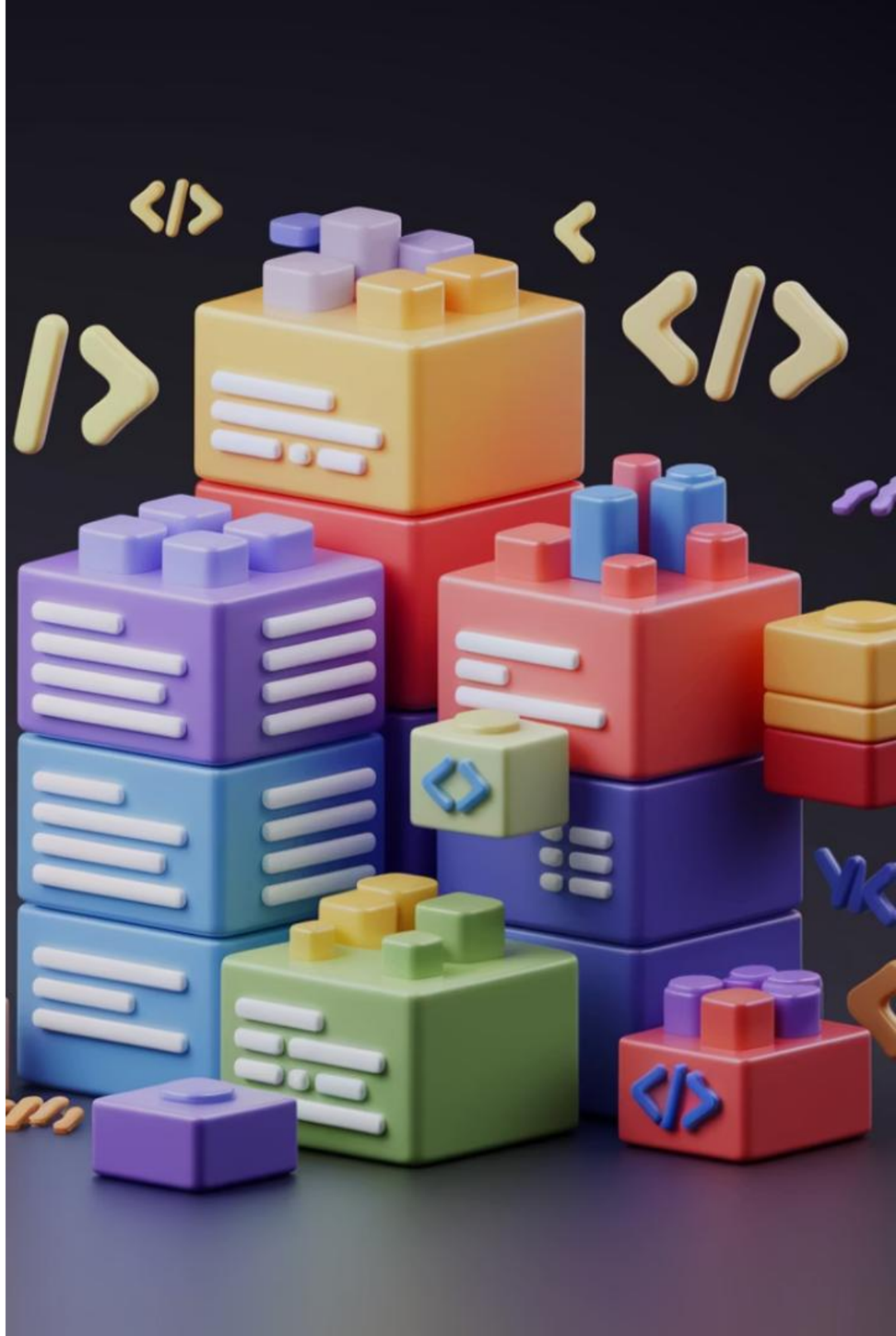
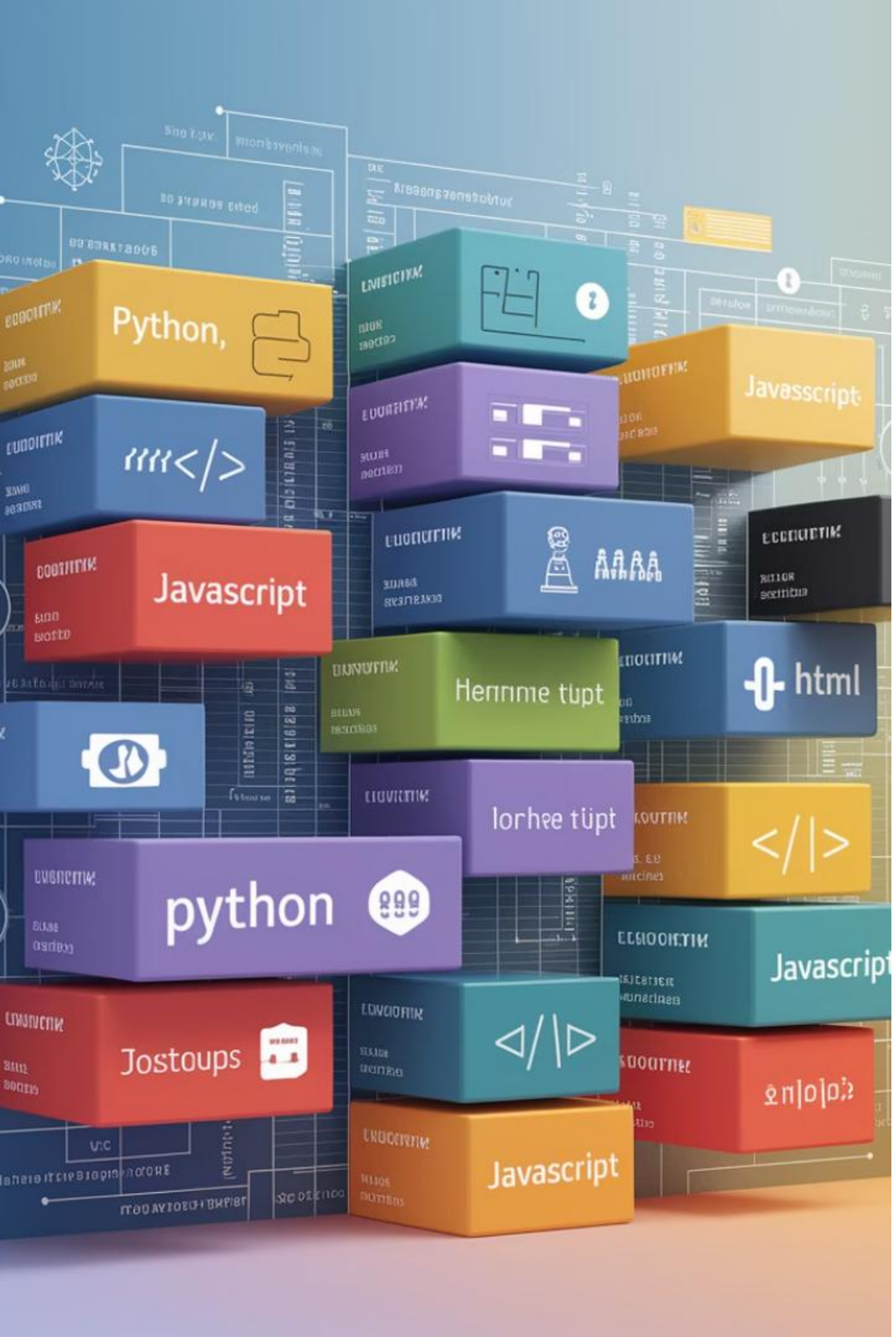
Introducción a la Programación Orientada a Objetos (POO)

La Programación Orientada a Objetos (POO) es un paradigma que permite modelar el mundo real utilizando "objetos". Cada objeto combina datos (atributos) y funciones (métodos) en una estructura llamada clase.

Este enfoque ayuda a organizar mejor el código, promueve la reutilización y facilita el mantenimiento de aplicaciones complejas. A diferencia de la programación estructurada, donde se priorizan las funciones, en la POO se trabaja pensando en entidades que interactúan entre sí.

por Kibernum Capacitación





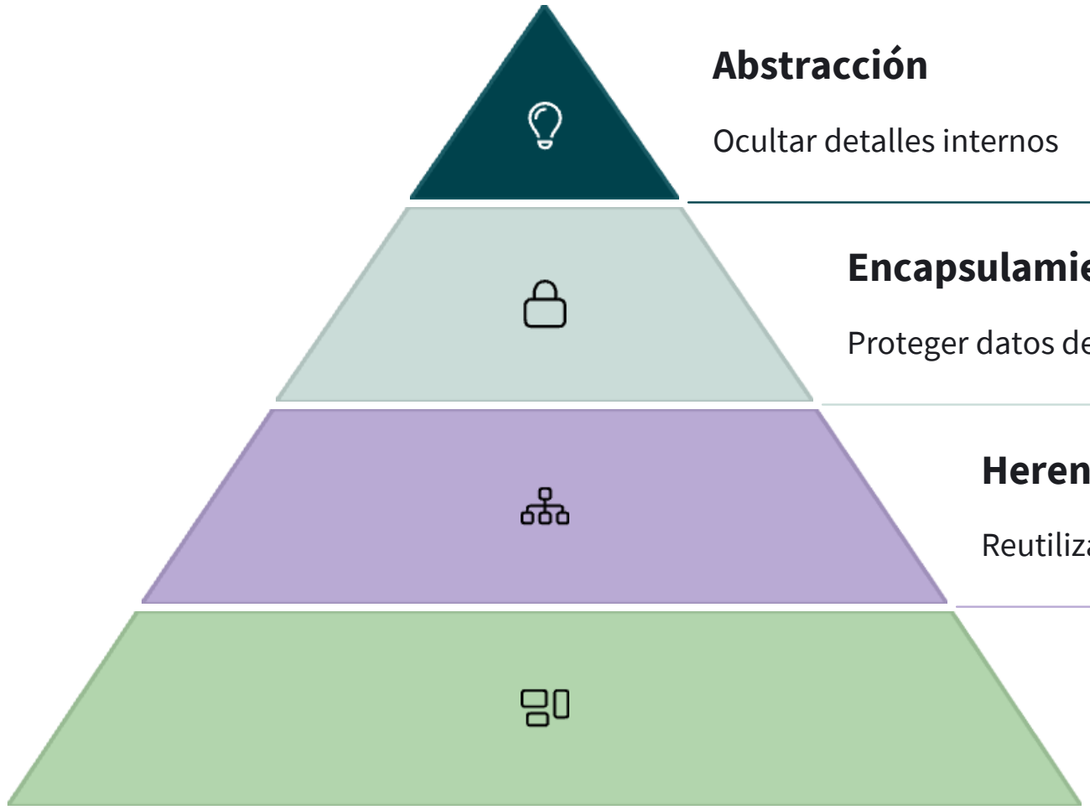
Preguntas de Activación de Contenido

¿Alguna vez has trabajado con clases o estructuras similares en otro lenguaje de programación (como Java, C#, etc.)?

¿Qué diferencias crees que existen entre una función y un método dentro de un programa?

¿Cómo crees que se podría representar un objeto del mundo real (como un auto, un usuario o una cuenta bancaria) dentro de un programa?

Principios Fundamentales de la POO



Abstracción

Ocultar detalles internos

Encapsulamiento

Proteger datos del objeto

Herencia

Reutilizar código entre clases

Polimorfismo

Comportamientos distintos según contexto

La POO se basa en estos cuatro principios fundamentales que permiten crear sistemas modulares y mantenibles. La abstracción oculta los detalles internos mostrando solo lo necesario, como un botón de encendido en un automóvil sin exponer el funcionamiento del motor.

El encapsulamiento protege los datos permitiendo acceso solo mediante métodos definidos, mientras que la herencia permite que una clase herede atributos y métodos de otra. El polimorfismo permite que métodos con el mismo nombre se comporten de manera distinta según el contexto.

Clases y Objetos: Conceptos Básicos

¿Qué es una clase?

Una clase es un molde o plantilla que define atributos y comportamientos comunes a un tipo de objeto. Funciona como un plano que establece las características que tendrán todos los objetos creados a partir de ella.

¿Qué es un objeto?

Un objeto es una instancia concreta de una clase. Representa una entidad específica con sus propios valores para los atributos definidos en la clase. Cada objeto puede comportarse de manera independiente siguiendo los métodos de su clase.

```
class Persona:
    pass  # Clase vacía por ahora
```

```
persona1 = Persona()
```

Las clases y objetos son los elementos fundamentales de la POO. Mientras que la clase define la estructura, los objetos son las manifestaciones reales de esa estructura con datos específicos. Esta relación permite modelar entidades del mundo real de forma intuitiva y organizada.

Definición de Clases en Python

Constructor (__init__)

Se llama automáticamente al crear un objeto. Inicializa los atributos con valores específicos.

Parámetro self

Se refiere al objeto actual. Es obligatorio como primer parámetro en todos los métodos de instancia.

Atributos

Variables que almacenan datos dentro de la clase (como nombre y edad).

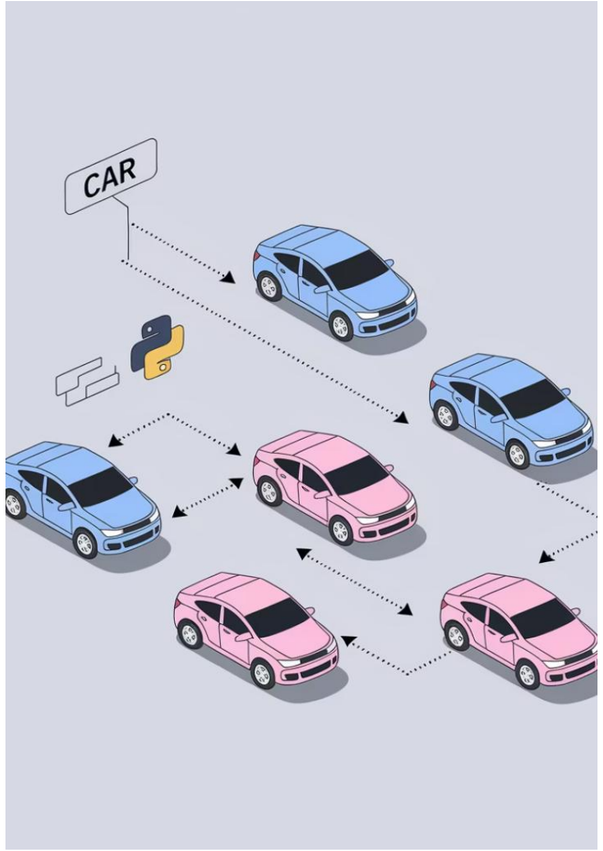
Métodos

Funciones definidas dentro de la clase que determinan el comportamiento de los objetos.

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

    def saludar(self):
        print(f"Hola, soy {self.nombre} y tengo {self.edad} años.")
```

En Python, una clase se define con la palabra clave "class" seguida del nombre de la clase. El constructor __init__ es un método especial que se ejecuta cuando se crea una nueva instancia, permitiendo inicializar los atributos del objeto con valores específicos.




```
persona1 = Persona("Ana", 30)
persona2 = Persona("Luis", 25)

persona1.saludar()  # Hola, soy Ana y tengo 30 años.
persona2.saludar()  # Hola, soy Luis y tengo 25 años.
```

Instanciación de Objetos

Definir la clase

Crear la plantilla con atributos y métodos que tendrán los objetos.

Crear la instancia

Llamar a la clase como si fuera una función, pasando los argumentos necesarios para el constructor.

Utilizar el objeto

Acceder a sus atributos y llamar a sus métodos mediante la notación de punto.

La instanciación es el proceso de crear objetos a partir de una clase. Cuando instanciamos un objeto, estamos creando una entidad única con su propio espacio de memoria, que sigue el comportamiento definido en su clase pero mantiene su propio estado independiente.

En Python, esto se realiza simplemente llamando a la clase como si fuera una función, pasando los argumentos que requiera el constructor.

Encapsulamiento y Visibilidad

Notación	Significado	Ejemplo
variable	Pública (accesible desde fuera)	nombre
_variable	Protegida (acceso interno sugerido)	_edad
__variable	Privada (nombre "enmangado")	__saldo

Python no tiene modificadores de acceso como otros lenguajes, pero utiliza convenciones de nomenclatura para indicar la visibilidad de atributos y métodos. Los atributos privados (con doble guion bajo) no son realmente inaccesibles, pero Python modifica su nombre internamente para dificultar el acceso directo desde fuera de la clase.

Esta convención permite implementar el encapsulamiento, protegiendo los datos internos de modificaciones no controladas y exponiendo solo las interfaces necesarias.

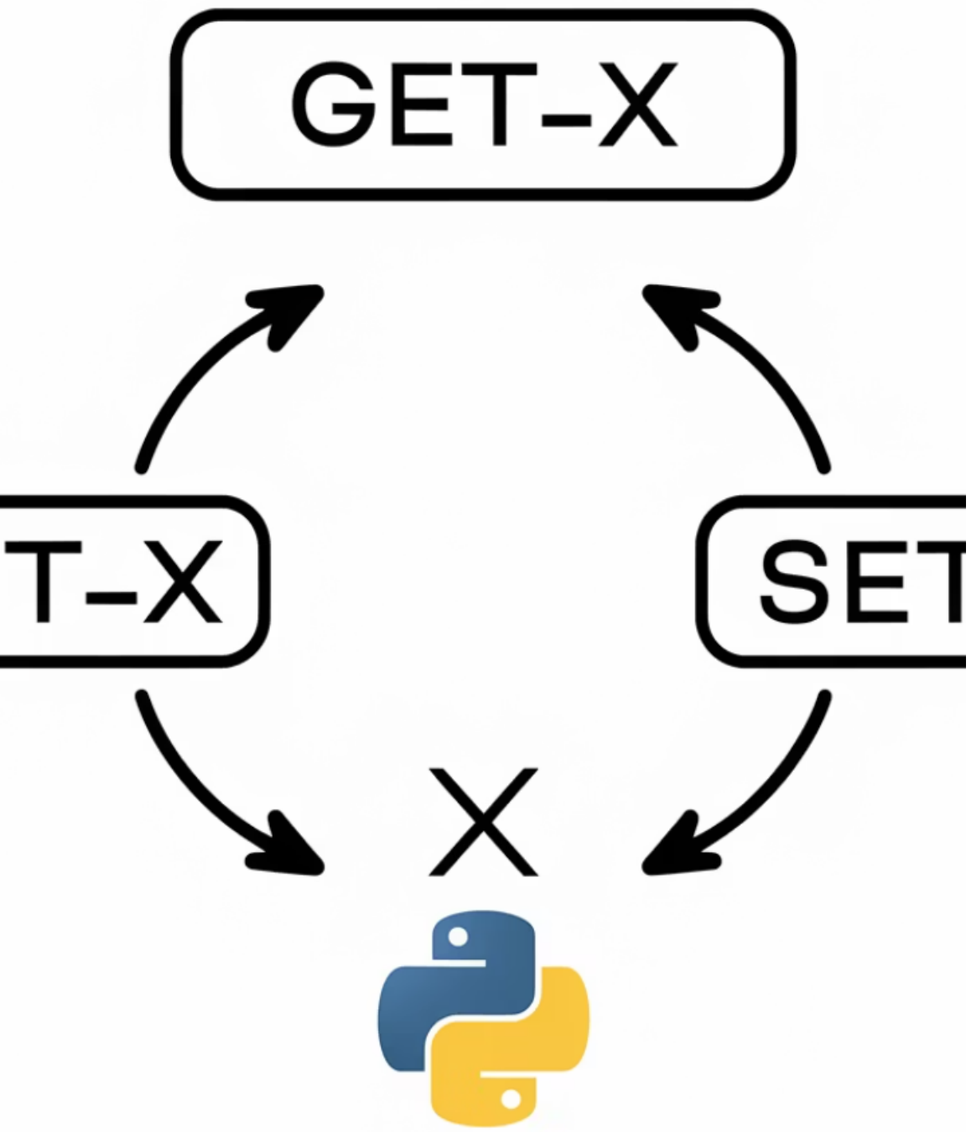
Atributos y Métodos Privados

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

    def ver_saldo(self):
        return self.__saldo

    def depositar(self, monto):
        if monto > 0:
            self.__saldo += monto
```

No se puede acceder directamente a __saldo desde fuera de la clase.



Getters y Setters Tradicionales

Definir atributos privados

Crear variables con prefijo __ para proteger los datos.

Implementar métodos getter

Crear funciones que devuelvan el valor del atributo privado.

Implementar métodos setter

Crear funciones que modifiquen el valor del atributo con validaciones.

Utilizar los métodos

Acceder y modificar atributos a través de los métodos definidos.

Los getters y setters son métodos que permiten acceder y modificar atributos privados de forma controlada. Proporcionan una capa de abstracción que oculta los detalles de implementación y permite validar los datos antes de asignarlos, manteniendo la integridad del objeto.

Ejemplo Utilizando Getters y Setters

```python
class Producto:
    def __init__(self, nombre, precio):
        self.__nombre = nombre
        self.__precio = precio

    def get_precio(self):
        return self.__precio

    def set_precio(self, nuevo_precio):
        if nuevo_precio > 0:
            self.__precio = nuevo_precio
```

```python
p = Producto("Laptop", 1200)
print(p.get_precio())
p.set_precio(1500)
```

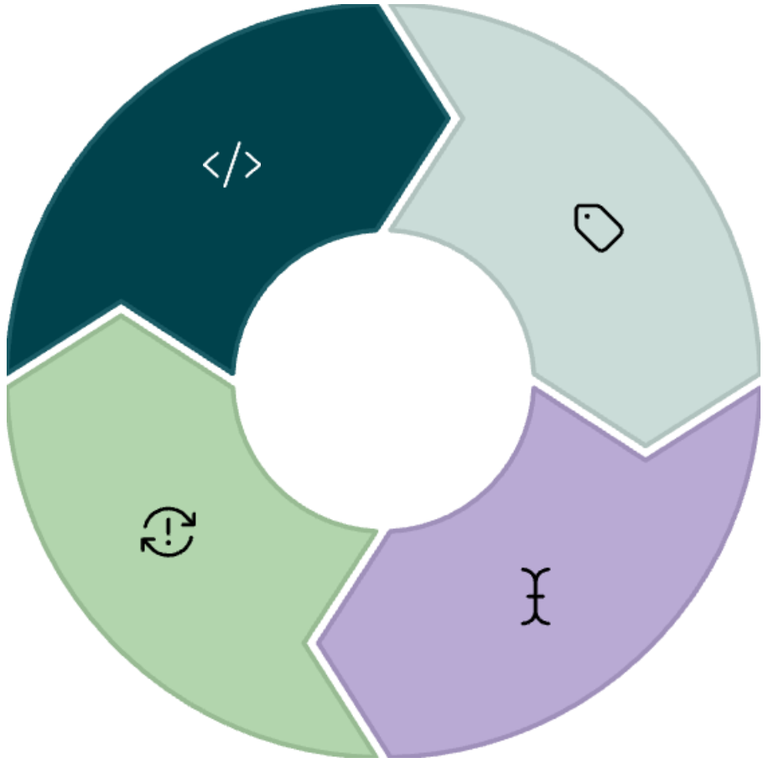
Decoradores @property y @setter

Definir atributo protegido

Crear variable con prefijo _ para indicar acceso restringido

Crear método @property

Permite acceder al atributo como propiedad



Usar como atributo normal

Acceder y modificar usando notación de punto

Implementar método @setter

Valida y modifica el valor del atributo

Los decoradores @property y @setter proporcionan una sintaxis más elegante y pythónica para implementar getters y setters. Permiten acceder y modificar atributos protegidos como si fueran públicos, pero manteniendo el control sobre las operaciones que se realizan.

Esta aproximación hace que el código sea más limpio y natural, ocultando la complejidad de la validación y el acceso controlado.

Ejemplo Uso de Decoradores

```python
class Producto:
    def __init__(self, nombre, precio):
        self._nombre = nombre
        self._precio = precio

    @property
    def precio(self):
        return self._precio

    @precio.setter
    def precio(self, nuevo_precio):
        if nuevo_precio > 0:
            self._precio = nuevo_precio
```

```python
p = Producto("Tablet", 300)
print(p.precio)      # Accede como atributo
p.precio = 400       # Modifica como atributo
```

Actividad Práctica Guiada: Modelando Clases en Python

Objetivo de la Actividad:

Aplicar los principios de la Programación Orientada a Objetos (POO) en Python mediante la creación guiada de clases, atributos, métodos, encapsulamiento y uso de decoradores @property, desarrollando un sistema básico de gestión de productos.

Contexto de la Actividad:

Imagina que estás desarrollando un pequeño sistema para registrar productos de una tienda. Cada producto debe tener un nombre, un precio y debe permitir modificar el precio bajo ciertas condiciones. Aplicarás los conceptos de **clases**, **constructores**, **atributos**, **métodos**, **encapsulamiento** y **getters/setters** usando @property.

Actividad Práctica Guiada: Modelando Clases en Python

Paso a Paso de la Actividad:

Paso 1: Crear la clase Producto

1.Abre tu editor de código (recomendado: VS Code o Jupyter Notebook).

2.Crea un archivo llamado producto.py.

3.Escribe el siguiente código:

```
class Producto:
    def __init__(self, nombre, precio):
        self._nombre = nombre
        self._precio = precio
```

Explicación: El constructor inicializa el objeto con nombre y precio. Los atributos son **protegidos** por convención (_nombre, _precio).

Actividad Práctica Guiada: Modelando Clases en Python

Paso a Paso de la Actividad:

Paso 2: Definir método para mostrar información

```
    def mostrar_info(self):
        print(f"Producto: {self._nombre} - Precio: ${self._precio}")
```

Explicación: Este método imprime los detalles del producto usando print() y accede a los atributos internos.

Actividad Práctica Guiada: Modelando Clases en Python

Paso a Paso de la Actividad:

Paso 3: Implementar decoradores @property y @setter

```python
    @property
    def precio(self):
        return self._precio

    @precio.setter
    def precio(self, nuevo_precio):
        if nuevo_precio > 0:
            self._precio = nuevo_precio
        else:
            print("El precio debe ser mayor que cero.")
```

Explicación: @property permite acceder al precio como si fuera un atributo, y el setter controla que no se asignen valores inválidos.

Actividad Práctica Guiada: Modelando Clases en Python

Paso a Paso de la Actividad:

Paso 4: Crear un objeto y probar los métodos

```
producto1 = Producto("Teclado", 15000)
producto1.mostrar_info()

producto1.precio = 18000
producto1.mostrar_info()

producto1.precio = -500  # Esto debe mostrar un mensaje de error
```

Explicación: Se crea una instancia de Producto, se muestra su información y se prueba la validación del setter.

Resultado Esperado:

- Mostrar la información del producto con el precio original.
- Actualizar correctamente el precio si es válido.
- Impedir la asignación de un precio inválido.

Desafío Final Caso de Uso para Implementación Autónoma

Enunciado:

Imagina que estás desarrollando una aplicación para una biblioteca digital. Cada libro que se registra debe contener el **título**, el **autor** y el **número de páginas**. Además, debe existir la posibilidad de **modificar la cantidad de páginas** solamente si el nuevo número es mayor a cero.

Desafío: crea una clase llamada Libro que implemente los conceptos de Programación Orientada a Objetos vistos en la actividad anterior.

Requisitos del ejercicio:

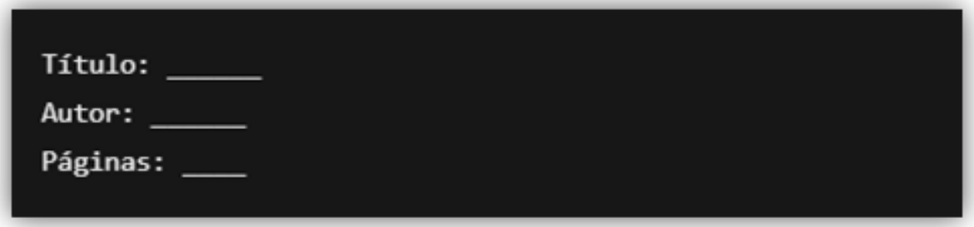
1.Crea la clase Libro con atributos protegidos para:

- •Título del libro (_titulo)
- •Autor (_autor)
- •Número de páginas (_paginas)

Desafío Final Caso de Uso para Implementación Autónoma

2.Implementa un **constructor** que permita inicializar los tres atributos al crear un objeto.

3.Crea un método mostrar_info() que imprima en consola los datos del libro en formato:



4.Aplica **encapsulamiento** sobre el atributo _paginas mediante decoradores @property y @setter, validando que el nuevo valor sea mayor que cero.

5.Crea una instancia de la clase y:

- Muestra la información original.
- Modifica el número de páginas con un valor válido.
- Intenta modificar el número de páginas con un valor inválido (negativo o cero).

Material Complementario

https://www.youtube.com/watch?v=Qc0qD00RmRA

📽️ **“POO en Python - Aprende desde cero”** – Este video explica paso a paso los conceptos fundamentales de la Programación Orientada a Objetos en Python, con ejemplos prácticos de clases, herencia, encapsulamiento y uso de decoradores.

Preguntas de Reflexión Final

¿Qué ventaja te ofrece definir tus propios objetos y clases frente a usar únicamente funciones?

¿Cómo ayuda el encapsulamiento a proteger los datos en tus programas?

¿Cómo entendiste el uso de decoradores como @property y cómo mejorarían el control sobre los atributos?