Clase 6: Uso de `super()` en Python 🚀

La función `super()` en Python es una herramienta poderosa que permite acceder a métodos y atributos de la superclase (o clase padre) desde una subclase (o clase hija). Esto facilita la extensión de funcionalidades sin necesidad de nombrar explícitamente la superclase, lo cual es especialmente útil en jerarquías de clases complejas.

1. Conceptos Básicos 📚

- **Atributos:** Son características que definen a una clase. Por ejemplo, en una clase `Persona`, podríamos tener atributos como `nombre`, `edad`, y `carné de identidad`.
- **Métodos:** Son acciones que puede realizar una clase. Por ejemplo, una `Persona` puede tener métodos como `saludar()`, `despedirse()`, y `hablar()`.
- **Constructor:** Es un método especial que se utiliza para inicializar los atributos de una clase. En Python, se define mediante `__init__()`.

2. Ejemplo de Herencia y Uso de `super()` 👨‍🎓

A continuación, desglosamos un ejemplo básico:

Definición de Clases

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

    def saludar(self):
        return f"Hola, soy {self.nombre}."

class Estudiante(Persona):  # Clase hija
    def __init__(self, nombre, edad, student_id):
        super().__init__(nombre, edad)  # Llama al constructo
r de la clase padre
        self.student_id = student_id

    def saludar(self):  # Sobrescribiendo el método
        return f"Hola, soy {self.nombre}, y mi ID de estudian
te es {self.student_id}."
```

3. Ejemplo de Uso de Clases 📦

Creación de un Estudiante

```
# Creación de un objeto de la clase Estudiante
estudiante1 = Estudiante("Ana", 20, "12345")

# Llamando al método saludar
print(estudiante1.saludar())
```

Salida:

```
Hola, soy Ana, y mi ID de estudiante es 12345.
```

4. Ejemplo de Herencia Multinivel 🌳

Podemos agregar un nivel más de herencia:

```
class SerVivo:  # Clase base
    def __init__(self, nombre):
        self.nombre = nombre

class Persona(SerVivo):  # Hereda de SerVivo
    def __init__(self, nombre, edad):
        super().__init__(nombre)
        self.edad = edad

class Estudiante(Persona):  # Hereda de Persona
    def __init__(self, nombre, edad, student_id):
        super().__init__(nombre, edad)  # Llama al constructo
r de Persona
        self.student_id = student_id
```

5. Ejercicios Prácticos 💻

Ejercicio 1: Crear una clase `Empleado`

1. Define una clase base llamada `Empleado` con atributos `nombre` y `salario`.
2. Crea una subclase `Gerente` que herede de `Empleado` y añada el atributo `departamento`.
3. Implementa un método `presentar()` en ambas clases que muestre información sobre el empleado o el gerente.

Solución:

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

```
    def presentar(self):
        return f"Soy {self.nombre} y mi salario es {self.sala
rio}."

class Gerente(Empleado):
    def __init__(self, nombre, salario, departamento):
        super().__init__(nombre, salario)
        self.departamento = departamento

    def presentar(self):
        return f"Soy {self.nombre}, gerente del departamento
de {self.departamento}, y mi salario es {self.salario}."
```

Ejercicio 2: Comparar métodos en subclases

1. Define dos subclases de `Persona`: `Estudiante` y `Profesor`.
2. Ambos deben implementar el método `saludar()`, pero con mensajes diferentes.

Solución:

```
class Profesor(Persona):
    def saludar(self):
        return f"Hola, soy el profesor {self.nombre}."

estudiante1 = Estudiante("Ana", 20, "12345")
profesor1 = Profesor("Dr. Smith", 45)

print(estudiante1.saludar())  # Saludo de Estudiante
print(profesor1.saludar())     # Saludo de Profesor
```

Conclusión 🏁

El uso de `super()` es fundamental en la programación orientada a objetos, ya que permite mantener el código limpio y eficiente al evitar redundancias. Además, fomenta la reutilización de código y mejora la mantenibilidad en aplicaciones complejas.
