POO en Python: 🐣 De Principiante 🚀 a Héroe 🦸

Introducción

La Programación Orientada a Objetos (POO) es una manera de pensar y organizar el código que resulta muy útil cuando se quiere crear software complejo. En lugar de escribir todo en un solo bloque de código, la POO permite dividir el programa en pequeñas "cajas" llamadas **objetos** que contienen tanto datos como funciones para manipular esos datos.

Conceptos Básicos de POO

Objetos

Un **objeto** es algo que representa una cosa en tu programa, y puede hacer cosas (métodos) o tener datos (atributos). Los objetos son creados a partir de **clases**.

Clases

Una **clase** es un plano o plantilla para crear objetos. Es como una receta que te dice qué ingredientes necesitas (atributos) y cómo prepararlos (métodos).

Atributos

Los **atributos** son como las variables que pertenecen a un objeto. Por ejemplo, un objeto "Coche" podría tener atributos como "color", "marca" y "modelo".

Métodos

Los **métodos** son funciones que pertenecen a un objeto. Estos métodos pueden hacer cosas, como encender un coche o acelerar. Los métodos se llaman usando la notación de punto: `miObjeto.miMetodo()`.

Creación y Uso de Objetos

Creando Instancias

Para crear un objeto a partir de una clase, se utiliza la palabra clave `__init__` (constructor) que se llama automáticamente cuando se crea un objeto. En Python, no se utiliza `new`, simplemente se llama a la clase. Cada objeto creado es una instancia única de la clase.

Accediendo a Atributos

Se acceden a los atributos de un objeto utilizando la notación de punto. Por ejemplo, `objeto.atributo`.

Invocando Métodos

Se invocan métodos de un objeto utilizando la notación de punto seguida del nombre del método y argumentos entre paréntesis. Por ejemplo, `objeto.metodo(argumento1, argumento2)`.

Encapsulación

La encapsulación es el mecanismo para ocultar la implementación interna de una clase y solo exponer los métodos y atributos necesarios para su uso. Promueve la modularidad y protege los datos internos de modificaciones accidentales.

Niveles de Acceso

Python no tiene modificadores de acceso estrictos como público, privado y protegido. Se pueden simular utilizando convenciones:

- **Público**: Atributos y métodos sin guiones bajos iniciales.
- **Privado**: Atributos y métodos con un guion bajo inicial (`_`).
- **Protegido**: Atributos y métodos con dos guiones bajos iniciales (`__`).

Herencia

La herencia permite crear clases que heredan atributos y métodos de otras clases existentes (clases base). Promueve la reutilización de código y la organización jerárquica de clases.

Clases Base y Subclases

Una clase que hereda de otra se denomina subclase. La subclase hereda todos los atributos y métodos públicos de la clase base. Se puede extender o modificar la funcionalidad heredada en la subclase.

Polimorfismo

El polimorfismo permite que objetos de diferentes clases respondan al mismo mensaje de manera diferente. Se basa en la sobreescritura de métodos en clases derivadas.

Métodos Sobreescritos

Cuando una subclase define un método con el mismo nombre que un método heredado de la clase base, se produce una sobreescritura. La subclase proporciona su propia implementación del método, redefiniendo el comportamiento heredado.

Abstracción

La abstracción se centra en las características esenciales de un objeto o proceso, ignorando detalles de implementación. Permite crear clases que definen interfaces sin proporcionar implementaciones completas.

Clases Abstractas

Python no posee clases abstractas como tal, pero se puede simular su comportamiento mediante clases base que definen métodos sin implementar (usando `pass`). Las subclases deben implementar estos métodos para proporcionar funcionalidad completa.

Ejemplos Prácticos

Creando una Clase de Vehículo

```
class Vehiculo:
    def __init__(self, marca, modelo, color):
        self.marca = marca
        self.modelo = modelo
        self.color = color

    def arrancar(self):
        print(f"El vehículo {self.marca} {self.modelo} está arrancando.")

    def acelerar(self):
        print(f"El vehículo {self.marca} {self.modelo} está acelerando.")

    def frenar(self):
        print(f"El vehículo {self.marca} {self.modelo} está frenando.")

miAuto = Vehiculo("Toyota", "Corolla", "Azul")
miMoto = Vehiculo("Honda", "CBX", "Negra")

# Accionando los métodos de los objetos:
miAuto.arrancar()
miMoto.acelerar()
miAuto.frenar()

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

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

juan = Persona("Juan", 30)
juan.saludar()
```

Actividades!

Ejercicio 1: Crear una Clase de Rectángulo

Objetivo: Crear una clase llamada Rectangulo que tenga dos atributos: ancho y alto.

Ejercicio 2: Crear una clase de Cuenta Bancaria

Objetivo: Implementar una clase CuentaBancaria que permita a los usuarios realizar operaciones bancarias básicas como depositar, retirar y consultar el saldo.

Ejercicio 3: Crear una Clase de Producto

Objetivo: Crear una clase llamada Producto que tenga tres atributos: nombre, precio y cantidad. Y tenga 3 metodos, Actualizar Precio, Actualizar Cantidad y Calcular valor total, el primero modificara el precio del producto, el segundo la cantidad y el ultimo debera multiplicar la cantidad por el precio del producto seleccionado para saber el valor total del inventario.

Ejercicio 4: Crear una Clase de Contador

Objetivo: Crear una clase llamada Contador que tenga un atributo cuenta que empieza en 0. Y que posea 4 metodos, Incrementar(sumará 1), Decrementar(restará 1, no admite negativos osea hasta maximo 0), Mostrar Contador(deberá devolver el valor actual de contador) y Reiniciar(volverá a cero el contador)