



[LCD] MÓDULO 4

Bienvenido al **Módulo 4** del curso **Lenguaje de Ciencia de Datos I.**

INTRODUCCIÓN

Estimado estudiante:

En el presente módulo aprenderás los principios de la programación funcional avanzada en Python, comprendiendo el uso de funciones lambda, map, filter, reduce, comprensiones, iteradores y generadores. Aprenderás cómo aplicar estos recursos para optimizar procesos, transformar colecciones y manejar grandes volúmenes de datos de manera eficiente y expresiva.

Te deseamos una excelente experiencia de aprendizaje en este curso.

;Muchos éxitos!

 TEMA 4 : PROGRAMACIÓN FUNCIONAL AVANZADA

 CIERRE

TEMA 4 : PROGRAMACIÓN FUNCIONAL AVANZADA

PROGRAMACIÓN FUNCIONAL AVANZADA

Llegó el momento para que te conviertas en el protagonista de tu aprendizaje.

- Descubre y analiza la información que se presenta en la lección interactiva para transformarla en nuevo conocimiento.

La programación funcional en Python permite escribir código más expresivo y eficiente para transformar y analizar grandes volúmenes de datos.





La programación funcional es un paradigma que trata a las funciones como ciudadanos de primera clase. En Python, este enfoque permite un código más expresivo y eficiente, especialmente útil en ciencia de datos para la transformación de colecciones, análisis de grandes volúmenes de datos y optimización de operaciones.

Funciones anónimas Lambda

Las funciones lambda representan una forma concisa y eficiente de definir operaciones simples en Python. Este tipo de funciones, también conocidas como funciones anónimas, permiten expresar cálculos o transformaciones en una sola línea de código, lo que las convierte en una herramienta esencial para el procesamiento rápido de datos y la programación funcional.



- Son funciones sin nombre y de una sola expresión.

- Se utilizan en situaciones donde definir una función con `def` sería innecesario.
- No deben usarse en operaciones demasiado complejas (mala práctica).



PROS

- Código más compacto.
- Ideal para operaciones rápidas y locales.



CONTRAS

- Dificultan la lectura si la lógica es extensa.
- No pueden contener múltiples sentencias.



Ejercicios

Los siguientes ejercicios permitirán poner en práctica el uso de funciones lambda para resolver tareas comunes en ciencia de datos. A través de ejemplos simples y aplicados, los estudiantes podrán observar cómo estas funciones contribuyen a escribir código más compacto, claro y eficiente, especialmente en operaciones de transformación y filtrado de datos.

EJERCICIO 1: AVANZADO EN DATA SCIENCE**EJERCICIO 2: CLASIFICACIÓN DE NOTAS CON FUNCIONES LAMBDA****EJERCICIO 3: COMPRENSIÓN DE LISTAS**

Este ejercicio muestra cómo una función lambda puede emplearse para realizar transformaciones rápidas sobre un conjunto de datos, optimizando el cálculo de precios en contextos de análisis o procesamiento de información comercial.

```
import pandas as pd
# Dataset simple
data = pd.DataFrame({
    "producto": ["A", "B", "C"],
    "precio": [100, 200, 300]
})
# Aplicar aumento de 15% con lambda
data["precio_final"] = data["precio"].apply(lambda x: x * 1.15)
print(data)
```

EJERCICIO 1: AVANZADO EN DATA SCIENCE**EJERCICIO 2: CLASIFICACIÓN DE NOTAS CON FUNCIONES LAMBDA****EJERCICIO 3: COMPRENSIÓN DE LISTAS**

Crear una función lambda que clasifique los valores de una lista de notas (≥ 11 aprobado, < 11 desaprobado).

```
# Lista de notas
notas = [8, 12, 15, 9, 10, 18]

# Función lambda para clasificar
clasificacion = list(map(lambda x: "Aprobado" if x >= 11 else "Desaprobado", notas))

# Mostrar resultados junto a las notas originales
```

```
resultado = list(zip(notas, clasificacion))
print(resultado)
```

Salida esperada:

```
[(8, 'Desaprobado'), (12, 'Aprobado'), (15, 'Aprobado'),
(9, 'Desaprobado'), (10, 'Desaprobado'), (18, 'Aprobado')]
```

👉 De esta manera, cada nota queda emparejada con su clasificación.

Este ejercicio busca reforzar la comprensión de cómo las expresiones lambda permiten escribir código más conciso, eficiente y funcional, sin necesidad de definir funciones tradicionales, favoreciendo así la transformación rápida de datos en Python.

EJERCICIO 1: AVANZADO EN DATA SCIENCE

EJERCICIO 2: CLASIFICACIÓN DE NOTAS CON FUNCIONES LAMBDA

EJERCICIO 3: COMPRENSIÓN DE LISTAS

Crea un programa en Python que, a partir de una lista de notas, clasifique cada valor como “Aprobado” si es mayor o igual a 11, o “Desaprobado” si es menor. Utiliza una comprensión de listas para realizar esta tarea de forma concisa y eficiente.

```
# Lista de notas
notas = [8, 12, 15, 9, 10, 18]

# Usando comprensión de listas para clasificar
resultado = [(n, "Aprobado" if n >= 11 else "Desaprobado") for n in notas]
print(resultado)
```

Salida esperada:

```
[(8, 'Desaprobado'), (12, 'Aprobado'), (15, 'Aprobado'),
(9, 'Desaprobado'), (10, 'Desaprobado'), (18, 'Aprobado')]
```

En este ejercicio se aplica el uso de comprensiones de listas como alternativa a las funciones lambda y map, reforzando la escritura de código claro, legible y orientado al procesamiento de colecciones en Python.

Comparación de enfoques

En Python, existen distintas formas de aplicar transformaciones a conjuntos de datos. Dos de las más comunes son el uso combinado de lambda + map y las comprensiones de listas. Ambas permiten escribir código más eficiente y expresivo, aunque con enfoques diferentes: mientras lambda + map favorece la programación funcional y el encadenamiento de operaciones, las comprensiones destacan por su claridad y legibilidad en proyectos colaborativos.

- Lambda + map: más alineado con la programación funcional.
- Comprensión de listas: más natural y legible en Python, recomendada para código colaborativo.

Enfoque	Ejemplo de sintaxis	Ventajas ✓	Desventajas !	Cuándo usarlo
lambda + map	<code>list(map(lambda x: "Aprobado" if</code>	Conciso y funcional.-	Menos legible para	Cuando se trabaja en

	<pre>x >= 11 else "Desaprobado", notas))</pre>	Útil cuando se combina con otras funciones de alto nivel (filter, reduce).- Facilita pipelines funcionales.	principiantes.- Difícil de depurar en expresiones largas.- No siempre es el estilo más "Pythonic".	programación funcional o en transformaciones encadenadas.
Comprendión de listas	<pre>[("Aprobado" if n >= 11 else "Desaprobado") for n in notas]</pre>	Sintaxis clara y legible.- Recomendado por la comunidad Python.- Fácil de extender con condiciones adicionales.	Menos "funcional" en sentido estricto.- Puede volverse ilegible si se anidan muchos bucles.	Cuando se prioriza la claridad y legibilidad del código (colaboración en equipos).



- Si estás construyendo pipelines de datos con estilo funcional, lambda + map es muy útil.

- Si tu prioridad es escribir código claro y mantenible, la comprensión de listas es la opción más recomendable en Python.

Ejemplo: pipeline funcional con datos de ventas

Supongamos que tenemos una lista con los ingresos mensuales de una empresa y queremos:

1. Aumentar cada valor en 10% (map).
2. Filtrar solo los ingresos mayores a 5,000 (filter).
3. Calcular el total acumulado (reduce).

Código en Python

```
from functools import reduce

# Datos iniciales: ingresos mensuales en dólares
ingresos = [3200, 4800, 5100, 7000, 3000, 8200, 10000]

# 1. Aumentar 10% con map
ajustados = list(map(lambda x: x * 1.10, ingresos))

# 2. Filtrar solo los > 5000
filtrados = list(filter(lambda x: x > 5000, ajustados))

# 3. Calcular total con reduce
```

```
total = reduce(lambda x, y: x + y, filtrados)

print("Ingresos originales:", ingresos)
print("Ingresos ajustados:", ajustados)
print("Ingresos filtrados (>5000):", filtrados)
print("Total acumulado:", total)
```

Salida esperada

- Ingresos originales: [3200, 4800, 5100, 7000, 3000, 8200, 10000]
- Ingresos ajustados: [3520.0, 5280.0, 5610.0, 7700.0, 3300.0, 9020.0, 11000.0]
- Ingresos filtrados (>5000): [5280.0, 5610.0, 7700.0, 9020.0, 11000.0]
- Total acumulado: 38610.0

Este pipeline muestra cómo la programación funcional:

- Permite procesar datos en pasos claros y encadenados.
- Hace que el código sea más modular y expresivo.
- Resulta especialmente útil en ETL (extracción, transformación y carga) y preprocesamiento de datos en ciencia de datos.

CONTINUAR

Funciones de Alto Nivel

Las funciones de alto nivel (high-order functions) son un pilar de la programación funcional.

Se caracterizan por:

1. **Aceptar funciones como parámetros** → permiten aplicar operaciones genéricas sobre colecciones sin necesidad de escribir bucles explícitos.
2. **Devolver funciones como resultado** → posibilitan la creación de funciones dinámicas (ej. “factories” de funciones).

En Python, estas funciones fomentan un estilo de programación más declarativo, donde se describe qué hacer en lugar de cómo hacerlo paso a paso.

A continuación te mostramos la jerarquía de este tipo de funciones, mostrando cómo pueden recibir otras funciones como parámetros o devolver funciones como resultado, permitiendo así la creación de código más modular, reutilizable y expresivo.



Principales ejemplos en Python

Python incorpora diversas funciones de alto nivel que facilitan el procesamiento de colecciones de datos sin necesidad de escribir bucles explícitos. Estas funciones permiten transformar, filtrar, reducir y ordenar información de manera más clara y eficiente, fomentando un estilo de programación funcional que mejora la legibilidad y la modularidad del código.

Map

(función, iterable)



Transforma cada elemento de una colección aplicando una función. Es ideal para realizar operaciones masivas, como normalizar datos o aplicar cálculos a una lista completa.



Filter

(función, iterable)



Selecciona los elementos que cumplen una condición. Se usa comúnmente para **limpiar datos**, eliminar valores nulos o filtrar registros específicos.



Reduce (función, iterable)



Combina todos los elementos de una colección en un solo valor acumulado, como una suma, un promedio o una concatenación. Es útil en tareas de **agregación de información**.



Sorted (iterable, key = función)



Ordena los elementos de una colección según un criterio definido por una función. Permite organizar datos de forma **personalizada y flexible**, por ejemplo, ordenar listas de objetos o diccionarios por una clave específica.



Ejemplo práctico básico

```
from functools import reduce
```



```
# Calcular la suma de una lista usando reduce

numeros = [1, 2, 3, 4]

suma = reduce(lambda x, y: x + y, numeros)

print(suma) # 10
```

Equivalente a usar `sum(numeros)`, pero muestra el poder de `reduce` en operaciones acumulativas más complejas.

Caso aplicado en ciencia de datos

```
# Lista de empleados
empleados = [
    {"nombre": "Ana", "salario": 5000},
    {"nombre": "Luis", "salario": 3000},
    {"nombre": "Carmen", "salario": 7000}
]

# Filtrar empleados con salario mayor a 4000
altos_salarios = list(filter(lambda e: e["salario"] > 4000, empleados))

# Transformar resultados para extraer solo los nombres
nombres = list(map(lambda e: e["nombre"], altos_salarios))

print("Empleados con alto salario:", nombres)
```

Aplicación real: Seleccionar empleados en recursos humanos para planes de retención.

Ejemplo avanzado: procesamiento de métricas

Supongamos que tenemos los ingresos mensuales de ventas y queremos:

1. Incrementar en 10% (map).
2. Filtrar solo valores superiores a 5000 (filter).
3. Calcular el promedio final (reduce).

```
from functools import reduce

ingresos = [3200, 4800, 5100, 7000, 8200, 10000]

# 1. Ajustar con map
ajustados = list(map(lambda x: x * 1.10, ingresos))

# 2. Filtrar con filter
filtrados = list(filter(lambda x: x > 5000, ajustados))

# 3. Calcular promedio con reduce
promedio = reduce(lambda x, y: x + y, filtrados) / len(filtrados)

print("Ingresos ajustados:", ajustados)
print("Ingresos filtrados:", filtrados)
print("Promedio final:", promedio)
```

Aplicación real: Análisis financiero para determinar si las ventas ajustadas cumplen con un umbral de rentabilidad.

Buenas prácticas

El uso de funciones de alto nivel en Python requiere no solo conocer su sintaxis, sino también aplicarlas de manera adecuada según el contexto. Las siguientes buenas prácticas orientan al estudiante en cómo utilizar map, filter, reduce y lambda de forma eficiente, priorizando la claridad, legibilidad y mantenimiento del código, elementos esenciales en la programación funcional y en la ciencia de datos.



Cuadro comparativo

El siguiente cuadro resume las principales funciones de alto nivel en Python empleadas en la programación funcional. Estas funciones —map, filter, reduce y sorted— permiten realizar transformaciones, filtrados, acumulaciones y ordenamientos sobre colecciones de datos de manera más eficiente. Su aplicación es fundamental en la ciencia de datos, ya que

facilitan el procesamiento, la limpieza y el análisis de grandes volúmenes de información.

Función	Descripción	Ejemplo	Aplicación en ciencia de datos
map	Transforma elementos	map(lambda x: x**2, lista)	Normalizar variables
filter	Filtrar elementos	filter(lambda x: x>0, lista)	Seleccionar datos válidos
reduce	Acumula valores	reduce(lambda x,y:x+y, lista)	Calcular suma/promedio
sorted	Ordena elementos	sorted(lista, key=lambda x:...)	Ranking de ventas, salarios

Ejercicios

EJERCICIO 1

EJERCICIO 2

EJERCICIO 3

Enunciado: Usar map y filter para obtener de una lista de números solo los pares y calcular su cuadrado.

Código en Python:

```
# Lista de números  
numeros = [1, 2, 3, 4, 5, 6]  
  
# Filtrar pares con filter  
pares = list(filter(lambda x: x % 2 == 0, numeros))  
  
# Elevar al cuadrado con map  
cuadrados = list(map(lambda x: x**2, pares))  
print("Números pares:", pares)  
print("Cuadrados de pares:", cuadrados)
```

Salida esperada:

Números pares: [2, 4, 6]
Cuadrados de pares: [4, 16, 36]

Aplicación real: seleccionar registros válidos (condición) y transformarlos.

EJERCICIO 1

EJERCICIO 2

EJERCICIO 3

Enunciado: Dada una lista de nombres, usar map para convertirlos en mayúscula, filter para quedarse solo con los que empiezan con vocal, y reduce para concatenarlos en una sola cadena.

Código en Python:

```
from functools import reduce  
  
# Lista de nombres  
nombres = ["ana", "Luis", "Esteban", "Oscar", "Carmen"]
```

1. Convertir a mayúscula

```
mayus = list(map(lambda x: x.upper(), nombres))
```

2. Filtrar los que empiezan con vocal

```
vocales = list(filter(lambda x: x[0] in "AEIOU", mayus))
```

3. Concatenar con reduce

```
concatenado = reduce(lambda x, y: x + " - " + y, vocales)
```

```
print("Nombres en mayúscula:", mayus)
```

```
print("Nombres que empiezan con vocal:", vocales)
```

```
print("Concatenados:", concatenado)
```

Salida esperada:

Nombres en mayúscula: ['ANA', 'LUIS', 'ESTEBAN', 'OSCAR', 'CARMEN']

Nombres que empiezan con vocal: ['ANA', 'ESTEBAN', 'OSCAR']

Concatenados: ANA - ESTEBAN - OSCAR

Aplicación real: filtrar y transformar nombres de clientes para reportes o dashboards.

EJERCICIO 1

EJERCICIO 2

EJERCICIO 3

Enunciado: Con un dataset de precios de productos, aplicar un 12% de descuento con map, filtrar los precios que quedan por debajo de 50 con filter, y calcular el total de ingresos con reduce.

Código en Python:

```
from functools import reduce
```

```
# Precios originales
precios = [120, 80, 60, 40, 30]

# 1. Aplicar descuento del 12%
descuento = list(map(lambda x: x * 0.88, precios))

# 2. Filtrar precios menores a 50
filtrados = list(filter(lambda x: x < 50, descuento))

# 3. Calcular ingresos totales con reduce
total = reduce(lambda x, y: x + y, descuento)
print("Precios originales:", precios)
print("Precios con descuento:", descuento)
print("Precios filtrados (<50):", filtrados)
print("Total de ingresos:", total)
```

Salida esperada:

```
Precios originales: [120, 80, 60, 40, 30]
Precios con descuento: [105.6, 70.4, 52.8, 35.2, 26.4]
Precios filtrados (<50): [35.2, 26.4]
Total de ingresos: 290.4
```

Aplicación real: cálculo de ingresos después de promociones y análisis de productos con precios bajos.

Las funciones de alto nivel constituyen una herramienta esencial dentro de la programación funcional en Python, ya que permiten trabajar con colecciones de datos de manera más eficiente y expresiva. Su aplicación a

través de map, filter, reduce y sorted fomenta la creación de pipelines de procesamiento claros, reutilizables y escalables, fundamentales en la ciencia de datos. Comprender y aplicar correctamente estas funciones facilita el desarrollo de soluciones más limpias, modulares y alineadas con las mejores prácticas del análisis y transformación de datos.

CONTINUAR

Comprendión de Colecciones

La **comprensión de colecciones** (list, set y dict comprehensions) es una característica de Python que permite construir estructuras en una sola línea de código, combinando:

- **Iteraciones** (recorrido de un iterable).
- **Transformaciones** (aplicación de una expresión).
- **Filtrado opcional** (condiciones lógicas).

Se caracterizan por:



- Sintaxis más **concisa y expresiva** que los bucles tradicionales.
- Más **eficiente en tiempo de ejecución**, ya que se implementan internamente en C.
- Mayor **legibilidad** en operaciones simples.



CONTRAS

- Si se anidan más de 2 niveles, el código pierde claridad.
- Puede generar estructuras muy grandes en memoria si no se controla.

Sintaxis general

[expresión for elemento in iterable if condición]

{expresión for elemento in iterable if condición} # set

{clave: valor for elemento in iterable if condición} # dict

Ejemplo práctico básico

```
# Lista de cuadrados de números pares  
  
cuadrados_pares = [x**2 for x in range(10) if x % 2 == 0]  
  
print(cuadrados_pares)
```

```
# Salida: [0, 4, 16, 36, 64]
```

Caso aplicado en ciencia de datos

Extraer iniciales de nombres

```
nombres = ["Ana Torres", "Luis Gómez", "Carmen Ríos"]  
  
iniciales = [n[0] for n in nombres]  
  
print(iniciales)  
  
# ['A', 'L', 'C']
```

Diccionario con longitudes de palabras

```
palabras = ["Python", "Datos", "Lambda"]  
  
longitudes = {p: len(p) for p in palabras}  
  
print(longitudes)  
  
# {'Python': 6, 'Datos': 5, 'Lambda': 6}
```

Ejercicios

1. Comprensión anidada (listas de listas)

```
# Matriz identidad 3x3  
matriz = [[1 if i == j else 0 for j in range(3)] for i in range(3)]  
print(matriz)  
  
# [[1, 0, 0], [0, 1, 0], [0, 0, 1]]
```

2. Filtrado de datos numéricos

```
import numpy as np  
  
edades = np.random.randint(15, 50, size=10)  
adultos = [e for e in edades if e >= 18]  
print("Edades:", list(edades))  
print("Adultos:", adultos)
```

Aplicación real: Filtrar registros en un dataset de encuestas.

3. Transformación en estructuras de texto

```
# Crear un set con las vocales presentes en una frase  
frase = "Programación en Python"  
vocales = {c.lower() for c in frase if c.lower() in "aeiou"}  
print(vocales)  
  
# {'a', 'o', 'e', 'i'}
```

Aplicación real: Extracción de caracteres únicos para análisis de texto (NLP).

4. Comprensión de diccionarios con condición

```
# Clasificar productos en baratos (<50) y caros (>=50)
precios = {"laptop": 800, "mouse": 25, "teclado": 45, "monitor": 150}
clasificacion = {k: ("Barato" if v < 50 else "Caro") for k, v in precios.items()}
print(clasificacion)

# {'laptop': 'Caro', 'mouse': 'Barato', 'teclado': 'Barato', 'monitor': 'Caro'}
```

Aplicación real: Segmentación de productos según precios en comercio electrónico.

5. Diccionario con los números del 1 al 10 como claves y sus cubos como valores.

Código en Python:

```
cubos = {x: x**3 for x in range(1, 11)}
print(cubos)
```

Salida esperada:

```
{1: 1, 2: 8, 3: 27, 4: 64, 5: 125, 6: 216, 7: 343, 8: 512, 9: 729, 10: 1000}
```

Aplicación real: Generar tablas matemáticas o valores de referencia para análisis.



La comprensión de colecciones es una de las herramientas más potentes y elegantes de Python, ya que permite crear estructuras de datos complejas en una sola línea de código, combinando iteración, filtrado y transformación. Su uso adecuado mejora la legibilidad y eficiencia del código, lo que resulta especialmente útil en la limpieza y preparación de datos dentro de la ciencia de datos. Dominar este recurso ayuda a desarrollar soluciones más claras, expresivas y acordes con el estilo Pythonic.

CONTINUAR

Iteradores y Generadores

En Python, los **iteradores** y generadores son mecanismos que permiten recorrer datos de manera eficiente, evitando cargar todo en memoria.

2

Al llamar `next()`, devuelve el siguiente elemento; cuando no quedan más, lanza `StopIteration`.

Ejemplo: los objetos `list`, `dict` o `tuple` se pueden recorrer mediante un iterador.

Al contrario de una función normal, no pierde su estado entre llamadas, lo que permite producir secuencias grandes de manera eficiente.

Muy útil en Big Data, donde procesar millones de registros a la vez es inviable en memoria.

Ventajas de los generadores

Los **generadores** son una característica avanzada de Python que permite manejar grandes volúmenes de datos de forma eficiente. A diferencia de las listas tradicionales, no almacenan todos los elementos en memoria, sino que los **producen uno a uno bajo demanda**. Esta capacidad los convierte en una herramienta esencial para optimizar el rendimiento en **procesos de análisis**,

transformación y carga de datos dentro de la ciencia de datos y la programación funcional.

- Ocupan poca memoria (producen un elemento a la vez).
- Mejoran la eficiencia en pipelines de procesamiento.
- Se integran naturalmente con `for` y expresiones de comprensión (`(x for x in iterable)`).

Generadores en Ciencia de Datos

01

Carga de Datos



Carga datos de forma rápida y sin agotar la memoria.

02

Análisis de Datos



Permiten análisis detallados sin sobrecargar la memoria.

03

Uso Eficiente de Memoria



Generadores minimizan el uso de memoria al generar datos bajo demanda.

04

Transformación de Datos



Transforman datos eficientemente y sin consumir recursos excesivos.

05

Procesamiento de Datos



Facilitan el manejo eficiente de grandes conjuntos de datos.

Ejemplos

EJEMPLO BÁSICO CON ITERADORES

EJEMPLO APLICADO EN CIENCIA DE DATOS:
GENERADOR DE LOTES (BATCH PROCESSING)

EJEMPLO AVANZADO:
LECTURA DE ARCHIVOS GRANDES

EJEMPLO AVANZADO:
PIPELINE DE ETL CON GENERADORES

Crear un iterador manualmente

```
numeros = [1, 2, 3]
```

```
it = iter(numeros)
```

```
print(next(it)) # 1  
print(next(it)) # 2  
print(next(it)) # 3  
  
# Si se ejecuta otra vez: StopIteration
```

EJEMPLO BÁSICO CON ITERADORES

EJEMPLO APLICADO EN CIENCIA DE DATOS: GENERADOR DE LOTES (BATCH PROCESSING)

EJEMPLO AVANZADO: LECTURA DE ARCHIVOS GRANDES

EJEMPLO AVANZADO: PIPELINE DE ETL CON GENERADORES

```
# Generador para dividir datos en lotes  
def lotes(data, n):  
    for i in range(0, len(data), n):  
        yield data[i:i+n]  
  
# Dataset simulado  
dataset = list(range(1, 21))  
for lote in lotes(dataset, 5):  
    print(lote)
```

Salida esperada:

```
[1, 2, 3, 4, 5]  
[6, 7, 8, 9, 10]  
[11, 12, 13, 14, 15]  
[16, 17, 18, 19, 20]
```

Aplicación real: Procesamiento por mini-batches en machine learning, donde los modelos se entrena en fragmentos de datos para mejorar eficiencia y convergencia.

EJEMPLO BÁSICO CON ITERADORES	EJEMPLO APLICADO EN CIENCIA DE DATOS: GENERADOR DE LOTES (BATCH PROCESSING)	EJEMPLO AVANZADO: LECTURA DE ARCHIVOS GRANDES	EJEMPLO AVANZADO: PIPELINE DE ETL CON GENERADORES
-------------------------------	---	---	---

```
# Generador para leer un archivo grande línea a línea
def leer_lineas(ruta):
    with open(ruta, "r") as f:
        for linea in f:
            yield linea.strip()

# Uso (simulado)
for i, linea in enumerate(leer_lineas("dataset.csv")):
    if i < 5: # mostrar solo primeras 5
        print(linea)
```

Aplicación real: Procesar archivos CSV de millones de registros sin cargarlos en memoria.

EJEMPLO BÁSICO CON ITERADORES	EJEMPLO APLICADO EN CIENCIA DE DATOS: GENERADOR DE LOTES (BATCH PROCESSING)	EJEMPLO AVANZADO: LECTURA DE ARCHIVOS GRANDES	EJEMPLO AVANZADO: PIPELINE DE ETL CON GENERADORES
-------------------------------	---	---	---

```
# Pipeline ETL simple con generadores
def extraer(datos):
    for d in datos:
        yield d

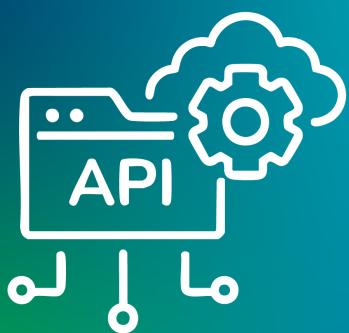
def transformar(datos):
```

```
for d in datos:  
    yield d * 2  
  
def cargar(datos):  
    for d in datos:  
        print(f"Cargando dato: {d}")  
  
# Simular pipeline  
datos = [1, 2, 3, 4, 5]  
pipeline = cargar(transformar(extraer(datos)))
```

Aplicación real: Pipelines de ETL (extracción-transformación-carga) en ingeniería de datos.

Buenas prácticas

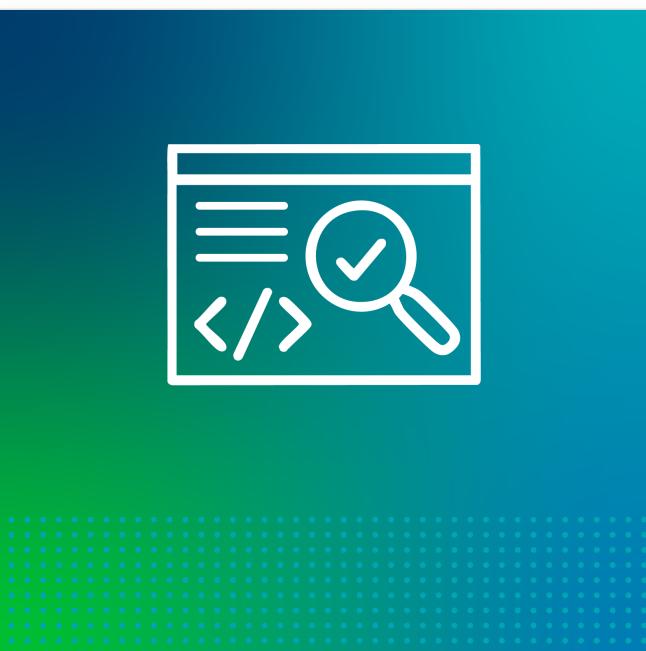
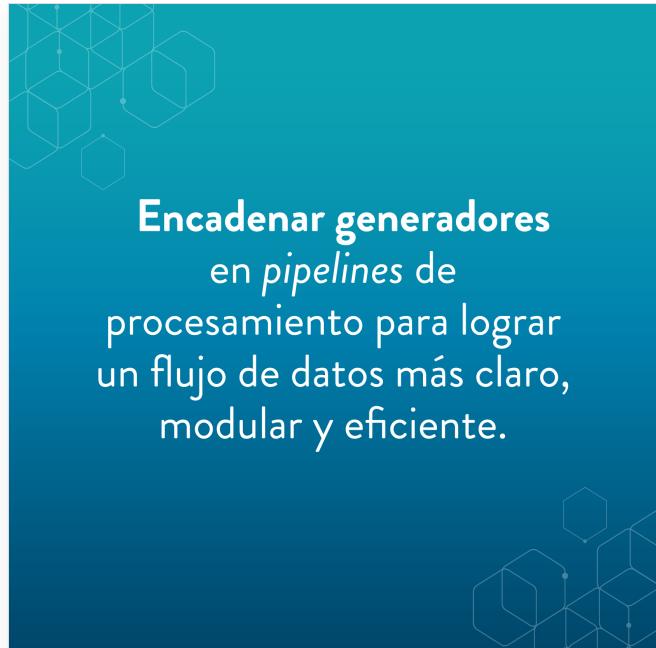
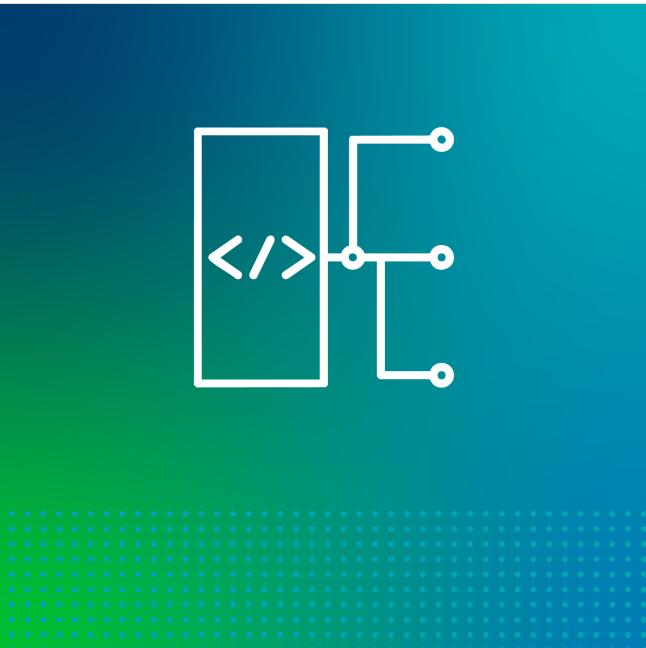
El uso de generadores en Python es una estrategia fundamental para optimizar el rendimiento y la eficiencia del código, especialmente cuando se trabaja con grandes volúmenes de datos. Su aplicación correcta permite aprovechar la evaluación perezosa (lazy evaluation), reduciendo el consumo de memoria y mejorando la velocidad de ejecución.



Usar **generadores** para manejar flujos de datos extensos, como en procesos de streaming, lectura de archivos masivos o conexión con **APIs en tiempo real**.



En contextos de **procesamiento científico o análisis de datos**, preferir la evaluación perezosa frente a la creación de listas completas, ya que evita la sobrecarga de memoria.



En resumen: aplicar buenas prácticas en el uso de generadores permite escribir código más eficiente, escalable y sostenible, cualidades esenciales en

proyectos de análisis de datos y programación avanzada en Python.

Ejercicio adicional

Implementar un generador que devuelva infinitamente números primos.

```
def generador_primos():

    n = 2

    while True:

        if all(n % i != 0 for i in range(2, int(n**0.5)+1)):

            yield n

        n += 1

    # Uso: mostrar los primeros 10 primos

    primos = generador_primos()

    for _ in range(10):

        print(next(primos))
```

Salida esperada: 2, 3, 5, 7, 11, 13, 17, 19, 23, 29

Aplicación real: Generadores infinitos se utilizan en simulaciones estadísticas, generación de secuencias pseudoaleatorias o cálculos matemáticos en investigación.

Cuadro comparativo

El siguiente cuadro presenta una **síntesis de los principales conceptos** abordados en la programación funcional avanzada en Python. Cada elemento —*Lambda*, *Funciones de alto nivel*, *Comprensión de colecciones* e *Iteradores y generadores*— se analiza en función de sus **características**, **aplicaciones en ciencia de datos y buenas prácticas**. Esta comparación permite visualizar las fortalezas de cada recurso y orientar su uso adecuado en el desarrollo de código eficiente, modular y legible.

Concepto	Características	Caso en ciencia de datos	Buenas prácticas
Lambda	Funciones anónimas rápidas	Normalización de datos	Usarlas solo para lógica simple

Funciones de alto nivel	Reciben/devuelven funciones	Filtrar ventas, acumular valores	Combinarlas con lambdas cortas
Comprensión de colecciones	Crear listas/sets/dicts en una línea	Clasificar edades, frecuencias	Evitar anidar más de 2 niveles
Iteradores y generadores	Evaluación perezosa, memoria eficiente	Procesar lotes de datos	Útiles en ETL y Big Data

Los iteradores y generadores representan una de las herramientas más eficientes de Python para trabajar con grandes volúmenes de datos. Su uso permite recorrer y procesar información sin sobrecargar la memoria, gracias a la evaluación perezosa (lazy evaluation). Además, los generadores facilitan la creación de pipelines de datos claros y escalables, ideales para proyectos de Big Data y machine learning.

CONTINUAR

La programación funcional avanzada en Python ofrece un enfoque poderoso para desarrollar código más claro, modular y eficiente. A través del uso de **funciones lambda**, **funciones de alto nivel**, **comprensiones de colecciones e iteradores y generadores**, los estudiantes pueden procesar y transformar grandes volúmenes de datos de forma óptima. Este paradigma fomenta la escritura de programas expresivos y sostenibles, esenciales en la **ciencia de datos y la ingeniería de software moderna**, donde la eficiencia, la escalabilidad y la claridad son fundamentales para la resolución de problemas complejos.



CONTINUAR

Para finalizar el tema, recordemos algunas ideas claves.

1

Funciones Lambda: permiten definir operaciones rápidas y locales sin necesidad de crear funciones completas, optimizando la escritura de transformaciones simples.

2

Funciones de Alto Nivel: como map, filter y reduce, que facilitan la manipulación de colecciones y el procesamiento en pipelines funcionales.

3

Comprendión de Colecciones: sintaxis compacta y eficiente para construir listas, sets o diccionarios, útiles en transformaciones y filtrados de datos.

4

Iteradores y Generadores: permiten recorrer datos y producir secuencias infinitas o masivas con **evaluación perezosa**, siendo esenciales en **Big Data y machine learning**

CONTINUAR

CIERRE



¡FELICITACIONES!

Hemos llegado al final del tema.

Continúa tu aprendizaje en el siguiente módulo.