

PYTHON NIVEL INTERMEDIO



INFOTEC

Centro de investigación e innovación
en tecnologías de la información y
comunicación

TÍTULO DEL PROYECTO

**Lambda, decoradores, y
generadores.**

PRESENTA

Armando Gabriel Jacinto López

DOCENTE

Vladimir Morales Pérez

Fecha: 08/11/2025

Introducción

Este reporte documenta el desarrollo y la aplicación de un conjunto de técnicas de programación avanzada en Python, centrándose en el procesamiento eficiente de un conjunto de datos de temperaturas. El proyecto sirve como una demostración práctica de cuatro pilares clave que optimizan la escritura y ejecución del código:

1. **Generadores:** Son funciones que devuelven un iterador utilizando la palabra clave `yield`. Se implementan aquí para la ingesta de datos de manera *perezosa* (lazy) y eficiente en memoria, permitiendo procesar secuencias de datos potencialmente grandes sin cargarlas completamente.
2. **Funciones de Orden Superior y Lambda:** Las HOFs, como `map`, `filter` y `reduce`, son funciones que operan sobre otras funciones. Se combinan con las funciones **Lambda** (funciones anónimas de una sola expresión) para aplicar transformaciones, filtrados y reducciones a colecciones de datos de manera declarativa, siguiendo el paradigma de programación funcional.
3. **Decoradores:** Permiten envolver y extender la funcionalidad de una función existente sin modificar su código fuente. En este proyecto, se implementa un decorador personalizado para agregar una capa de auditoría y medición de rendimiento (meta-programación) a una función de salida.

El objetivo principal es ilustrar cómo la combinación de estos conceptos resulta en un código más **conciso, legible** y con una gestión de recursos significativamente más **eficiente**, aplicado al análisis básico de temperaturas para la identificación de alertas y el cálculo de promedios.

Desarrollo

El código se estructura para manejar un flujo de datos desde su generación hasta su procesamiento final, utilizando un enfoque funcional siempre que es posible.

1. Generadores para la Ingesta de Datos

Para manejar la secuencia de datos de manera eficiente en memoria, se implementó la función **Generador leer_temparaturas**.

- **Uso de yield:** En lugar de devolver una lista completa, esta función utiliza la palabra clave `yield` para pausar y reanudar su ejecución, devolviendo los registros uno a uno. Esto es ideal para conjuntos de datos grandes, ya que reduce significativamente el consumo de memoria.
- **Flujo:** Los datos originales se iteran y se almacenan en la lista `registros`.

```
# Generador de datos
tempuraturas_datos = [("CDMX", 26), ("Monterrey", 34), ("Toluca", 19), ("Cancún", 38), ('Oaxaca', 30)]
registros = []

def leer_temparaturas(registros):
    for registro in registros:
        yield registro

for temperatura in leer_temparaturas(tempuraturas_datos):
    registros.append(temperatura)
```

Ilustración 1 Generador de datos

2. Funciones de Orden Superior y Lambda

Las **Funciones de Orden Superior** son aquellas que toman otras funciones como argumentos, o devuelven una función como resultado. En combinación con las funciones **Lambda** (funciones anónimas pequeñas), permiten un manejo de datos potente y expresivo.

- **filter() y lambda:**
 - Se utiliza `filter()` para seleccionar únicamente aquellos registros donde la temperatura sea igual o superior a 30°C.

- La función lambda registro: registro[1] >= 30 actúa como el predicado de filtrado, operando sobre el segundo elemento ([1], la temperatura) de cada tupla de registro.

```
# Filtrado con filter() y Lambda
temperaturas_mayores = list(filter(lambda registro: registro[1] >= 30, registros))
```

Ilustración 2 Uso de la función Filter para filtrar temperaturas mayores a 30°C

- **map() y lambda:**

- Se utiliza map() para transformar los datos filtrados (temperaturas_mayores) en una nueva estructura.
- La función lambda genera mensajes de alerta con formato de cadena (f-string), proporcionando una salida legible para el usuario.

```
# Transformacion con map() y Lambda
alertas = list(map(lambda registro: f'Alerta de calor en {registro[0]}: {registro[1]}°C.', temperaturas_mayores))
```

Ilustración 3 Transformación de datos usando la función Map

- **sorted() y lambda:**

- Se utiliza sorted() para ordenar los registros seleccionados. Aunque el código utiliza sorted(temperaturas_mayores) sin un argumento key, en escenarios más complejos, una función lambda podría usarse aquí para ordenar por un campo específico (e.g., key=lambda x: x[1] para ordenar por temperatura).

```
# Ordenamiento con sorted() y Lambda
temperaturas_ordenadas = list(sorted(temperaturas_mayores))
```

Ilustración 4 Función Sorted para ordenar las temperaturas de forma ascendente

- **reduce() y lambda (Resumen):**

- reduce() se utiliza para calcular un único valor a partir de una lista. Requiere la importación de functools.
- La función lambda x, y: x + y realiza la suma acumulada de todos los valores de temperatura en la lista, permitiendo posteriormente calcular el promedio.

```
# Resumen con reduce()
temperaturas_valores = list(map(lambda valor: valor[1], temperaturas_ordenadas))
suma_temperaturas = reduce(lambda x, y: x + y, temperaturas_valores)

promedio_temperaturas = suma_temperaturas / len(temperaturas_valores)
```

Ilustración 5 Uso de la función Reduce para obtener el promedio de las temperaturas

3. Decoradores Personalizados

El **Decorador** auditar_funcion es una forma de meta-programación que permite modificar o mejorar el comportamiento de una función sin alterar su código fuente.

```
# Decorador personalizado
def auditar_funcion(funcion):
    # Variable para contar el numero de llamadas
    n = 0
    # Preversar metadatos de la funcion original
    @wraps(funcion)
    # El decorador puede aceptar argumentos posicionales o por nombre
    def wrapper(*args, **kwargs):
        # Inicio de ejecucion
        inicio = time.time()
        # Imprimir nombre de la funcion
        print('\n' + '-' * 40)
        print(f'Nombre de la funcion: {funcion.__name__}')
        # Habilitar el retorno de valores para la funcion wrapper()
        resultado = funcion(*args, **kwargs)
        # Aumentar el numero de llamadas de la funcion
        nonlocal n
        n += 1
        # Fin de ejecucion
        fin = time.time()
        # Veces que la funcion fue llamada
        print(f'Veces que la funcion fue llamada: {n}')
        # Tiempo de ejecucion
        print(f'Tiempo de ejecucion: {fin - inicio:.4f}')
        return resultado

    return wrapper
```

Ilustración 6 Decorador para auditar funciones

- **Estructura:** Es una función que recibe otra función (funcion) y devuelve una nueva función (wrapper).
- **Funcionalidad:**
 - Mide el tiempo de ejecución (time.time()).
 - Utiliza la palabra clave **nonlocal** para mantener un contador (n) de cuántas veces ha sido llamada la función original, asegurando que el contador no sea local al wrapper sino a la función externa (auditar_funcion).
 - Utiliza `@wraps(funcion)` (importado de `functools`) para preservar los metadatos (como el nombre de la función `__name__`) de la función original, lo cual es crucial para la depuración.
- **Aplicación:** Se aplica a la función imprimir mediante la sintaxis `@auditar_funcion`. Cada vez que se llama a imprimir, el decorador envuelve la ejecución, audita el tiempo y el número de llamadas, e imprime estos datos.

Conclusión

La implementación demostró con éxito el potencial de las herramientas de programación funcional y modularidad de Python.

1. **Eficiencia y Legibilidad:** El uso de Generadores y funciones de primer orden con funciones Lambda resultó en un código más conciso para el procesamiento de datos (filtrado, mapeo y resumen), reduciendo la necesidad de bucles for explícitos.
2. **Modularidad y Reutilización:** El Decorador `auditar_funcion` probó ser una solución elegante y reutilizable para añadir funcionalidades de *audición* y medición de rendimiento a cualquier otra función del proyecto con una sola línea de código (`@auditar_funcion`).

Como resultado, se logró procesar los datos de temperatura, generar alertas y calcular el promedio de las temperaturas críticas, todo dentro de una estructura de código bien definida que maximiza la eficiencia y la expresividad del lenguaje.