Ejercicio Completo: Sistema de Gestión de Empleados en una Empresa

Contexto del Ejercicio:

Vamos a crear un sistema para gestionar empleados en una empresa. Cada empleado tendrá un nombre, un salario y una categoría (por ejemplo, gerente, empleado regular). También necesitaremos calcular el salario anual, realizar validaciones sobre los datos y mostrar las diferentes subclases de empleados que podemos tener en el sistema.

Objetivos:

- Definir diferentes tipos de métodos (instancia, clase, estático).
- Usar el decorador `@property` para controlar el acceso y modificación de algunos atributos.
- Definir decoradores personalizados.
- Implementar los métodos dunder `__repr__`, `__name__` y `__subclasses__()` para interactuar con las clases y los objetos.
- Mostrar cómo estos conceptos pueden relacionarse con herencia y polimorfismo.

Paso 1: Definir la Clase Base `Empleado`

Primero, creamos una clase base `Empleado` con los atributos comunes, un método para calcular el salario anual, y otro para validar los datos. Usamos `@property` para controlar el acceso al salario y al nombre.

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

    # Usamos @property para controlar el acceso y modificación del salario
    @property
    def salario(self):
        return self._salario

    @salario.setter
    def salario(self, valor):
        if valor < 0:
            raise ValueError("El salario no puede ser negativo")
        self._salario = valor

    # Método de instancia para calcular el salario anual
    def salario_anual(self):
        return self.salario * 12

    # Método estático para validar si un nombre es válido (solo letras)
    @staticmethod
    def nombre_valido(nombre):
        return nombre.isalpha()

    # Método de clase para crear un empleado desde una cadena "nombre-salario"
    @classmethod
    def desde_cadena(cls, cadena):
        nombre, salario = cadena.split('-')
```

```
        return cls(nombre, float(salario))

    # Método dunder __repr__ para representar el objeto como cadena
    def __repr__(self):
        return f"Empleado(nombre={self.nombre}, salario={self.salario})"
```

Explicación:

1. **`@property` y su setter** controlan el acceso y validación del atributo `salario`. Evitamos que se asigne un salario negativo.
2. **Método de instancia `salario_anual()`**: Calcula el salario anual multiplicando el salario mensual por 12.
3. **Método estático `nombre_valido()`**: Verifica que el nombre del empleado esté compuesto solo de letras.
4. **Método de clase `desde_cadena()`**: Crea una instancia de `Empleado` a partir de una cadena de texto, aplicando herencia polimórfica si es necesario.
5. **Método `__repr__()`**: Devuelve una representación clara del objeto, útil para depuración.

Paso 2: Subclases para Tipos de Empleados

Creamos subclases que sobrescriben algunos métodos de `Empleado`, añadiendo comportamiento específico para diferentes tipos de empleados.

```
class Gerente(Empleado):
    def salario_anual(self):
        # Los gerentes tienen un bono del 10% sobre el salario anual
        salario_base = super().salario_anual()
        return salario_base * 1.1

    def __repr__(self):
        return f"Gerente(nombre={self.nombre}, salario={self.salario})"

class EmpleadoRegular(Empleado):
    def __repr__(self):
        return f"EmpleadoRegular(nombre={self.nombre}, salario={self.salario})"
```

Explicación:

1. **Sobrescritura de `salario_anual()` en `Gerente`**: Calcula el salario anual con un bono del 10%, usando `super()` para llamar al método de la clase base.
2. **Sobrescritura de `__repr__()`** en ambas subclases para diferenciarlas claramente cuando imprimimos los objetos.

Paso 3: Decoradores Personalizados

Creamos un **decorador personalizado** para registrar cada vez que se calcula el salario de un empleado, usando un contador simple.

```
# Decorador personalizado para registrar el cálculo del salario
def registrar_calculo(func):
    def wrapper(*args, **kwargs):
        print(f"Calculando salario para {args[0].nombre}...")
```

```
        return func(*args, **kwargs)
    return wrapper
```

Aplicamos este decorador al método `salario_anual()` en la clase `Empleado`.

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

    # Decoramos el método salario_anual
    @registrar_calculo
    def salario_anual(self):
        return self.salario * 12
```

Paso 4: Uso de Métodos Dunder y Subclases

Podemos explorar las subclases de `Empleado` utilizando `__subclasses__()` para ver todas las subclases registradas en el sistema.

```
print(Empleado.__subclasses__())
# Salida: [<class '__main__.Gerente'>, <class '__main__.EmpleadoRegular'>]
```

Paso 5: Interacción Completa con el Sistema

1. Crear Empleados y Calcular el Salario

```
# Crear instancias de diferentes tipos de empleados
gerente = Gerente("Ana", 5000)
empleado_regular = EmpleadoRegular("Pedro", 3000)

# Calcular el salario anual de cada uno
print(gerente.salario_anual())         # Salida: Calculando salario... 66000.0
print(empleado_regular.salario_anual())# Salida: Calculando salario... 36000.0
```

2. Usar el Método de Clase para Crear un Empleado

```
empleado_desde_cadena = Empleado.desde_cadena("Carlos-4000")
print(empleado_desde_cadena)  # Salida: Empleado(nombre=Carlos, salario=4000.0)
```

3. Validar el Nombre con el Método Estático

```
print(Empleado.nombre_valido("Ana"))  # Salida: True
print(Empleado.nombre_valido("Ana123"))  # Salida: False
```

Conceptos Aplicados

1. **Decoradores**: Creamos un decorador personalizado para registrar el cálculo del salario de los empleados.
2. **Métodos de Clase**: El método `desde_cadena()` nos permite crear empleados a partir de una cadena.

3. **Métodos Estáticos**: Usamos `nombre_valido()` para validar nombres sin necesidad de instanciar la clase.
4. **Métodos de Instancia**: `salario_anual()` es un método de instancia que calcula el salario anual de un empleado.
5. **`@property`**: Controlamos el acceso y validación del salario mediante `@property`.
6. **Métodos Dunder**: Utilizamos `__repr__()` para mejorar la representación de los objetos y `__subclasses__()` para listar todas las subclases de `Empleado`.
7. **Herencia y Polimorfismo**: Implementamos herencia entre `Empleado`, `Gerente`, y `EmpleadoRegular`, y sobrescribimos métodos en las subclases para aplicar polimorfismo.