Módulo 1: NumPy - El Poder de la Vectorización

Filosofía: Has aprendido a manipular listas de Python. Ahora, olvida todo eso. Las listas son flexibles, pero *terriblemente* lentas. En data science no trabajamos con 100 elementos, trabajamos con 10 millones.
Un bucle for de Python sobre 10 millones de elementos tarda una eternidad. Una operación de NumPy sobre esos mismos 10 millones de elementos es (casi) instantánea.
¿Por qué? **Vectorización**. NumPy no ejecuta bucles en Python; delega las operaciones a código C precompilado y optimizado que opera sobre bloques de memoria contiguos.
El "Ostión" de NumPy: Tu trabajo ya no es *iterar*. Tu trabajo es *definir operaciones sobre ejes (axis)*.

Parte 1: La Explicación RiguroSA

1. El Objeto Clave: ndarray

Lo primero que hay que entender es que un ndarray (N-dimensional array) no es una lista de Python.

- **Lista Python:** Una colección de punteros a objetos en memoria, que pueden ser de cualquier tipo (int, str, dict...). Están dispersos.
- **Array NumPy:** Un bloque de **memoria contigua y homogénea**. Todos los elementos *deben* ser del mismo tipo (p.ej., float64, int32). Esto es lo que permite las operaciones de C.

```
import numpy as np

# Desde una lista de Python
lista_py = [1, 2, 3, 4, 5]
arr_np = np.array(lista_py)

print(f"Lista de Python: {lista_py}")
print(f"Array de NumPy: {arr_np}")
print(f"Tipo de dato (dtype) del array: {arr_np.dtype}") # crucial

# Array de floats
arr_float = np.array([1.0, 2.5, 3.14])
print(f"Array de floats dtype: {arr_float.dtype}")
```

2. El "BOOM": Vectorización vs. Bucles

Aquí está la magia. ¿Quieres multiplicar cada elemento de una lista por 10?

```
# --- El método LENTO (Módulo 0) ---
lista_py = [i for i in range(1000000)] # 1 millón de elementos
# (Iniciamos cronómetro)
resultado_py = []
for x in lista_py:
    resultado_py.append(x * 10)
# (Paramos cronómetro) -> Tarda un tiempo X

# --- El método RÁPIDO (Módulo 1) ---
arr_np = np.arange(1000000) # np.arange es el range() de NumPy
# (Iniciamos cronómetro)
resultado_np = arr_np * 10
# (Paramos cronómetro) -> Tarda un tiempo X/1000 (o menos)

print(f"Forma del array (shape): {resultado_np.shape}")
print(f"Primeros 5 elementos: {resultado_np[:5]}")
print(f"Últimos 5 elementos: {resultado_np[-5:]}")
```

Cualquier operación matemática (+, -, *, /, **, np.log, np.exp, np.sin...) está vectorizada. Se llaman **ufuncs** (Universal Functions).

3. Creación de Arrays (El Taller del Data Scientist)

Rara vez crearás un array a mano. Casi siempre lo *generarás*.

```
# --- Arrays básicos ---
zeros = np.zeros((3, 4)) # Matriz de 3x4 (3 filas, 4 cols) de ceros
print("Zeros:\n", zeros)

ones = np.ones(5) # Vector de 5 unos
print("\nOnes:\n", ones)

# --- Secuencias ---
# Como range(), pero para floats
rango = np.arange(0, 10, 0.5) # De 0 a 10 (excl.), en pasos de 0.5
print("\narange:\n", rango)

# MUY útil: Divide un rango en N puntos exactos
# "Quiero 50 puntos entre 0 y 100"
lineal = np.linspace(0, 100, 50)
print("\nlinspace (50 puntos):\n", lineal)
```

4. Generación de Datos (Tu Petición Clave)

Aquí está el núcleo de tu petición. El módulo np.random es tu fábrica de datos falsos.

```
# Fijamos la "semilla" (seed) para que los resultados sean reproducibles
np.random.seed(42)

# --- Distribución Uniforme (Todos los valores igual de probables) ---
# 100 valores entre 0 (incl.) y 1 (excl.)
uniforme = np.random.rand(100)
# 100 valores entre 10 y 20
uniforme_rango = np.random.uniform(low=10, high=20, size=100)

# --- Distribución Normal (Gaussiana) ---
# La más usada. Simula fenómenos naturales (altura, temperatura, error...)
# 1000 valores con media (loc) -25 y desviación estándar (scale) 5
# (Como tus temperaturas antárticas)
normal = np.random.normal(loc=-25.0, scale=5.0, size=1000)
print(f"\nDist. Normal (Media): {normal.mean():.2f}") # Debería ser cercano a -25
print(f"Dist. Normal (Std): {normal.std():.2f}")   # Debería ser cercano a 5

# --- Datos Categóricos y Discretos ---
# Simula el lanzamiento de un dado (números de 1 a 6) 20 veces
dados = np.random.randint(low=1, high=7, size=20)
print("\nLanzamiento de dados:\n", dados)

# (Como tus 'sensor_ids' o 'ice_condition')
# Elegir 10 veces de una lista, con probabilidades
opciones = ['TEMP_A', 'TEMP_B', 'HUM_A']
probabilidades = [0.6, 0.3, 0.1] # TEMP_A es más probable
elegidos = np.random.choice(opciones, size=10, p=probabilidades)
print("\nSensores elegidos:\n", elegidos)

# (Como tu 'penguin_count')
# Distribución de Poisson (para "conteos" en un intervalo)
# Media de 500 pingüinos
pinguinos = np.random.poisson(lam=500, size=50)
print("\nConteo de pingüinos (Poisson):\n", pinguinos)
```

5. El Infierno: np.nan (Not a Number)

En data science, los datos faltantes son la norma. En Python usamos None. En NumPy, usamos np.nan. Es un float especial.
Propiedad Clave: np.nan NUNCA es igual a nada, ni siquiera a sí mismo.

```
print(f"\n¿np.nan == np.nan?  ->  {np.nan == np.nan}") # ¡Falso!

# Cómo comprobar si hay un NaN:
```

```
x = np.array([1, 2, np.nan, 4])
print(f"Array con NaN: {x}")
print(f"Check con np.isnan(): {np.isnan(x)}")

# Las operaciones matemáticas con NaN se propagan
print(f"Suma con NaN: {x.sum()}") # -> nan

# ¡Solución! Usa las funciones "nan-safe"
print(f"Suma segura (nan-safe): {np.nansum(x)}")
print(f"Media segura (nan-safe): {np.nanmean(x)}")
print(f"Máximo seguro (nan-safe): {np.nanmax(x)}")
```

Parte 2: El Taller - Indexing y Masking

Aquí es donde NumPy barre a las listas.

1. Slicing 2D (Matrices)

Olvida las listas anidadas lista[0][1]. NumPy usa comas: arr[fila, columna].

```
# Matriz de 4x3 con números del 0 al 11
arr2d = np.arange(12).reshape((4, 3))
print("Matriz 2D:\n", arr2d)

# --- Slicing ---
# sintaxis: [filas, columnas]

# Elemento en fila 1, columna 2
print(f"\nElemento [1, 2]: {arr2d[1, 2]}") # -> 5

# TODA la fila 2
print("\nFila 2 [2, :]:\n", arr2d[2, :]) # -> [6, 7, 8]

# TODA la columna 1
print("\nColumna 1 [:, 1]:\n", arr2d[:, 1]) # -> [1 4 7 10]

# Sub-matriz (filas 0 y 1, columnas 1 y 2)
print("\nSub-matriz [0:2, 1:3]:\n", arr2d[0:2, 1:3])
```

2. Boolean Masking (El "Poder" Real)

Esta es la herramienta más importante. Es el reemplazo de if en un bucle. Es la base del filtro en Pandas.

Concepto: Creas un array de True/False (una "máscara") y lo usas para indexar tu array de

```
datos.
data = np.random.normal(0, 10, 10) # 10 valores aleatorios
print("Datos originales:\n", data)

# 1. Crear la máscara (no filtra, solo pregunta)
mascara_positivos = (data > 0)
print("\nMáscara (¿es > 0?):\n", mascara_positivos)

# 2. Aplicar la máscara (filtrar)
print("\nSolo positivos:\n", data[mascara_positivos])

# --- Todo en una línea (lo habitual) ---
print("\nSolo > 5:\n", data[data > 5])

# --- Múltiples condiciones ---
# & (AND), | (OR), ~ (NOT)
# ¡OBLIGATORIO usar paréntesis!
mascara_multiple = (data > -5) & (data < 5)
print("\nEntre -5 y 5:\n", data[mascara_multiple])

# --- Aplicar a matrices (ej: outliers) ---
arr2d = np.array([[10, 20, 999], [40, -999, 50]])
print("\nMatriz con outliers:\n", arr2d)

# Reemplazar valores "malos" (> 100) con 0
arr2d[arr2d > 100] = 0
print("\nMatriz limpia:\n", arr2d)
```

Parte 3: Desafíos (Generación y Limpieza)

Desafío 1: Generación de Dataset (Tu Petición)

Escenario: Generar un dataset de 1000 registros de 3 sensores.

- col_0 (Timestamp): Segundos Unix. Empezando en 1700000000, incrementos de 60 segundos.
- col_1 (Sensor A - Temp): Distribución normal (Media 20, Std 2).
- col_2 (Sensor B - Humedad): Distribución uniforme (Rango 40 a 60).
- Combinarlo todo en una única matriz NumPy de 1000x3.

```
print("\n--- Desafío 1: Generación Dataset ---")
np.random.seed(42)
n_records = 1000
```

```
# Col 0: Timestamps (usando linspace para precisión)
# Queremos 1000 pasos desde T_0 hasta T_0 + (999 * 60 segundos)
ts_start = 1700000000
ts_end = ts_start + (n_records - 1) * 60
col_0_ts = np.linspace(ts_start, ts_end, num=n_records)

# Col 1: Sensor A (Normal)
col_1_temp = np.random.normal(loc=20, scale=2, size=n_records)

# Col 2: Sensor B (Uniforme)
col_2_hum = np.random.uniform(low=40, high=60, size=n_records)

# --- Combinar ---
# Usamos np.column_stack para "apilar" los vectores como columnas
dataset = np.column_stack([col_0_ts, col_1_temp, col_2_hum])

print(f"Shape del dataset final: {dataset.shape}")
print("Head (primeras 5 filas):\n", dataset[:5, :])
```

Desafío 2: Limpieza de Outliers y NaNs (Tu Petición)

Escenario: Usando la columna col_1_temp del desafío anterior:

1. Introducir 5% de NaNs (datos faltantes).
2. Introducir 2% de Outliers (p.ej., valores > 1000).
3. Calcular la media y la desviación estándar *limpias* (ignorando NaNs y outliers).

```
print("\n--- Desafío 2: Limpieza de Outliers y NaNs ---")
np.random.seed(42)
n_records = 1000
data_sucia = col_1_temp.copy() # Copiar para no modificar el original

# 1. Introducir 5% de NaNs
n_nan = int(n_records * 0.05)
idx_nan = np.random.choice(n_records, n_nan, replace=False)
data_sucia[idx_nan] = np.nan
print(f"Total de NaNs introducidos: {np.isnan(data_sucia).sum()}")

# 2. Introducir 2% de Outliers
n_outlier = int(n_records * 0.02)
idx_outlier = np.random.choice(n_records, n_outlier, replace=False)
data_sucia[idx_outlier] = np.random.uniform(1000, 2000, n_outlier)
print(f"Media con datos sucios (sin NaN): {np.nanmean(data_sucia):.2f}") # Media
distorsionada
```

```
# 3. Limpieza y Cálculo
# Definimos "outlier" como cualquier valor > 100 (límite físico)
limite_fisico = 100

# Usamos masking para encontrar outliers y reemplazarlos por NaN
# (sin incluir los que *ya* son NaN)
mask_outlier = (data_sucia > limite_fisico) & (~np.isnan(data_sucia))
data_sucia[mask_outlier] = np.nan

print(f"Total de NaNs + Outliers convertidos: {np.isnan(data_sucia).sum()}")

# 4. Cálculo final (nan-safe)
media_limpia = np.nanmean(data_sucia)
std_limpio = np.nanstd(data_sucia)

print(f"Media Limpia (Original era ~20): {media_limpia:.2f}")
print(f"Std Limpio (Original era ~2): {std_limpio:.2f}")
```

Desafío 3: Simulación de TimeSeries (Paseo Aleatorio)

Escenario: Simular el precio de una acción (o una temperatura con "memoria") durante 500 días. El precio de hoy es el precio de ayer + un cambio aleatorio (normal, media 0).

```
print("\n--- Desafío 3: Paseo Aleatorio (TimeSeries) ---")
np.random.seed(42)
precio_inicial = 100.0
n_dias = 500

# 1. Generar los "cambios" diarios
# (Media 0, Std 1.5)
cambios = np.random.normal(loc=0, scale=1.5, size=n_dias)

# 2. Calcular la serie temporal usando .cumsum() (Suma Acumulada)
# Esta es una ufunc potentísima
paseo = cambios.cumsum()

# 3. Sumar el precio inicial
serie_temporal = precio_inicial + paseo

print(f"Forma de la serie: {serie_temporal.shape}")
print("Primeros 10 'días':\n", serie_temporal[:10])
print("Últimos 10 'días':\n", serie_temporal[-10:])
```

Conclusión: El Puente a Pandas

Acabamos de ver NumPy. **Un DataFrame de Pandas no es más que un diccionario de Arrays de NumPy**. La columna temperature_c de tu función generate_antarctic_data es un ndarray de NumPy.
Cuando haces df['temperature_c'] > 0, Pandas está usando *Boolean Masking* de NumPy por debajo. Cuando haces df['temperature_c'].mean(), Pandas llama a np.nanmean().
Has aprendido el "motor". En el Módulo 2, aprenderemos a manejar el "chasis" (Pandas), que nos da etiquetas (índices y nombres de columna) para manejar estos arrays.