Informe Final del Trabajo Práctico

Materia: Programación Avanzada
Tema: Decoradores y conceptos avanzados de POO aplicados a un sistema de gestión de empleados
Docente: Gianluca Piriz
Integrantes:

- Lucas Avila
- Ciro Soto
- Denisse Sajama
- Lucas Capocasa

Introducción

En este trabajo abordamos conceptos avanzados de Programación Orientada a Objetos (POO) en Python, haciendo foco en el uso de decoradores como mecanismo para añadir funcionalidad transversal. Como producto final desarrollamos un sistema de gestión de empleados que permite cargar empleados, visualizarlos, aumentar salarios y guardar la información en un archivo JSON.

Durante el desarrollo se aplicaron principios clave del diseño orientado a objetos: herencia, encapsulamiento, clases abstractas, propiedades y polimorfismo. También se implementó un decorador personalizado para registrar acciones relevantes, como aumentos salariales o persistencia de datos.

Fundamentación y Motivación

Los decoradores en Python nos permiten modificar o extender el comportamiento de funciones sin alterar su definición original. Es por esto que decidimos integrar un decorador central que registrara las operaciones importantes del sistema, dejando el código de cada función limpio.

Además, reforzamos conceptos vistos en clase como:

- Encapsulamiento y validación de atributos
- Herencia jerárquica con clases abstractas

También nos propusimos simular un sistema real de desarrollo que pueda actualizarse a futuro.

Descripción del sistema desarrollado

Arquitectura general

La arquitectura del sistema se basa en una clase abstracta Empleado, que define el atributo común nombre y salario, así como el método abstracto **mostrar_info()**. A partir de esta clase se crean dos subclases:

- Gerente: incluye un bono privado adicional.
- Desarrollador: contiene un atributo para la tecnología principal.

Ambas redefinen **mostrar_info()** y sobrecargan el método **__str__()** para mejorar la legibilidad en consola. Se respeta el principio de encapsulamiento: el salario se mantiene protegido y se manipula mediante un setter con validación.

Decorador como herramienta transversal

El decorador **@log_operacion** encapsula el registro de acciones relevantes como guardar o aumentar salarios. Cada vez que se ejecuta una función decorada, se imprime automáticamente en consola la hora y el nombre de la función ejecutada.

Se aplicó el módulo **functools.wraps** para preservar los metadatos de la función original, cumpliendo buenas prácticas.

Persistencia y serialización

La persistencia se implementó utilizando el formato **JSON**. Cada clase implementa un método **to_dict()** que permite serializar sus atributos. La función **guardar_empleados()** escribe los datos al archivo **empleados.json**, y **cargar_empleados()** los recupera generando instancias según el tipo especificado.

Este mecanismo permite mantener los datos entre sesiones sin requerir una base de datos, facilitando la portabilidad del sistema.

Interfaz por consola

La función **menu()** organiza un menú simple y efectivo que permite al usuario:

1. Ver empleados cargados
2. Agregar Gerente
3. Agregar Desarrollador
4. Aumentar salario a todos
5. Guardar y salir

Los datos ingresados por el usuario se validan y se manejan errores comunes como entradas inválidas, sin interrumpir el sistema.

Detalles técnicos

Encapsulamiento

El atributo _**salario** se declara como protegido y se accede mediante una propiedad que valida que el nuevo valor sea mayor a cero. Esta técnica permite proteger los datos internos y aplicar lógica adicional en el acceso.

Método especial **__str__()**

Aunque el método abstracto principal es **mostrar_info(),** todas las clases también implementan **__str__()**, lo que permite imprimir directamente las instancias para una mejor experiencia de usuario.

Decorador con **@wraps**

El decorador **log_operacion** se implementa usando **@wraps** para mantener el nombre y datos originales de la función decorada.

Manejo de errores

El sistema incluye control de errores para:

- Archivos ausentes (manejo de **FileNotFoundError**)
- Conversión de tipos numéricos (**ValueError**)
- Validación de entradas del usuario

Reflexiones y mejoras futuras

Este proyecto fue una oportunidad para integrar múltiples conceptos de programación avanzada en un sistema básico. Los decoradores fueron una gran herramienta para su implementación.

Aprendimos a estructurar un sistema escalable, mantener un menú interactivo funcional y aplicar abstracción de forma efectiva.

Posibles mejoras futuras:

- Reemplazar el menú por una API REST con FastAPI
- Incorporar una base de datos SQLite
- Añadir autenticación con decoradores para restringir accesos
- Crear una interfaz gráfica con Tkinter o PyQt

Conclusión

Este trabajo permitió nos permitió aplicar lo aprendido en un proyecto real. Construimos un sistema útil, claro y fácil de mantener. El uso de decoradores fue una gran herramienta para practicar cómo extender funciones, sin repetir código.

Además, trabajar en equipo nos ayudó a organizarnos mejor, dividir tareas y pensar en conjunto cómo resolver los problemas.

Muchas gracias.