Python

Programación Orientada a Objetos (OOP)

CertiDevs

Índice de contenidos

1. Programación Orientada a Objetos

Programación Orientada a Objetos (OOP) es un paradigma de programación que utiliza objetos y clases para estructurar y organizar el código.

Python es un lenguaje de programación que admite OOP y proporciona una sintaxis y características fáciles de usar para trabajar con clases y objetos.

La OOP es un paradigma de programación que se basa en la organización y estructuración del código utilizando **objetos**, que son entidades que encapsulan **datos** y **comportamientos** relacionados. Este enfoque permite diseñar soluciones de software más **modulares**, **reutilizables** y **escalables**.

La POO se basa en varios conceptos clave, que incluyen:

- **Clases**: Las clases son plantillas o "moldes" que definen la estructura y el comportamiento de los objetos. Una clase especifica los atributos (datos) y métodos (funciones) que los objetos creados a partir de esa clase tendrán.
- **Objetos**: Los objetos son instancias de clases. Cada objeto creado a partir de una clase tiene su propio conjunto de atributos y puede utilizar los métodos definidos en la clase.
- **Encapsulamiento**: El encapsulamiento es un principio que permite ocultar los detalles de implementación de una clase y controlar el acceso a sus atributos y métodos. Esto facilita el mantenimiento y mejora la modularidad del código.
- **Herencia**: La herencia es un mecanismo que permite crear una nueva clase basada en otra existente. La nueva clase hereda los atributos y métodos de la clase base y puede agregar o sobrescribir características según sea necesario.
- **Polimorfismo**: El polimorfismo permite que diferentes clases tengan métodos con el mismo nombre pero con comportamientos diferentes. Esto facilita la reutilización de código y permite diseñar interfaces más flexibles.
- **Composición**: La composición se logra al incluir objetos de otras clases como atributos dentro de una clase. Esto permite que una clase tenga acceso a los atributos y métodos de otras clases, lo que facilita la reutilización y el mantenimiento del código.

Aprender POO tiene varias **ventajas**:

- **Modularidad**: La POO facilita la organización del código en componentes independientes y fácilmente intercambiables. Esto facilita el mantenimiento y la extensión del código a lo largo del tiempo.
- **Reutilización de código**: La POO permite reutilizar y compartir código a través de la herencia y la composición de clases. Esto reduce la duplicación de código y mejora la eficiencia del desarrollo.
- **Abstracción**: La POO permite abstraer y modelar conceptos del mundo real en términos de objetos y clases. Esto facilita la comprensión y el diseño de soluciones de software.
- **Flexibilidad**: La POO permite diseñar sistemas de software que son más fáciles de adaptar y extender a medida que cambian los requisitos. Esto es especialmente útil en proyectos de

desarrollo de software a largo plazo o en equipos de desarrollo grandes.

2. Clases en Python

Una **clase** en Python es una **plantilla** que define un conjunto de **atributos** y **métodos** para un objeto.

Los objetos son instancias de una clase y pueden tener **estados** y **comportamientos** específicos según la definición de la clase.

Para crear una clase en Python, utilice la palabra clave class seguida del nombre de la clase y dos puntos `:`

```
class Persona:
    pass
```

3. Constructor

El **constructor** es un método especial que se llama automáticamente cuando se crea un objeto de una clase.

En Python, el constructor se define con el nombre `init()`.

El primer parámetro de este método siempre debe ser `self`, que es una referencia al objeto que se está creando.

```
class Persona:
    def __init__(self):
        print("Se ha creado una persona.")
```

Para crear un objeto de la clase `Persona`, simplemente llame al nombre de la clase seguido de paréntesis `()`.

```
persona = Persona()  # Se imprime "Se ha creado una persona."
```

4. Atributos

Los **atributos** son variables asociadas a un objeto.

Se pueden definir **atributos** en el **constructor** utilizando `self` seguido del nombre del atributo y asignándoles un valor.

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

```
        self.nombre = nombre
        self.edad = edad
```

Para acceder a los atributos de un objeto, se utiliza la notación de punto `.` seguida del nombre del atributo.

```
persona = Persona("Juan", 30)
print(persona.nombre)  # "Juan"
print(persona.edad)  # 30
```

5. Métodos

Los **métodos** son **funciones** asociadas a un objeto.

Se definen dentro de la **clase** utilizando la palabra clave `def`, seguida del nombre del método y paréntesis `()`.

Al igual que en el constructor, el primer parámetro de un método siempre debe ser `self`.

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

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

Para **llamar a un método** de un **objeto**, utilice la notación de punto `.` seguida del nombre del método y paréntesis `()`.

```
persona = Persona("Juan", 30)
persona.presentarse()  # "Hola, mi nombre es Juan y tengo 30 años."
```

6. Ejemplo 1: Clase Estudiante

Crear una clase para representar a los estudiantes de una universidad.

Los estudiantes tienen un **nombre**, una **carrera** y una **calificación promedio**.

Además, pueden estudiar y realizar exámenes.

```
class Estudiante:
    def __init__(self, nombre, carrera, promedio):
        self.nombre = nombre
```

```
        self.carrera = carrera
        self.promedio = promedio

    def estudiar(self, horas):
        print(f"{self.nombre} estudia durante {horas} horas.")

    def realizar_examen(self, nota):
        self.promedio = (self.promedio + nota) / 2
        print(f"{self.nombre} ha obtenido una nota de {nota} en el examen.")
```

Ahora, creamos objetos de la clase `Estudiante` y utilizamos sus atributos y métodos.

```
# Crear estudiantes
juan = Estudiante("Juan", "Ingeniería", 8.5)
maria = Estudiante("María", "Matemáticas", 9.2)

# Utilizar atributos
print(juan.nombre)  # "Juan"
print(maria.carrera)  # "Matemáticas"

# Utilizar métodos
juan.estudiar(3)  # "Juan estudia durante 3 horas."
maria.realizar_examen(8.0)  # "María ha obtenido una nota de 8.0 en el examen."
print(maria.promedio)  # 8.6
```

7. Ejemplo 2: Clase CuentaBancaria

Las cuentas bancarias tienen un **titular**, un **número de cuenta** y un **saldo**.

Los titulares pueden depositar y retirar dinero.

```
class CuentaBancaria:
    def __init__(self, titular, numero_de_cuenta, saldo=0):
        self.titular = titular
        self.numero_de_cuenta = numero_de_cuenta
        self.saldo = saldo

    def depositar(self, cantidad):
        self.saldo += cantidad
        print(f"{self.titular} depositó ${cantidad}. Nuevo saldo: ${self.saldo}")

    def retirar(self, cantidad):
        if cantidad > self.saldo:
            print(f"Saldo insuficiente. El saldo actual es de ${self.saldo}")
        else:
            self.saldo -= cantidad
            print(f"{self.titular} retiró ${cantidad}. Nuevo saldo: ${self.saldo}")
```

Creemos objetos de la clase `Cuenta Bancaria` y utilicemos sus atributos y métodos.

```
# Crear cuentas bancarias
cuenta1 = CuentaBancaria("Pedro", "123456", 500)
cuenta2 = CuentaBancaria("Laura", "789012", 2000)

# Utilizar atributos
print(cuenta1.titular)  # "Pedro"
print(cuenta2.numero_de_cuenta)  # "789012"

# Utilizar métodos
cuenta1.depositar(300)  # "Pedro depositó $300. Nuevo saldo: $800"
cuenta2.retirar(500)  # "Laura retiró $500. Nuevo saldo: $1500"

# Intentar retirar más dinero del disponible
cuenta1.retirar(1000)  # "Saldo insuficiente. El saldo actual es de $800"
```

8. Ejemplo 3: Clase Círculo

Considere una clase para representar círculos en un plano.

Los círculos tienen un **radio** y se pueden calcular su **área** y **perímetro**.

```
import math

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

    def area(self):
        return math.pi * self.radio ** 2

    def perimetro(self):
        return 2 * math.pi * self.radio
```

Creemos objetos de la clase `Circulo` y utilicemos sus **atributos** y **métodos**.

```
# Crear círculos
circulo1 = Circulo(5)
circulo2 = Circulo(10)

# Utilizar atributos
print(circulo1.radio)  # 5

# Utilizar métodos
print(circulo1.area())  # 78.53981633974483
```

```
print(circulo2.perimetro())  # 62.83185307179586
```

9. Ejemplo 4: Clase Estadisticas

Esta clase permite calcular estadísticas básicas como la **media**, **mediana** y **desviación estándar** de una lista de números.

```
import statistics

class Estadisticas:
    def __init__(self, data):
        self.data = data

    def media(self):
        return sum(self.data) / len(self.data)

    def mediana(self):
        return statistics.median(self.data)

    def desviacion_estandar(self):
        return statistics.stdev(self.data)

# Usar la clase Estadisticas
datos = [1, 2, 3, 4, 5, 6, 7, 8, 9]
estadisticas = Estadisticas(datos)

print("Media:", estadisticas.media())
print("Mediana:", estadisticas.mediana())
print("Desviación estándar:", estadisticas.desviacion_estandar())
```

10. Ejemplo 5: Clase Contador

Esta clase permite contar la **frecuencia** de elementos en una lista y calcular **el elemento más frecuente**.

```
from collections import Counter

class Contador:
    def __init__(self, elementos):
        self.elementos = elementos

    def frecuencias(self):
        return Counter(self.elementos)

    def elemento_mas_frecuente(self):
        frecuencias = self.frecuencias()
```

```
        return frecuencias.most_common(1)[0]

# Usar la clase Contador
elementos = ["A", "B", "A", "C", "B", "A", "D", "C", "C"]
contador = Contador(elementos)

print("Frecuencias:", contador.frecuencias())
print("Elemento más frecuente:", contador.elemento_mas_frecuente())
```

11. Composición de clases en Python

La **composición** es un concepto importante en la Programación Orientada a Objetos que permite construir clases más complejas utilizando otras clases más simples.

La **composición** se logra al incluir objetos de otras clases como atributos dentro de una clase.

Esto permite que una clase tenga acceso a los atributos y métodos de otras clases, lo que facilita la reutilización y el mantenimiento del código.

12. Ejemplo: Clase Empleado y Departamento

Suponga que desea modelar **empleados** y **departamentos** de una empresa.

Un **empleado** tiene un **nombre**, una **posición** y un **salario**.

Un **departamento** tiene un **nombre** y una **lista de empleados** que trabajan en ese departamento.

Primero, creemos la clase `Empleado`:

```
class Empleado:
    def __init__(self, nombre, posicion, salario):
        self.nombre = nombre
        self.posicion = posicion
        self.salario = salario

    def mostrar_info(self):
        print(f"{self.nombre} - {self.posicion} (${self.salario})")
```

A continuación, creemos la clase `Departamento` utilizando la composición con la clase `Empleado`:

```
class Departamento:
    def __init__(self, nombre):
        self.nombre = nombre
        self.empleados = []
```

```
    def agregar_empleado(self, empleado):
        self.empleados.append(empleado)

    def mostrar_empleados(self):
        print(f"Empleados del departamento {self.nombre}:")
        for empleado in self.empleados:
            empleado.mostrar_info()
```

En este ejemplo, la clase `Departamento` tiene un atributo llamado `empleados`, que es una lista de objetos de la clase `Empleado`. Esto permite que un departamento tenga acceso a los atributos y métodos de sus empleados.

Ahora, creemos objetos de las clases `Empleado` y `Departamento` y utilicemos la **composición**:

```
# Crear empleados
empleado1 = Empleado("Juan", "Desarrollador", 60000)
empleado2 = Empleado("María", "Analista", 55000)
empleado3 = Empleado("Pedro", "Desarrollador", 65000)

# Crear un departamento
departamento_tecnologia = Departamento("Tecnología")

# Agregar empleados al departamento
departamento_tecnologia.agregar_empleado(empleado1)
departamento_tecnologia.agregar_empleado(empleado2)
departamento_tecnologia.agregar_empleado(empleado3)

# Mostrar empleados del departamento
departamento_tecnologia.mostrar_empleados()
```

La salida será:

```
Empleados del departamento Tecnología:
Juan - Desarrollador ($60000)
María - Analista ($55000)
Pedro - Desarrollador ($65000)
```

La composición de clases en Python es una forma poderosa de combinar clases más simples para crear clases más complejas y funcionales. Esto mejora la reutilización y el mantenimiento del código y permite modelar de manera más efectiva las relaciones entre diferentes conceptos en su programa.