



1B. EJERCICIO. LAMBDA, DECORADORES, GENERADORES

Unidad 1. Funciones para modularidad

Datos

Nombre: Keith Alexis Gutiérrez Ibarra

Docente: Adolfo Aldair Duque Borja

Fecha de elaboración: 20/11/2025

Introducción

En este trabajo se desarrollará un ejercicio para realizar un sistema de monitoreo ambiental en Python, a través de un módulo encargado de procesar alertas de temperatura de diferentes ciudades, este sistema permitirá filtrar, transformar, ordenar y analizar datos utilizando herramientas como funciones lambda, funciones de orden superior, decoradores y generadores.

El propósito principal de este ejercicio es fortalecer la modularidad y eficiencia del código, haciendo uso adecuado del manejo de memoria mediante generadores, la automatización de procesos con decoradores y el procesamiento funcional de datos con `map()`, `filter()`, `sorted()` y `reduce()`, consolidando así los conceptos fundamentales de programación y estructurada en Python.

Desarrollo

Ejercicio 1

a) Análisis del problema: Para el ejercicio se nos piden las siguientes condiciones que debe tener el Código.

- Crea una función generadora `leer_temperaturas()` que devuelva, una a una, tuplas con el formato: ("Ciudad", temperatura)
- Hacer un filtrado con `filter()` y `lambda` los registros que tengan temperaturas mayores o iguales a 30°C.
- Realizar una transformación con `map()` y `lambda` convirtiendo cada tupla a un string con el formato: "Alerta de calor en [Ciudad]: [Temperatura]°C"
- Ordenar con `sorted()` y `key=lambda` los registros filtrados por temperatura en orden descendente.
- Hacer un Resumen con `reduce()` donde se calcula el promedio de temperaturas mayores a 30°C y usa `reduce()` desde el módulo `functools`.
- Crea un decorador llamado `@auditar_funcion` que Imprima el nombre de la función que se está ejecutando, cuente cuántas veces ha sido llamada y opcionalmente muestre la duración de ejecución
- Imprimir la lista ordenada de alertas e imprimir el promedio de temperaturas en formato: "Temperatura promedio de alertas: 33.5°C"

b) Diseño del algoritmo

Se importan las librerías necesarias (time, functools) y se define la estructura de datos base.

- a) Se inicializa una lista interna de tuplas que contiene pares de datos (Ciudad, Temperatura).

Se crea una función decoradora llamada auditoria_funcion para medir el rendimiento y conteo de llamadas.

- a) Dentro del decorador se define una función que gestiona un contador de ejecuciones propio.
- b) Se captura el tiempo de inicio donde se ejecuta la función original recibida y se captura el tiempo final para calcular la duración.
- c) Se imprime el nombre de la función, el número de llamada actual y el tiempo transcurrido.

Se crea una función generadora llamada leer_temperaturas (decorada) para el manejo eficiente de memoria.

- a) Recibe la lista de datos como parámetro e itera sobre ella.
- b) Utiliza yield para devolver cada tupla de temperatura una por una bajo demanda, en lugar de procesar toda la lista a la vez.

Creación de la función monitoreo (decorada) que aplica la parte de procesamiento de los datos.

- a) Se utiliza filter junto con una expresión lambda para extraer únicamente los estados con temperaturas mayores a 30°C provenientes del generador.
- b) Se usa sorted para ordenar la lista resultante de forma descendente (de mayor a menor temperatura).
- c) Se aplica map para transformar la lista de datos en una lista de cadenas de texto con el formato de "Alerta de calor".
- d) Se emplea reduce para acumular la suma de las temperaturas filtradas y posteriormente calcular el promedio aritmético.
- e) La función retorna el promedio calculado y la lista de mensajes de alerta.

Se define la función principal main (decorada) que inicializa la salida al usuario.

- a) Recibe como parámetros el promedio y la lista de transformación generados anteriormente.
- b) Ejecuta un bucle para recorrer e imprimir cada mensaje de alerta generado.

- c) Imprime el promedio final de las temperaturas y muestra el mensaje de fin del programa.

Se realiza la llamada a la función main, usando resultados retornados por la función monitoreo.

c) Código en Python

```

Welcome  Lambda, decoradores, generadores.py X .py
Lambda, decoradores, generadores.py > ...
1  # Importar time
2  import time
3  # Importar reduce desde functools
4  from functools import reduce
5  # Lista interna de tuplas con estado y temperatura
6  lista = [
7      ("Guanajuato", 38),
8      ("Jalisco", 35),
9      ("Michoacan", 25),
10     ("Puebla", 20),
11     ("Queretaro", 15),
12     ("Quintana Roo", 42),
13     ("San Luis Potosi", 32),
14     ("Sinaloa", 36),
15 ]
16 # Decorador para auditar funciones
17 def auditoria_funcion(funcion):
18     def funcion_tiempo(*args, **kwargs):
19         # Incrementar el contador de llamadas
20         funcion_tiempo.conteo += 1
21         # Imprimir información de auditoría
22         print(f"\nEjecucion de la función: {funcion.__name__}")
23         print(f"Número de llamadas: {funcion_tiempo.conteo}")
24         # Medir el tiempo de ejecución
25         inicio = time.time()
26         resultado = funcion(*args, **kwargs)
27         fin = time.time()
28         tiempo = fin - inicio
29         # Imprimir información de tiempo de ejecución
30         print(f"Tiempo de ejecución de {funcion.__name__}: {tiempo:.6f} segundos")
31         return resultado
32
33     # Inicializar el contador de llamadas
34     funcion_tiempo.conteo = 0
35     return funcion_tiempo
36
37 # Generador de temperaturas de diferentes estados
38 @auditoria_funcion
39 def leer_temperaturas(datos):
40     # Iterar sobre los datos y yield cada temperatura
41     for temperatura in datos:
42         yield temperatura
43     print("Lectura de temperaturas finalizada.")
44
45 # Función principal para monitorear temperaturas
46 @auditoria_funcion
47 def monitoreo():
48     # Filtrar estados con temperaturas mayores a 30 grados Celsius usando filter y lambda
49     calor = list(filter(lambda x: x[1] > 30, leer_temperaturas(lista)))
50     # Ordenar la lista resultante de mayor a menor temperatura usando sorted y lambda
51     orden = sorted(calor, key=lambda x: x[1], reverse=True)
52     # Transformar la lista en mensajes de alerta usando map y lambda
53     transformacion = list(map(lambda x: f"Alerta de calor en {x[0]}: {x[1]}°C", orden))
54     # Calcular el promedio de las temperaturas usando reduce y lambda
55     suma = reduce(lambda x, y: x + y[1], orden, 0)
56     # Calcular el promedio dividiendo la suma entre el número de elementos
57     promedio = suma / len(orden)
58     print("\nMonitoreo finalizado.")
59     return promedio, transformacion
60
61 # Función main para ejecutar el monitoreo y mostrar resultados
62 @auditoria_funcion
63 def main(promedio, transformacion):
```

```

63     for alerta in transformacion:
64         print(alerta)
65     print(f"\nEl promedio de la temperatura es: {promedio}")
66     print("Fin del programa.")
67
68 main(*monitoreo())
69

```

d) Ejecución y pruebas

Ejemplos del funcionamiento del código en diversos escenarios:

- Imprimir la lista ordenada de alertas

```

Alerta de calor en Quintana Roo: 42°C
Alerta de calor en Guanajuato: 38°C
Alerta de calor en Sinaloa: 36°C
Alerta de calor en Jalisco: 35°C
Alerta de calor en San Luis Potosi: 32°C

```

- Imprimir el promedio de temperaturas en formato:

```

El promedio de la temperatura es: 36.6
Fin del programa.

```

- Imprima el nombre de la función que se está ejecutando:

```

Ejecucion de la función: monitoreo

```

- Cuente cuántas veces ha sido llamada.

```

Número de llamadas: 1

```

- Opcional: muestre la duración de ejecución

```

Monitoreo finalizado.
Tiempo de ejecución de monitoreo: 0.000300 segundos

```

Conclusiones

En este trabajo se aplicaron conceptos fundamentales de programación para desarrollar un módulo capaz de procesar alertas de temperatura de forma eficiente integrando generadores, funciones lambda, decoradores y funciones de orden superior, lo que permitió filtrar, transformar y analizar los datos simulados de distintas ciudades, siendo modular y fácil de ampliar.

El uso de estas herramientas contribuye a mejorar el manejo de memoria, automatizar tareas mediante el decorador y organizar el procesamiento de información de manera más clara, este proyecto ayudó a reforzar la lógica de programación y demostró cómo el uso adecuado de estas técnicas permite crear soluciones más limpias, reutilizables y adaptables, favoreciendo un desarrollo de software mejor estructurado.