Módulo 1 (Final): Broadcasting y Arrays Estructurados

Filosofía: Este es el último módulo de NumPy. Hemos visto la fuerza bruta (vectorización) y la limpieza (masking, nan-safe). Ahora veremos la *elegancia* y la *flexibilidad*.
Estos son los conceptos que separan a un "usuario" de NumPy de un "arquitecto" de datos.

1. Desafío: El Poder del Broadcasting (Calibración)

Escenario: Tienes un array de (N, 4) con datos de sensores: (Temp, Hum, Pres, Viento). Cada sensor tiene su propia escala y offset de calibración.
El Reto: Aplicar la calibración valor_calibrado = (valor_crudo * escala) + offset a *todo* el dataset de 1000x4... **sin usar un solo bucle.**

```
import numpy as np

print("--- 1. Desafío: Broadcasting para Calibración ---")
np.random.seed(42)

# 1. Generar 1000 lecturas de 4 sensores
n_lecturas = 1000
n_sensores = 4
datos_crudos = np.random.rand(n_lecturas, n_sensores) * 100
print(f"Shape de datos crudos: {datos_crudos.shape}")

# 2. Definir los parámetros de calibración (¡diferentes shapes!)
#    Queremos aplicar un array (1, 4) a un array (1000, 4)
escalas = np.array([0.5, 1.1, 1.0, 2.5]) # Shape: (4,)
offsets = np.array([-10, +5, 0, -50])    # Shape: (4,)

# 3. La "Magia": Broadcasting
#    NumPy "estira" (broadcast) los arrays (4,) para que encajen
#    con la operación en el array (1000, 4).
#    Lo hace de forma virtual, sin consumir memoria extra.
datos_calibrados = (datos_crudos * escalas) + offsets

print(f"Shape de datos calibrados: {datos_calibrados.shape}")

# 4. Verificación
print("\n--- Verificación ---")
print("Datos Crudos (fila 0):")
print(datos_crudos[0, :])
```

```
print("\nEscalas:", escalas)
print("Offsets:", offsets)
print("\nDatos Calibrados (fila 0):")
print(datos_calibrados[0, :])

# Verifiquemos el sensor 0 (Temp): (37.45 * 0.5) - 10 = 8.72
# Verifiquemos el sensor 3 (Viento): (95.07 * 2.5) - 50 = 187.68
print(f"\nVerificación Temp (Sensor 0): {(datos_crudos[0, 0] * escalas[0]) + offsets[0]:.2f}")
print(f"Verificación Viento (Sensor 3): {(datos_crudos[0, 3] * escalas[3]) + offsets[0]:.2f}")
```

2. Desafío: Fancy Indexing y np.argsort (Ranking de Anomalías)

Escenario: Tienes el dataset datos_calibrados del desafío 1. Quieres encontrar las 10 lecturas (filas completas) que tienen la **temperatura (col 0) más alta**.
El Reto: Obtener esas 10 filas *sin* ordenar todo el dataset, usando solo indexación con arrays de enteros.

```
import numpy as np

# (Continuación... asumimos que 'datos_calibrados' existe)
# Por si se ejecuta solo, lo regeneramos:
if 'datos_calibrados' not in locals():
    np.random.seed(42)
    datos_crudos = np.random.rand(1000, 4) * 100
    escalas = np.array([0.5, 1.1, 1.0, 2.5])
    offsets = np.array([-10, +5, 0, -50])
    datos_calibrados = (datos_crudos * escalas) + offsets


print("\n--- 2. Desafío: Fancy Indexing y `np.argsort` ---")

# 1. Extraer la columna de interés
temperaturas = datos_calibrados[:, 0] # Shape: (1000,)

# 2. Obtener los ÍNDICES que ordenarían esta columna
#    `np.argsort` no devuelve valores, devuelve la *posición* original
#    de los elementos si estuvieran ordenados.
indices_ordenados = np.argsort(temperaturas)

print(f"Índice de la temp. más baja: {indices_ordenados[0]}")
print(f"Índice de la temp. más alta: {indices_ordenados[-1]}")
```

```
# 3. Obtener los 10 índices de las temperaturas más altas
indices_top_10 = indices_ordenados[-10:]
print(f"\nÍndices de las 10 temps más altas:\n {indices_top_10}")

# 4. "Fancy Indexing": Usar un array de enteros para indexar otro array
#    Le pedimos a NumPy las filas [915, 249, 137, ...] del dataset
top_10_lecturas_completas = datos_calibrados[indices_top_10]

print("\n--- Top 10 Lecturas (Temp, Hum, Pres, Viento) ---")
print(top_10_lecturas_completas)

# Verificación: Las temperaturas (col 0) deben estar ordenadas de menor a mayor
print(f"\nTemperaturas (Col 0) de las 10 lecturas:\n {top_10_lecturas_completas[:, 0]}")
```

3. Desafío: Structured Arrays (El "Proto-DataFrame")

Escenario: Tus datos de entrada no son homogéneos. Vienen de un log y son una mezcla de fechas, IDs de texto y valores numéricos.
('2025-10-26T22:00:00', 'SENSOR_A', -25.5, 'OK')
El Reto: Almacenar esto en un array de NumPy *nativo*, conservando los tipos de datos de cada columna, y luego filtrar por una columna de texto.

```
import numpy as np

print("\n--- 3. Desafío: Structured Arrays (Proto-DataFrame) ---")

# 1. Definir la "Estructura" (el dtype)
#    Esto es como un CREATE TABLE en SQL
#    'U10' = String Unicode de 10 chars
#    'M8[s]' = Datetime64 con precisión de segundo
#    'f8' = float64
#    '?' = boolean
log_dtype = [
    ('timestamp', 'M8[s]'),
    ('sensor_id', 'U10'),
    ('value', 'f8'),
    ('status_ok', '?')
]

# 2. Datos de entrada (lista de tuplas, Módulo 0)
datos_log = [
    ('2025-01-01T12:00:00', 'TEMP_A', 15.2, True),
    ('2025-01-01T12:00:00', 'HUM_B', 45.1, True),
```

```
    ('2025-01-01T12:01:00', 'TEMP_A', 15.3, True),
    ('2025-01-01T12:01:00', 'HUM_B', 44.9, False), # Falla
    ('2025-01-01T12:02:00', 'TEMP_A', 15.4, True)
]

# 3. Crear el array estructurado
struct_array = np.array(datos_log, dtype=log_dtype)

print("Array Estructurado Completo:\n", struct_array)
print(f"\nShape: {struct_array.shape}") # Shape (5,) -> 5 *registros*

# 4. Magia: Acceder a una "columna" por su nombre
print("\nAcceso a la columna 'value':")
print(struct_array['value'])

# 5. Magia: Filtrar (masking) usando una columna de texto
mask_temp_a = (struct_array['sensor_id'] == 'TEMP_A')
print(f"\nMáscara para 'TEMP_A':\n {mask_temp_a}")

print("\nSolo registros de 'TEMP_A':\n", struct_array[mask_temp_a])

# 6. Calcular la media de 'value' solo para 'TEMP_A' que están 'OK'
mask_final = (struct_array['sensor_id'] == 'TEMP_A') & (struct_array['status_ok'] == True)
mean_temp_a_ok = np.mean(struct_array[mask_final]['value'])

print(f"\nMedia de 'TEMP_A' con estado 'OK': {mean_temp_a_ok:.2f}")

# Conclusión: Esto es increíblemente potente, pero la sintaxis es verbosa.
# ... y si quisiéramos añadir una columna? Es un infierno.
# POR ESTO se inventó Pandas. Pandas es (básicamente) un diccionario
# de estos arrays con un índice común.
```

4. Desafío: Guardado y Carga de Arrays (.npz)

Escenario: Has procesado tus datos. Tienes el array datos_calibrados (float) y un array info_sensores (strings) que describe las columnas. Quieres guardar ambos en un *único archivo comprimido* para pasárselo a otro script.
El Reto: Usar np.savez_compressed para empaquetar múltiples arrays.

```
import numpy as np
import os # Para verificar que el archivo existe

print("\n--- 4. Desafío: Guardado y Carga (.npz) ---")
```

```
# 1. Datos a guardar
# (Asumimos que 'datos_calibrados' existe)
info_sensores = np.array(['Temperatura_C', 'Humedad_Rel', 'Presion_hPa', 'Viento_kmh'])

print(f"Guardando array {datos_calibrados.shape} y array {info_sensores.shape}...")

# 2. Guardar
#    np.savez_compressed crea un .npz
#    Usamos kwargs (key=value) para nombrar los arrays dentro del archivo
np.savez_compressed(
    'sensores_procesados.npz',
    datos=datos_calibrados,
    info=info_sensores
)

print(f"Archivo 'sensores_procesados.npz' creado.")
print(f"Tamaño: {os.path.getsize('sensores_procesados.npz')} bytes")

# --- En otro script (o más tarde) ---

# 3. Cargar el archivo .npz
#    .load() devuelve un objeto "NpzFile" (parecido a un dict)
print("\nCargando datos desde 'sensores_procesados.npz'...")
loaded_data = np.load('sensores_procesados.npz')

# 4. Acceder a los arrays por los nombres que les dimos
loaded_datos = loaded_data['datos']
loaded_info = loaded_data['info']

print(f"Shape de datos cargados: {loaded_datos.shape}")
print(f"Info de sensores cargada: {loaded_info}")

# 5. Limpiar
loaded_data.close() # Buena práctica
os.remove('sensores_procesados.npz')
print("\nArchivo de limpieza eliminado.")
```