Lectura

Decoradores y Generadores

Python Intermedio

Introducción

En esta lectura se abordan dos herramientas de Python: los **decoradores** y los **generadores**.

En el contexto de la programación en Python, estas herramientas son fundamentales, porque permiten mejorar la modularidad y la eficiencia del código. Los decoradores son funciones que modifican el comportamiento de otras funciones, sin alterar su implementación original, mientras que los generadores facilitan la creación de secuencias eficientes, al producir valores de manera perezosa.

El dominio de estas técnicas resulta especialmente útil en entornos de desarrollo profesional, donde es necesario escribir programas más **claros, escalables** y **fáciles de mantener**. Por ejemplo, los decoradores se aplican ampliamente en frameworks web para validar accesos o registrar información, mientras que los generadores son esenciales en procesamiento de datos y en escenarios que requieren manejar volúmenes grandes de información, sin comprometer el rendimiento.

El estudio de estas dos herramientas no solo enriquecerá tu conocimiento del lenguaje, sino que también te proporcionará habilidades prácticas de gran utilidad en la resolución de problemas reales, preparándote para abordar proyectos más complejos con un enfoque estructurado y eficiente.

Decoradores:

Son funciones que reciben otra función como **argumento** y devuelven una nueva función que amplía o modifica el comportamiento de la original. Se usan con la sintaxis @nombre_decorador.

Su sintaxis es:

```
@nombre_del_decorador
def mi_funcion():
```

Sus características son:

- Son una forma de “envolver” funciones para añadirles funcionalidades, **sin modificar su código original**.
- Son muy usados en **frameworks web** (Flask, Django) y en **buenas prácticas** como:
  - Validar datos antes de ejecutar una función.
  - Medir tiempos de ejecución.
  - Mostrar mensajes de depuración.
- Funcionan gracias al concepto de que en Python **las funciones son objetos** (se pueden pasar como parámetros o devolver de otras funciones).

A continuación se presentan algunos ejemplos:

- El siguiente ejemplo **muestra un decorador básico:**

```python
def mi_decorador(funcion):
    def nueva_funcion():
        print("Antes de la función")
        funcion()
        print("Después de la función")
    return nueva_funcion

@mi_decorador
def saludar():
    print("Hola!")

saludar()
```

- ➔ mi_decorador recibe como parámetro la función saludar.
- ➔ Dentro de nueva_funcion, se imprime un mensaje antes, y otro después de ejecutar saludar().
- ➔ Cuando escribimos @mi_decorador, Python automáticamente transforma saludar() en mi_decorador(saludar).

- El siguiente ejemplo **presenta un decorador que muestra el tiempo de ejecución:**

```python
import time

def medir_tiempo(funcion):
    def wrapper():
        inicio = time.time()
        funcion()
        fin = time.time()
        print(f"Tiempo: {fin - inicio:.4f} segundos")
    return wrapper

@medir_tiempo
def contar():
    suma = 0
    for i in range(1000000):
        suma += i
    print("Suma lista")

contar()
```

- ➔ medir_tiempo calcula cuánto tarda en ejecutarse una función.
- ➔ contar hace un ciclo grande, y el decorador mide cuánto tardó.
- ➔ Así, el decorador nos da información extra **sin modificar** contar().

- El siguiente ejemplo **presenta un decorador que recibe argumentos:**

```python
def repetir(n):
    def decorador(funcion):
        def wrapper():
            for _ in range(n):
                funcion()
        return wrapper
    return decorador

@repetir(3)
def hola():
    print("Hola mundo!")

hola()
```

➔ @repetir(3) indica que la función hola() debe ejecutarse 3 veces.

Para saber más

Los casos de uso prácticos de los **decoradores** son:

- **Validaciones:** comprobar que un usuario tenga permisos antes de ejecutar una función.
- **Logs:** registrar cada vez que se llama a una función.
- **Pruebas:** simular acciones repetidas, sin duplicar código.

Generadores:

Funciones especiales que producen **valores** de manera secuencial y "perezosa" (uno en uno) usando la palabra clave ***yield***. No almacenan toda la secuencia en memoria, lo que los hace eficientes para grandes volúmenes de datos.

Sus características son:

- Un generador es como una **función normal**, pero en vez de return usa yield.
- Con yield, la función **recuerda en qué parte se quedó** y puede continuar, la siguiente vez que se llame.
- Se usan frecuentemente cuando trabajamos con **muchos datos**, porque no guardan toda la información en la memoria, sino que producen cada valor bajo demanda.
- Se pueden recorrer con for, igual que una lista, pero **sin tener que construir toda la lista antes**.

A continuación se presentan algunos ejemplos de generadores:

- El siguiente ejemplo muestra **un generador básico:**

```
1    def contar():
2        yield 1
3        yield 2
4        yield 3
5
6    for numero in contar():
7        print(numero)
```

➔ contar() es una función generadora porque usa yield.

➔ Cada vez que se ejecuta el bucle, Python obtiene el siguiente valor: primero 1, luego 2, después 3.

➔ Diferencia con return: si hubiéramos usado return, la función terminaría en el primer valor. Con yield, se **pausa y continúa**.

- El siguiente ejemplo **muestra un generador con un bucle:**

```
def cuadrados(n):
    for i in range(n):
        yield i ** 2

for valor in cuadrados(5):
    print(valor)
```

➔ cuadrados(5) va devolviendo los números al cuadrado: 0, 1, 4, 9, 16.

➔ Aquí yield evita que se cree toda la lista de cuadrados en memoria.

➔ Resulta muy útil en el caso de que n sea un número grande.

- El siguiente ejemplo **muestra un generador infinito controlado con break:**

```
def numeros_infinito():
    n = 1
    while True:
        yield n
        n += 1

for numero in numeros_infinito():
    if numero > 5:
        break
    print(numero)
```

- ➔ Este generador nunca se detiene por sí solo, porque el while True lo hace infinito.
- ➔ En el for, usamos un break para detenerlo después de 5 números.
- ➔ Esto muestra que los generadores también pueden ser **flujos continuos de datos**.

Para saber más

Los casos de uso práctico de **generadores** son:

- Leer archivos grandes línea por línea, sin cargarlos completos en memoria.
- Procesar datos en tiempo real (ej. un flujo de sensores o logs).
- Generar secuencias infinitas (ej. números primos, secuencia de Fibonacci).

Conclusión

Tanto los decoradores como los generadores son elementos clave en Python, que facilitan la escritura de código más limpio y eficiente. Los decoradores permiten añadir funcionalidades sin alterar el código original, mientras que los generadores optimizan el manejo de grandes volúmenes de datos, haciendo que la programación sea más efectiva y manejable.

Su práctica constante te permitirá integrar estas herramientas en tus propios proyectos, fortaleciendo tu capacidad para crear soluciones modulares, reutilizables y con un mejor desempeño. Estas técnicas representan un paso importante hacia un estilo de programación más profesional y sientan las bases para trabajar con temas avanzados como la programación orientada a objetos, el diseño de sistemas más complejos y la construcción de aplicaciones robustas.

Elaboró: Enrique Quezada Próspero

Contenido: Enrique Quezada Próspero

DI: Génesis Equihua

Referencias:

1. Campesato, O. (2023). *Intermediate Python*. Mercury Learning and Information.
2. Python Software Foundation. (2025a). *Built-in Functions*. Recuperado el 05 de agosto de 2025, de: https://docs.python.org/3/library/functions.html
3. Python Software Foundation. (2025b). *functools — Higher-order functions and operations on callable objects*. Recuperado el 05 de agosto de 2025, de: https://docs.python.org/3/library/functools.html
4. Python Software Foundation. (2025c). *itertools — Functions creating iterators for efficient looping*. Recuperado el 05 de agosto de 2025, de: https://docs.python.org/3/library/itertools.html
5. Python Software Foundation. (2025d). *The Python Tutorial – Functional Programming*. Recuperado el 05 de agosto de 2025, de: https://docs.python.org/3/tutorial/controlflow.html#lambda-expressions