Plan de Formación Profesional y Continua

cursos gratuitos
de formación profesional

[Programación en Python]

Programación Orientada a Objetos en Python

alkemy

fomentar empleo

Ministerio de Trabajo, Empleo y Seguridad Social
Argentina

Introducción

El paradigma de objetos es una forma de organizar y estructurar el código en programación, que se basa en la conceptualización de objetos como entidades que encapsulan datos y comportamiento. Python, como lenguaje de programación orientado a objetos, ofrece un conjunto de características y conceptos que permiten aplicar este paradigma de manera eficiente y elegante.

En este documento podrás acceder a lectura complementaria te será de utilidad para profundizar los temas vistos en clase y en los demás recursos asincrónicos. Aquí exploraremos algunos aspectos fundamentales del paradigma de objetos en Python, centrándonos en temas que complementan los conceptos básicos ya conocidos, como polimorfismo, herencia, abstracción y encapsulamiento. Nuestro objetivo es brindar una visión más completa y profunda de cómo utilizar el paradigma de objetos en Python para desarrollar aplicaciones robustas y flexibles.

En las secciones siguientes, exploraremos temas como decoradores, métodos especiales, composición, métodos estáticos y descriptores. Estos conceptos ampliarán tu comprensión de cómo diseñar y estructurar tus clases de manera eficiente, aprovechando al máximo las características y capacidades que Python ofrece.

Programación Orientada a Objetos

Composición

En Python, la composición es un concepto de programación orientada a objetos que permite crear objetos más grandes y complejos combinando varios objetos más pequeños. Esto se logra al incluir instancias de una clase dentro de otra clase como atributos. La clase que contiene las instancias de otras clases se conoce como la clase compuesta.

Para entender mejor la composición en Python, veamos un ejemplo. Supongamos que estamos modelando una tienda en línea y tenemos dos clases: `Producto` y `CarritoDeCompras`. La clase `Producto` representa un producto que se puede comprar y tiene atributos como `nombre`, `precio` y `cantidad en stock`. La clase `CarritoDeCompras` representa el carrito de compras de un cliente y contiene una lista de productos que el cliente ha seleccionado para comprar.

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


class CarritoDeCompras:
    def __init__(self):
        self.productos = []

    def agregar_producto(self, producto):
        self.productos.append(producto)

    def calcular_total(self):
```

```
        total = 0
        for producto in self.productos:
            total += producto.precio
        return total
```

👉🏽En este ejemplo, la clase `CarritoDeCompras` contiene una lista de productos (`self.productos`) como uno de sus atributos. Esto es un ejemplo de composición, ya que la clase `CarritoDeCompras` está compuesta por instancias de la clase `Producto`.

Podemos crear objetos de estas clases y utilizar la composición para agregar productos al carrito de compras y calcular el total de la compra:

```
# Creamos algunos productos
producto1 = Producto("Camiseta", 20, 5)
producto2 = Producto("Pantalón", 30, 3)
producto3 = Producto("Zapatos", 50, 2)

# Creamos un carrito de compras y agregamos productos
carrito = CarritoDeCompras()
carrito.agregar_producto(producto1)
carrito.agregar_producto(producto2)
carrito.agregar_producto(producto3)

# Calculamos el total de la compra
total_compra = carrito.calcular_total()
print("Total de la compra:", total_compra)
```

👉🏽En este ejemplo, hemos utilizado la composición para combinar varios objetos `Producto` y crear un objeto más grande y complejo `CarritoDeCompras`. La composición nos permite crear estructuras más

complejas y reutilizables en nuestros programas, lo que facilita la organización y el mantenimiento del código.

Herencia

En Python, la herencia es un mecanismo mediante el cual una clase puede heredar atributos y métodos de otra clase, llamada clase base o superclase. La clase que hereda se conoce como clase derivada o subclase. Este concepto es fundamental en la programación orientada a objetos y permite crear jerarquías de clases, donde las clases derivadas pueden extender y especializar la funcionalidad de la clase base.

La sintaxis para definir una clase con herencia en Python es la siguiente:

```
python
Copy code
class ClaseBase:
    # Definición de atributos y métodos de la clase base

class ClaseDerivada(ClaseBase):
    # Definición de atributos y métodos propios de la clase derivada
```

Al heredar de la clase base, la clase derivada adquiere todos los atributos y métodos de la clase base. Esto significa que la clase derivada puede utilizar y extender la funcionalidad definida en la clase base. Además, la clase derivada puede agregar nuevos atributos y métodos propios, lo que le permite especializarse y adaptarse a las necesidades específicas.

👉🏽Un ejemplo sencillo de herencia en Python sería el siguiente:

```
python
Copy code
class Animal:
    def __init__(self, nombre):
        self.nombre = nombre

    def hacer_sonido(self):
        pass
```

```
class Perro(Animal):
    def hacer_sonido(self):
        return "Guau!"

class Gato(Animal):
    def hacer_sonido(self):
        return "Miau!"
```

En este ejemplo, tenemos una clase base llamada Animal que tiene un atributo nombre y un método hacer_sonido(). Luego, creamos dos clases derivadas, Perro y Gato, que heredan de Animal. Cada clase derivada implementa su propio método hacer_sonido() para representar el sonido que hace cada animal.

La herencia es una herramienta poderosa en Python y nos permite crear jerarquías de clases que facilitan la reutilización de código y la organización de la funcionalidad en nuestros programas. Sin embargo, es importante usarla de manera adecuada y evitar jerarquías excesivamente complejas para mantener el código limpio y fácil de mantener.

Cierre

En conclusión, tanto la herencia como la composición son conceptos fundamentales en la programación orientada a objetos en Python. Ambos mecanismos permiten crear jerarquías de clases y reutilizar código de manera eficiente, pero tienen enfoques diferentes para lograrlo.

Es importante tener en cuenta que tanto la herencia como la composición son herramientas poderosas, pero deben utilizarse de manera adecuada y consciente. En general, se prefiere la composición sobre la herencia cuando sea posible, ya que la composición promueve una mayor flexibilidad y evita problemas de dependencias complejas.

En resumen, la elección entre herencia y composición dependerá de la estructura y las necesidades específicas de cada proyecto. Al entender estas dos técnicas, podemos diseñar aplicaciones más robustas, flexibles y fáciles de mantener en Python.

Referencias

- https://barcelonageeks.com/herencia-y-composicion-en-python/

¡Muchas gracias!

Nos vemos en la próxima unidad 👋