ Es decir, **3 argumentos**.
Pero tu método solo acepta **dos**, así que falla.

Conclusión importante

Si vas a llamar al método solo desde la clase → funciona aunque quites staticmethod.

Si vas a permitir usarlo desde instancias → NECESITAS @staticmethod.

```
------------------------------------------------------------
class Matematicas:
    #@staticmethod
    def sumar (a,b):
        return a+b

print (Matematicas.sumar(3,4))  #SÍ FUNCIONA PUES LLAMAMOS AL MÉTTODO
DESDE LA CLASE, SIN INSTANCIAR EL OBJETO.
```

Regla fácil:

En Python, **si el método usa self, no pongas decorador.**

Si el método **usa la clase (cls) o no usa ninguno de los dos**, entonces **sí debes indicarlo** con @classmethod o @staticmethod.

Significado de cls

En Python, **cls es al método de clase lo que self es al método de instancia**.

- self → representa **la instancia concreta** (el objeto creado con la clase).
- cls → representa **la propia clase** (el molde del que salen los objetos).

Imagina que defines una clase Persona:

```python
class Persona:
    total = 0  # atributo de clase

    def __init__(self, nombre):
        self.nombre = nombre  # atributo de instancia
        Persona.total += 1

    @classmethod
    def contar_personas(cls):
        print(f"Se han creado {cls.total} personas")
```

```python
p1 = Persona("Lucía")
p2 = Persona("Carlos")

p1.contar_personas()
Persona.contar_personas()
```

Salida:

```
Se han creado 2 personas
Se han creado 2 personas
```

EJEMPLO COMPLETO

```python
class Persona:
    total = 0     # atributo de clase

    def __init__(self, nombre):
        self.nombre = nombre   # atributo de instancia
        Persona.total += 1

    def saludar(self):          # usa self → método de instancia
        print(f"Hola, soy {self.nombre}")

    @classmethod                # usa cls → método de clase
    def contador(cls):
        return cls.total

    @staticmethod               # no usa nada → método estático
    def es_adulto(edad):
        return edad >= 18
```

3.4 Comparativa

En Java	En Python
método normal	método normal con `self`
método `static`	`@staticmethod`
método de clase (no existe igual)	`@classmethod`
constructor	`__init__`
this	self

4.Herencia

Python implementa la herencia **de forma muy parecida a Java**, solo que no se declara el tipo explícitamente.

En Java:

```java
public class Estudiante extends Persona {
    private String universidad;

    public Estudiante(String nombre, String universidad) {
        super(nombre);
        this.universidad = universidad;
    }

    @Override
    public void saludar() {
        System.out.println("Hola, soy " + nombre + " y estudio en " + universidad);
    }
}
```

En Python:

```
class Estudiante(Persona):
    def __init__(self, nombre, universidad):
        super().__init__(nombre)        # llama al constructor de la superclase
        self.__universidad = universidad

    def saludar(self):                  # sobrescribe el método
        print(f"Hola, soy {self.nombre} y estudio en {self.__universidad}")
```

Equivalencias:

- extends → (ClasePadre)
- super().__init__() → super(...) en Java
- Redefinición de métodos → sobrescritura con el mismo nombre

5. Herencia múltiple

A diferencia de Java, Python **sí permite herencia múltiple**, aunque se usa con precaución:

```
class Volador:
    def volar(self):
        print("Estoy volando")

class Animal:
    def comer(self):
        print("Comiendo...")

class Pato(Animal, Volador):
    pass
```

```
mi_pato = Pato()
```

Puede usar métodos de **Animal** y **Volador**.

Cuando varias clases aportan métodos o atributos mezclados, por eso hay que usarlo con precaución:

- cuesta entender quién aporta qué,
- cuesta ver el orden de ejecución,
- cuesta depurar errores.

Esto no pasa en Java, porque **Java NO permite herencia múltiple**

NOTA: pass se usa para indicar que no haga nada, pues en python no se puede dejar un bloque vacío.