Programación orientada a objetos en Python

M2: FUNDAMENTOS DE INGENIERÍA DE DATOS

AE1: DESARROLLAR UNA APLICACIÓN ORIENTADA A OBJETOS EN LENGUAJE PYTHON PARA RESOLVER UN PROBLEMA.

alkemy

Introducción 🤦‍♂️

La Programación Orientada a Objetos (POO) es un paradigma que ha revolucionado la forma en que desarrollamos software. Surgida como una respuesta a la creciente complejidad de los programas y la necesidad de manejarlos de manera más eficiente, la POO permite modelar problemas del mundo real de una forma más natural y organizada. A través de la **creación de objetos que representan entidades con atributos y comportamientos, los desarrolladores pueden crear sistemas más robustos, flexibles y mantenibles.**

Python, siendo un lenguaje multiparadigma, ofrece un soporte completo para la POO. Aunque muchos asocian Python con scripts sencillos y programación procedural, su capacidad para implementar conceptos avanzados de POO lo convierte en una herramienta poderosa para proyectos de cualquier escala. Este manual tiene como objetivo profundizar en los fundamentos de la POO en Python, explorando no solo los conceptos básicos sino también prácticas avanzadas y principios de diseño que te ayudarán a escribir código de alta calidad.

Aprendizajes esperados

Cuando finalices la lección serás capaz de:

- Definir y explicar qué es la Programación Orientada a Objetos y sus principios fundamentales.
- Identificar y aplicar los conceptos de abstracción, encapsulamiento, herencia y polimorfismo en Python.
- Crear clases y objetos, comprendiendo la diferencia entre ambos.
- Implementar atributos y métodos, diferenciando entre los de instancia y los de clase.
- Utilizar constructores e inicializadores para gestionar el estado inicial de los objetos.
- Aplicar el encapsulamiento y controlar la visibilidad de atributos y métodos.
- Utilizar getters, setters y decoradores para acceder y modificar atributos privados de forma segura.
- Entender y aplicar la herencia, incluyendo la herencia múltiple y el Método de Resolución de Métodos (MRO).
- Implementar polimorfismo y comprender su importancia en el diseño de software.
- Utilizar dunder methods para personalizar el comportamiento de objetos y operadores.
- Aplicar principios de diseño orientado a objetos como DRY y SOLID para mejorar la calidad y mantenibilidad del código.

Introducción a la POO

QUÉ ES LA POO

La Programación Orientada a Objetos es un paradigma que organiza el software en torno a "objetos" que representan entidades o conceptos del mundo real o abstracto. Cada objeto es una instancia de una clase y tiene atributos (datos) y métodos (funcionalidad). Este enfoque permite que el software sea más modular, facilitando la reutilización y el mantenimiento.

La POO se basa en el concepto de modelar sistemas como una colección de objetos que interactúan entre sí. Esto es especialmente útil en proyectos grandes y complejos, donde la organización y la escalabilidad son esenciales.

PRINCIPIOS BÁSICOS DE LA POO

1. **Abstracción**: Consiste en simplificar la complejidad al ocultar detalles innecesarios y resaltar las características esenciales de un objeto. En programación, esto se traduce en definir clases que capturen la esencia de las entidades que representan.
   Ejemplo: Una clase `Vehículo` que representa las características comunes de todos los vehículos, como `marca`, `modelo` y métodos como `arrancar()`.
2. **Encapsulamiento**: Es el mecanismo de restringir el acceso directo a algunos de los componentes de un objeto. Esto protege los datos y garantiza que el objeto maneje su propia información.
   Ejemplo: Atributos privados en una clase que solo pueden ser accedidos o modificados mediante métodos públicos.
3. **Herencia**: Permite crear nuevas clases basadas en clases existentes. La clase derivada hereda atributos y métodos de la clase base, lo que promueve la reutilización de código.
   Ejemplo: Una clase `Coche` que hereda de `Vehículo`, agregando atributos específicos como `tipo_de_combustible`.
4. **Polimorfismo**: Es la capacidad de presentar la misma interfaz para diferentes tipos de datos. En otras palabras, diferentes objetos pueden ser tratados de la misma manera aunque su comportamiento interno sea diferente.
   Ejemplo: Dos clases, `Perro` y `Gato`, ambas tienen un método `hacer_sonido()`, pero su implementación es diferente. Al iterar sobre una lista de animales, podemos llamar `hacer_sonido()` sin preocuparnos del tipo específico.

VENTAJAS DE LA POO

- **Modularidad**: Facilita la división del programa en partes más pequeñas y manejables.

- **Reutilización de código**: Mediante la herencia y la composición, es posible reutilizar código existente, reduciendo esfuerzos y errores.
- **Facilidad de mantenimiento**: Los cambios en el código son más fáciles de implementar y afectan menos a otras partes del programa.
- **Escalabilidad**: Permite añadir nuevas funcionalidades sin modificar el código existente de manera significativa.
- **Claridad y organización**: El código orientado a objetos es generalmente más legible y se alinea mejor con el pensamiento humano.

CLASES Y OBJETOS

DEFINICIÓN DE UNA CLASE

Una **clase** es un plano o plantilla que define las propiedades (atributos) y comportamientos (métodos) que tendrán los objetos creados a partir de ella.

Sintaxis básica:

```
class NombreDeLaClase:
    # Atributos y métodos
    pass
```

Ejemplo:

```
class Persona:
    especie = 'Humano'  # Atributo de clase

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

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

ATRIBUTOS Y MÉTODOS

- **Atributos**:
    - **De clase**: Compartidos por todas las instancias de la clase.
    - **De instancia**: Propios de cada objeto.
- **Métodos**:
    - Son funciones definidas dentro de una clase que describen los comportamientos de los objetos.

Ejemplo:

```
persona1 = Persona("Carlos", 28)
persona2 = Persona("María", 32)

print(persona1.especie)  # Output: Humano
print(persona2.nombre)   # Output: María
```

CONSTRUCTORES E INICIALIZADORES

El método `__init__` es el constructor en Python y se llama automáticamente al crear un nuevo objeto. Se utiliza para inicializar los atributos de la instancia.

Ejemplo:

```
class Circulo:
    def __init__(self, radio):
        self.radio = radio
```

INSTANCIACIÓN DE OBJETOS

Crear una instancia es crear un objeto a partir de una clase.

python
Copiar código

```
circulo1 = Circulo(5)
print(circulo1.radio)  # Output: 5
```

Cada objeto es independiente y tiene su propio espacio de datos.

ENCAPSULAMIENTO Y VISIBILIDAD

El encapsulamiento se logra en Python mediante convenciones de nombres:

- **Públicos**: Sin guiones bajos (e.g., `nombre`).
- **Protegidos**: Un guión bajo (e.g., `_nombre`).
- **Privados**: Dos guiones bajos (e.g., `__nombre`).

Estas convenciones indican al desarrollador el nivel de acceso esperado.

ATRIBUTOS Y MÉTODOS PRIVADOS

ATRIBUTOS Y MÉTODOS PRIVADOS

Los atributos y métodos privados no deben ser accedidos directamente desde fuera de la clase.

Ejemplo:

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

    def mostrar_saldo(self):
        print(f"Saldo: {self.__saldo}")
```

Intentar acceder a `__saldo` desde fuera de la clase resultará en un error.

GETTERS Y SETTERS

Los getters y setters permiten acceder y modificar atributos privados de forma controlada.

Ejemplo:

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

    def get_saldo(self):
        return self.__saldo

    def set_saldo(self, saldo):
        if saldo >= 0:
            self.__saldo = saldo
        else:
            print("El saldo no puede ser negativo.")
```

DECORADORES

Los decoradores `@property` y `@<atributo>.setter` simplifican el uso de getters y setters.

Ejemplo:

```
class Producto:
    def __init__(self, precio):
```

```
        self._precio = precio

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

    @precio.setter
    def precio(self, valor):
        if valor > 0:
            self._precio = valor
        else:
            print("El precio debe ser positivo.")
```

Ahora, podemos acceder y modificar `precio` como si fuera un atributo público:

```
p = Producto(100)
print(p.precio)  # Output: 100
p.precio = 150
```

HERENCIA EN PYTHON

CONCEPTO DE HERENCIA Y JERARQUÍA DE CLASES

La herencia permite que una clase (subclase) herede atributos y métodos de otra clase (superclase), formando una jerarquía.

Ejemplo:

python
Copiar código

```
class Animal:
    def comer(self):
        print("El animal está comiendo.")

class Perro(Animal):
    def ladrar(self):
        print("El perro está ladrando.")
```

`Perro` hereda de `Animal` y, por lo tanto, puede usar el método `comer()`.

INSTANCIACIÓN DE SUBCLASES

```
mi_perro = Perro()
mi_perro.comer()   # Output: El animal está comiendo.
mi_perro.ladrar()  # Output: El perro está ladrando.
```

HERENCIA MÚLTIPLE Y MRO (MÉTODO DE RESOLUCIÓN DE MÉTODOS)

La herencia múltiple permite que una clase herede de múltiples superclases.

Ejemplo:

```
class Terrestre:
    def desplazar(self):
        print("El animal camina.")

class Acuatico:
    def desplazar(self):
        print("El animal nada.")

class Anfibio(Terrestre, Acuatico):
    pass

anfibio = Anfibio()
anfibio.desplazar()  # Output: El animal camina.
```

El MRO determina que `desplazar()` se toma de `Terrestre` porque está antes en la lista de herencia.

Visualización del MRO:

```
print(Anfibio.__mro__)
```

Salida:

```
(<class '__main__.Anfibio'>, <class '__main__.Terrestre'>, <class
'__main__.Acuatico'>, <class 'object'>)
```

POLIMORFISMO

QUÉ ES POLIMORFISMO

El polimorfismo permite que objetos de diferentes clases sean tratados como objetos de una clase común, principalmente a través de interfaces compartidas.

Ejemplo:

python
Copiar código

```
class Ave:
    def volar(self):
        print("El ave vuela.")

class Avion:
    def volar(self):
        print("El avión vuela.")

def hacer_volar(objeto):
    objeto.volar()

pajaro = Ave()
boeing = Avion()

hacer_volar(pajaro)  # Output: El ave vuela.
hacer_volar(boeing)  # Output: El avión vuela.
```

No importa el tipo del objeto, mientras tenga el método `volar()`, la función `hacer_volar()` funcionará.

SOBRECARGA

Aunque Python no soporta sobrecarga de métodos de manera tradicional, se puede emular utilizando valores por defecto o argumentos variables.

Ejemplo con argumentos variables:

```
class Calculadora:
    def sumar(self, *args):
        return sum(args)

calc = Calculadora()
print(calc.sumar(1, 2))          # Output: 3
print(calc.sumar(1, 2, 3, 4))    # Output: 10
```

DUNDER METHODS

Los métodos especiales, o dunder methods, permiten personalizar el comportamiento de los objetos con respecto a operadores y funciones integradas.

Ejemplo del método `__str__`:

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

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

persona = Persona("Lucía", 27)
print(persona)  # Output: Lucía, 27 años
```

Ejemplo del método `__eq__` para comparación:

```
class Punto:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __eq__(self, otro):
        return self.x == otro.x and self.y == otro.y

p1 = Punto(1, 2)
p2 = Punto(1, 2)
print(p1 == p2)  # Output: True
```

PRINCIPIOS DE DISEÑO ORIENTADO A OBJETOS

PRINCIPIO DRY

Don't Repeat Yourself aboga por evitar la duplicación de código. Cada pieza de información debe tener una representación única y sin ambigüedades dentro del sistema.

Aplicación:

- Crear funciones o métodos para lógica repetida.
- Utilizar herencia o composición para compartir funcionalidad.
- Refactorizar código duplicado.

Ejemplo:

Antes (código duplicado):

```
def area_rectangulo(ancho, alto):
    return ancho * alto

def area_cuadrado(lado):
    return lado * lado
```

Después (aplicando DRY):

```
def area_rectangulo(ancho, alto=None):
    if alto is None:
        alto = ancho
    return ancho * alto
```

PRINCIPIO SOLID

1. **Single Responsibility Principle**: Una clase debe tener una sola responsabilidad.
2. **Open/Closed Principle**: Las entidades deben estar abiertas para extensión, pero cerradas para modificación.
3. **Liskov Substitution Principle**: Las clases derivadas deben poder sustituir a sus clases base sin alterar el correcto funcionamiento del programa.
4. **Interface Segregation Principle**: Los clientes no deben verse obligados a depender de interfaces que no utilizan.
5. **Dependency Inversion Principle**: Las dependencias deben ir de los detalles a las abstracciones.

Ejemplo del Principio Abierto/Cerrado:

En lugar de modificar una clase existente, se extiende su funcionalidad.

Antes (violando el principio):

```
class Procesador:
    def procesar(self, tipo):
        if tipo == 'texto':
```

```
            self.procesar_texto()
        elif tipo == 'imagen':
            self.procesar_imagen()
```

Después (aplicando el principio):

```
class Procesador:
    def procesar(self):
        pass

class ProcesadorTexto(Procesador):
    def procesar(self):
        # Lógica para procesar texto
        pass

class ProcesadorImagen(Procesador):
    def procesar(self):
        # Lógica para procesar imágenes
        pass
```

Ahora, podemos añadir nuevos procesadores sin modificar la clase base.

Cierre ✍️

La comprensión y aplicación de la Programación Orientada a Objetos es esencial para cualquier desarrollador que busque crear software escalable y mantenible. A través de los conceptos de clases, objetos, herencia y polimorfismo, es posible modelar soluciones que reflejen fielmente las complejidades del mundo real. Además, al adherirse a principios de diseño como DRY y SOLID, se promueve la creación de código limpio, eficiente y preparado para el cambio.

Python, con su sintaxis clara y su soporte robusto para la POO, es una excelente plataforma para aprender y aplicar estos conceptos. Te animamos a practicar y experimentar, creando tus propias clases y estructuras, y aplicando los principios aprendidos en este manual. La práctica constante es la clave para dominar la POO y convertirte en un desarrollador más competente y versátil.

Referencias 📚

- **Documentación oficial de Python**: Clases
  - Profundiza en la sintaxis y características de las clases en Python.
- **"Python Crash Course"** por Eric Matthes
  - Un libro práctico que introduce conceptos fundamentales de Python, incluyendo POO.
- **"Head First Object-Oriented Analysis and Design"** por Brett McLaughlin
  - Aborda principios de diseño orientado a objetos de forma amigable y visual.
- **"Clean Architecture"** por Robert C. Martin
  - Explora arquitecturas de software y cómo los principios SOLID se aplican en la práctica.
- **Cursos en línea**:
  - **Pluralsight**: Cursos sobre POO y diseño de software en Python.
  - **Udemy**: "Python 3: Deep Dive (Part 4 - OOP)" por Fred Baptiste.
- **Comunidades y foros**:
  - **Stack Overflow**: Respuestas y discusiones sobre problemas específicos.
  - **r/learnpython** en Reddit: Comunidad activa para aprender Python.
- **Blogs y artículos**:
  - **Real Python**: Artículos detallados sobre POO en Python.
  - **GeeksforGeeks**: Explicaciones y ejemplos prácticos.

¡Muchas gracias!

Nos vemos en la próxima lección