Lectura sesión Introducción a la Programación Orientada a Objetos (POO)

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

R **por Kibernum Capacitación S.A.**

¿Qué es la POO y cuáles son sus principios?

La POO se basa en 4 principios fundamentales:

Abstracción

Consiste en ocultar los detalles internos de un objeto y mostrar solo lo necesario. Ejemplo: un automóvil tiene un botón para encender, no necesitas saber cómo funciona el motor.

Herencia

Permite que una clase herede atributos y métodos de otra clase. Reutiliza código y favorece la jerarquía entre clases.

Encapsulamiento

Protege los datos del objeto, permitiendo el acceso solo a través de métodos definidos. Evita modificaciones directas no deseadas.

Polimorfismo

Permite que métodos con el mismo nombre se comporten de manera distinta según el contexto. Por ejemplo, un método dibujar() se comporta diferente si es llamado por un círculo o un cuadrado.

Ventajas de la POO

Mejora la organización del código

La estructura basada en objetos permite una organización más intuitiva y clara del código fuente, facilitando su comprensión.

Permite reutilizar componentes

Los objetos y clases pueden ser reutilizados en diferentes partes del programa o incluso en otros proyectos.

Hace el código más escalable y mantenible

La modularidad facilita la expansión del sistema y la corrección de errores sin afectar otras partes.

Fomenta el trabajo colaborativo

Al dividir el sistema en objetos o clases, diferentes desarrolladores pueden trabajar en distintas partes simultáneamente.

Clases y Objetos

¿Qué es una clase?

Una clase es un molde o plantilla que define atributos y comportamientos comunes a un tipo de objeto.

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

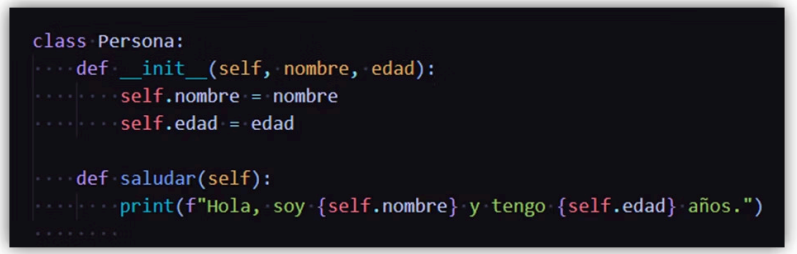
¿Qué es un objeto?

Un objeto es una instancia concreta de una clase.

```
persona1 = Persona()
```

La relación entre clases y objetos es fundamental en la POO. Las clases definen la estructura mientras que los objetos representan instancias específicas con valores concretos para sus atributos.

Definición de una clase con atributos y métodos



```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.")
```

Explicación:

- __init__ es el constructor, se llama automáticamente al crear un objeto.
- self se refiere al objeto actual.
- nombre y edad son atributos.
- saludar es un método.

La definición de una clase en Python sigue una estructura clara donde primero se declara el constructor y luego los métodos que implementan el comportamiento de los objetos de esa clase.

Instanciación de objetos

La instanciación es el proceso de crear objetos a partir de una clase. En Python, esto se realiza llamando a la clase como si fuera una función, pasando los argumentos necesarios para el constructor.

Una vez creado el objeto, podemos acceder a sus atributos y llamar a sus métodos utilizando la notación de punto.

```
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.
```

Encapsulamiento y visibilidad

Python no tiene modificadores como private o public, pero usa convenciones para controlar el acceso:

Notación	Significado
Variable	Pública (accesible desde fuera)
_variable	Protegida (acceso interno sugerido)
__variable	Privada (nombre "enmangado")

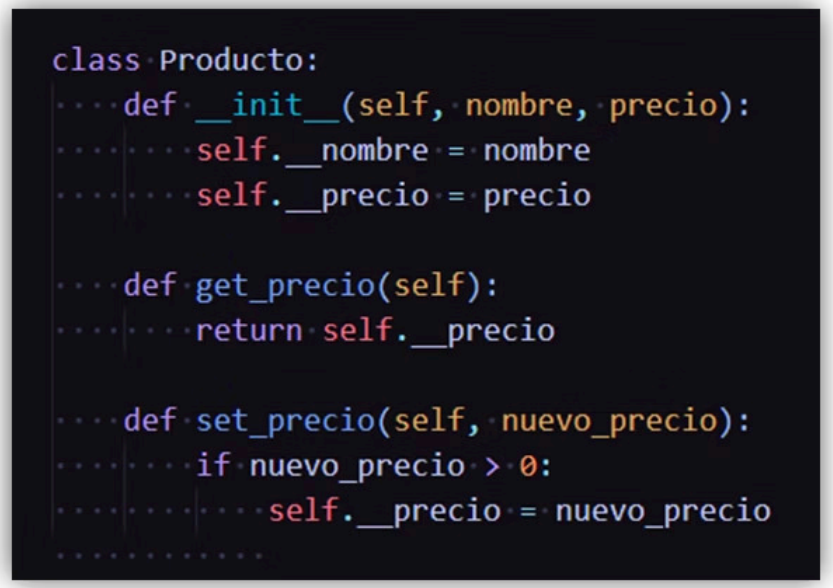
Atributos y métodos privados

```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
```

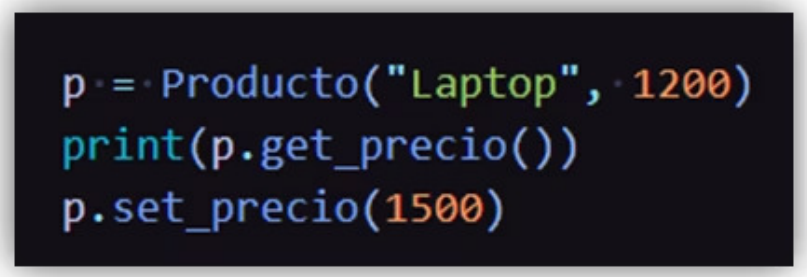
Nota: No se puede acceder directamente a __saldo desde fuera de la clase.

Getters y Setters

Permiten acceder y modificar atributos privados de forma controlada.



```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)
```

Los getters y setters son métodos que permiten leer y modificar atributos de manera controlada, aplicando validaciones o lógica adicional cuando sea necesario.

Al utilizar getters y setters, podemos encapsular la lógica de validación y transformación de datos, manteniendo la integridad de los objetos.

Decoradores @property y @setter

Sintaxis moderna y más limpia para definir getters y setters.

```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
```

Los decoradores @property y @setter proporcionan una forma más elegante y pythónica de implementar getters y setters, permitiendo acceder a los atributos como si fueran públicos mientras se mantiene el control sobre su acceso y modificación.

Conclusión

La Programación Orientada a Objetos permite modelar software más cercano al mundo real. Sus pilares —**abstracción**, **encapsulamiento**, **herencia** y **polimorfismo**— dan forma a sistemas modulares, reutilizables y fáciles de mantener.

Aprender a diseñar buenas clases, usar constructores, métodos y proteger atributos son pasos esenciales para dominar este paradigma.

Actividades Prácticas

Actividad Práctica Guiada

Objetivo: Aplicar los principios de la 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: 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.

Ejercicio Final

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.

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

Se espera que apliques correctamente el diseño de clases en Python, comprendas y utilices el constructor, los métodos, el encapsulamiento y los decoradores @property, y valides condiciones correctamente.

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 (puedes copiarlo):

```python
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).

Paso 2: Definir método para mostrar información

```python
    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.

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.

Paso 4: Crear un objeto y probar los métodos

```python
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.

Recomendaciones de entrega:

• Archivo producto.py con el código completo y funcional.

• Capturas de pantalla de cada paso del desarrollo y ejecución.

• Breve texto en archivo .txt o .md explicando:

- Qué representa la clase.
- Cómo se aplican los principios de POO usados: constructor, encapsulamiento, decoradores.

Ejercicio Final – Caso de Uso para Implementación Autónoma

E**nunciado:**

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)

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:

```
Título: ______
Autor: ______
Páginas: ____
```

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).

¿Qué se espera del estudiante?

- Que aplique correctamente el diseño de clases en Python.
- Que comprenda y utilice el constructor, los métodos, el encapsulamiento y los decoradores @property.
- Que valide condiciones correctamente y controle los errores de forma adecuada.