# Python y Computación Científica para el Agro: De Datos Crudos a Decisiones Inteligentes

Desarrollo + ejemplos comentados + ejercicios

Curso de IA Aplicada al Agro

Enero 2026

## Índice general

# 1 De Python básico a NumPy (el puente conceptual)

Este texto es una guía corta para que cualquier estudiante, incluso si apenas está empezando, entienda el **por qué** y el **para qué** de lo que vamos a construir en esta semana: validación de datos, limpieza de CSV, medición de rendimiento y procesamiento matricial con NumPy.

**Metas**

Al finalizar deberías poder leer el resto del manual sin sentir que "aparecen cosas mágicas". La idea es entender:

- Qué es un programa en Python y cómo se organiza.
- Cómo se toman decisiones (`if/elif/else`) y se repiten tareas (`for`).
- Por qué usamos funciones para empaquetar lógica.
- Cómo funcionan los errores y por qué los usamos (fail fast).
- Cómo leer archivos de texto/CSV de manera simple.
- Por qué NumPy cambia el juego en rendimiento (vectorización).
- Qué son máscaras booleanas y cómo `np.where` decide por celda.

## 1.1 Qué es un programa en Python (y qué significa "ejecutar")

Un programa en Python es un archivo `.py` con instrucciones que la máquina ejecuta en orden. En este curso, casi todos los scripts siguen esta estructura:

Listing 1: Estructura típica de un script

```python
def main():
    # aquí va la lógica principal
    print("Hola")

if __name__ == "__main__":
    main()
```

**Idea clave:** `main()` concentra el flujo del programa y el bloque `if __name__ == "__main__":` hace que el script se ejecute solo cuando lo corres directamente.

## 1.2 Tipos de datos: números, texto y booleanos

Para trabajar con sensores y mediciones necesitamos entender tres tipos básicos:

- **Números**: `int` y `float` (ej. temperatura, humedad).
- **Texto**: `str` (ej. id del sensor, fecha).
- **Booleanos**: `True/False` (ej. "está en sequía?" sí/no).

Listing 2: Ejemplo mínimo de tipos

```python
temp = 24.2          # float
hum = 55             # int
sensor_id = "SENSOR_01"  # str
alarma = hum < 40     # bool
```

# 2 Verificar y convertir tipos de datos en Python

En programación científica es fundamental asegurarse de que cada variable tenga el **tipo de dato** correcto (entero, número real, texto, etc.), especialmente antes de hacer cálculos o comparaciones.

## 2.1 Tipos básicos: `int`, `float`, `str`

En Python, los tipos más usados en este curso son:

- `int`: números enteros, por ejemplo `3`, `-10`, `42`.
- `float`: números reales (con decimales), por ejemplo `3.14`, `-0.5`, `25.0`.
- `str`: cadenas de texto, por ejemplo `"Café"`, `"Zona Norte"`, `"25.0"`.

## 2.2 Cómo saber de qué tipo es una variable

Python permite inspeccionar el tipo de una variable con la función `type()`:

Listing 3: Inspección de tipos de variables

```
temperatura = 25.0
parcela_id = "P001"
cantidad_sacos = 30

print(type(temperatura))   # <class 'float'>
print(type(parcela_id))    # <class 'str'>
print(type(cantidad_sacos))# <class 'int'>
```

Esto es útil para depurar errores cuando una operación no funciona como se esperaba.

## 2.3 Convertir entre tipos (casting)

Muchas veces los datos llegan como texto (`str`) desde un archivo CSV o desde la entrada del usuario, y es necesario convertirlos a número antes de operar.

Listing 4: Conversión de tipos

```
valor_bruto = "28.5"   # Esto es texto, no un número

temperatura = float(valor_bruto)  # str -> float
sacos_str = "30"
sacos = int(sacos_str)            # str -> int

print(temperatura + 1.5)          # 30.0
print(sacos + 5)                  # 35
```

Conversión típica en este curso:

- `str` a `int`: `int("42")`.
- `str` a `float`: `float("3.14")`.
- Cualquier tipo a `str`: `str(25.0)`.

## 2.4 Ejemplo aplicado al agro

Al leer un CSV con mediciones de parcelas, todas las columnas llegan como texto. Antes de tomar decisiones agronómicas, se convierten al tipo correcto:

Listing 5: Asegurar tipos correctos desde CSV

```
fila = {
    "parcela_id": "P010",
    "temperatura_c": "29.7",
    "humedad_suelo_pct": "38",
    "ph": "5.4"
}

parcela_id = fila["parcela_id"]                # str
temperatura = float(fila["temperatura_c"])     # str -> float
humedad = int(fila["humedad_suelo_pct"])       # str -> int
ph = float(fila["ph"])                         # str -> float

print(type(parcela_id))  # <class 'str'>
print(type(temperatura)) # <class 'float'>
print(type(humedad))     # <class 'int'>
print(type(ph))          # <class 'float'>
```

Si se omite esta conversión, las comparaciones como `temperatura > 30` o `humedad < 40` pueden fallar o producir resultados incorrectos.

# 3 Control de flujo: decisiones y reglas

Las reglas agronómicas se expresan como condiciones:

Listing 6: Decisión simple con if/elif/else

```
if hum < 40:
    print("Riego")
elif hum > 80:
    print("Drenaje")
else:
    print("Normal")
```

**Puente al manual:** más adelante no aplicaremos estas reglas a un solo valor, sino a una matriz completa con NumPy.

## 3.1 Repetición: recorrer muchos datos

Cuando hay múltiples sensores o múltiples filas en un archivo, repetimos acciones:

Listing 7: Recorrer una lista con for

```
humedades = [55, 60, 38, 92]
for h in humedades:
    if h < 40:
        print("Riego")
```

**Puente al manual:** recorrer 1,000,000 valores con un `for` es posible, pero lento; por eso aparece NumPy.

## 3.2 Estructuras clave: listas y diccionarios

En IA aplicada al agro se usan mucho:

- **Listas**: colecciones ordenadas (series de mediciones).
- **Diccionarios**: registros con campos (una fila de sensor con columnas).

Listing 8: Registro tipo diccionario (una fila de sensor)

```
row = {"id": "SENSOR_01", "temp": 24.2, "hum": 55}
```

**Puente al manual:** el validador fail-fast trabaja sobre diccionarios así.

# 4 Funciones: empaquetar lógica reutilizable

Cuando una regla se repite, se convierte en función:

Listing 9: Función simple reutilizable

```python
def necesita_riego(hum):
    return hum < 40

print(necesita_riego(35))  # True
```

**Puente al manual:** `validar_sensor_row(row)` es una función que formaliza una política de calidad de datos.

## 4.1 Errores y excepciones: fail fast en la práctica

En ingeniería, a veces la respuesta correcta es **parar**.

Listing 10: Excepciones con try/except

```python
def dividir(a, b):
    if b == 0:
        raise ValueError("No se puede dividir por cero")
    return a / b

try:
    print(dividir(10, 0))
except ValueError as e:
    print("Error:", e)
```

**Puente al manual:** cuando una fila del CSV está corrupta o fuera de rango, la descartamos y registramos el motivo; eso evita contaminar resultados.

## 4.2 Archivos: leer texto y CSV sin librerías

Un CSV es texto con comas. Podemos procesarlo línea por línea:

Listing 11: Lectura mínima de un CSV (sin pandas)

```python
with open("sensores.csv", "r", encoding="utf-8") as f:
    header = f.readline()  # primera línea: nombres de columnas
    for linea in f:
        partes = linea.strip().split(",")
        # aquí validas y conviertes tipos
```

**Puente al manual:** ese patrón es la base de la limpieza del CSV sucio y la escritura del CSV limpio.

# 5 Por qué NumPy: cuando el tamaño importa

Python es excelente para lógica, pero un `for` ejecuta iteración por iteración en el intérprete. NumPy permite operar con arreglos completos de números de forma eficiente (operaciones implementadas en bajo nivel).

**Idea**

En el manual medimos esto con un benchmark:

- versión lenta: bucle `for`
- versión rápida: `np.sin(x) + np.cos(x)`

## 5.1 Máscaras booleanas: decisiones sobre mapas

Una máscara booleana es una matriz de `True/False` que marca posiciones de interés:

Listing 12: Máscara booleana en matriz

```
import numpy as np
humedad = np.random.rand(5, 5)
sequia = humedad < 0.2  # matriz booleana del mismo tamaño
```

Esto permite seleccionar/contar/actuar sobre celdas sin iterar manualmente.

## 5.2 `np.where`: decidir por celda sin if por celda

`np.where(condicion, valor_si_true, valor_si_false)` construye una salida celda-a-celda.

Listing 13: np.where como motor de decisiones

```
acciones = np.where(humedad < 0.2, "R", ".")  # "R" regar, "." normal
```

**Puente al manual:** esto es el núcleo del “motor de decisión hídrica” sobre mapas.

# 6 Profundización: NumPy y su potencial

Esta sección amplía NumPy más allá de "hacer matrices": el objetivo es dominar las ideas que hacen a NumPy la base de casi todo en ciencia de datos y ML: **arrays**, **formas (shape)**, **tipos (dtype)**, **operaciones vectorizadas (ufuncs)**, **broadcasting** y **máscaras**.

## 6.1 El objeto central: `ndarray` (shape y dtype)

Un `ndarray` es un arreglo multidimensional homogéneo: todos los elementos tienen el mismo tipo numérico (`dtype`). Dos propiedades mandan:

- **shape**: cuántas filas/columnas (o dimensiones) tiene.
- **dtype**: tipo de dato (float64, int32, etc.).

Listing 14: Crear arrays y observar shape/dtype

```
import numpy as np

a = np.array([1, 2, 3])          # 1D
b = np.array([[1.0, 2.0], [3.0, 4.0]])  # 2D

print(a.shape, a.dtype)
print(b.shape, b.dtype)
```

**Idea práctica**

`shape` te dice el "tamaño geométrico" de tus datos; `dtype` te dice el "costo" en memoria y qué operaciones son posibles.

## 6.2 Creación eficiente de arrays

Algunas formas típicas de crear datos:

- `np.zeros`, `np.ones`: inicialización rápida.
- `np.arange`, `np.linspace`: secuencias numéricas.
- `np.random.rand`: simulación/ruido (para pruebas).

Listing 15: Creación rápida de matrices

```
import numpy as np

Z = np.zeros((3, 4))       # matriz 3x4 llena de 0
O = np.ones((2, 2))        # matriz 2x2 llena de 1
x = np.arange(0, 10)       # [0..9]
t = np.linspace(0, 1, 5)   # 5 puntos entre 0 y 1
```

## 6.3 Indexing y slicing (leer sub-zonas del lote)

Piensa una matriz como un mapa del lote: puedes seleccionar sub-regiones con slicing.

Listing 16: Slicing en 2D: sub-zona del mapa

```python
import numpy as np

humedad = np.random.rand(100, 100)

zona_centro = humedad[40:60, 40:60]   # 20x20
primera_fila = humedad[0, :]          # 100 valores
primera_col = humedad[:, 0]           # 100 valores
```

**Error común**

En NumPy, el orden es `[filas, columnas]`.

# 7 Vectorización con ufuncs: “sin for”

Las operaciones element-wise (por elemento) se aplican a todo el array:

Listing 17: Ufuncs: operaciones vectorizadas

```python
import numpy as np

x = np.random.rand(5)
y = np.sin(x) + np.cos(x)  # aplica a todo el vector
```

**Conexión con HPC**

Esto es la base del benchmark del manual: mover el trabajo del intérprete (loop Python) a operaciones vectorizadas.

## 7.1 Broadcasting: el superpoder

Broadcasting describe cómo NumPy trata arreglos de diferentes shapes en operaciones aritméticas; el arreglo pequeño se “expande” lógicamente para ser compatible, evitando bucles explícitos y usualmente sin copias innecesarias.

Listing 18: Broadcasting: sumar vector a cada fila

```python
import numpy as np

A = np.random.rand(3, 4)  # 3x4
b = np.array([10, 20, 30, 40])  # (4,)

# b se aplica a cada fila de A por broadcasting
C = A + b
```

**Ejercicio (medio): fertilización por franjas**

Simula un lote `A` de 100x100 y un vector `b` de 100 valores (fertilización por columna). Usa broadcasting para obtener el lote ajustado `A+b`.

## 7.2 Reducciones y el parámetro `axis`

Reducciones como `sum`, `mean`, `max` colapsan dimensiones. El argumento `axis` indica sobre qué eje reduces.

Listing 19: Reducciones por eje

```python
import numpy as np

M = np.random.rand(3, 4)

total = M.sum()            # escalar: suma todo
por_fila = M.mean(axis=1)  # (3,) promedio por fila
por_col  = M.mean(axis=0)  # (4,) promedio por columna
```

**Lectura rápida**

`axis=0` suele significar “colapsar filas” (resultado por columna). `axis=1` suele significar “colapsar columnas” (resultado por fila).

## 7.3 Máscaras booleanas (boolean indexing)

Una condición sobre un array produce un array booleano (True/False) del mismo shape, que puede usarse para seleccionar elementos.

Listing 20: Máscara + conteo de celdas en sequía

```python
import numpy as np

humedad = np.random.rand(100, 100)
sequia = humedad < 0.2

area_sequia = np.sum(sequia)  # cuenta True
```

## 7.4 `np.where`: decisiones vectorizadas

`np.where` permite asignar valores en función de una condición, por celda, sin iterar manualmente.

Listing 21: Mapa de acciones con np.where

```python
import numpy as np

humedad = np.random.rand(10, 10)
acciones = np.where(humedad < 0.2, "R", ".")  # R=regar, .=normal
```

**Ejercicio (difícil): política de riego y drenaje**

Crea `acciones` con 3 clases:

- `"R"` si `humedad < 0.2`
- `"D"` si `humedad > 0.85`
- `"."` en otro caso

y calcula los porcentajes de cada clase usando máscaras.

# 8 Visualización esencial con Matplotlib (desde NumPy)

La visualización es parte del flujo científico: permite validar datos, detectar outliers y comunicar resultados. Como ya estamos usando NumPy (`np`), el siguiente paso natural es graficar con Matplotlib a través de `matplotlib.pyplot`.

**Objetivo**

Aprender lo mínimo indispensable para:

- Graficar series 1D (líneas).
- Comparar mediciones (scatter/barras).
- Visualizar una matriz 2D (heatmap con `imshow`).
- Guardar gráficos en `reports/` con `savefig`.

## 8.1 `pyplot`: la interfaz rápida

El flujo típico con `pyplot` es: crear gráfico, etiquetar, y guardar/mostrar.

## 8.2 Gráfico de línea (1D)

Listing 22: Línea: serie 1D con etiquetas y leyenda

```
import numpy as np # importa la libreria numpy, con alias np
import matplotlib.pyplot as plt # importa la libreria matplotlib con su sublibrería pyplot y
    alias plt

# Simulación: 24 mediciones de humedad (una por hora)
# Evalúa la función en una línea que está en el intervalo [0,2pi)
# Con 24 puntos, no se incluye el extremo derecho

hum = 40 + 10*np.sin(np.linspace(0, 2*np.pi, 24))


plt.plot(hum, label="Humedad (%)")  # Grafica la curva
plt.title("Humedad por hora")       # Título
plt.xlabel("Hora")                  # Nombre del eje x
plt.ylabel("Humedad (%)")           # Nombre del eje y
plt.legend("Np") # leyenda
plt.show()                          # Muestra la gráfica
```

El gráfico se puede apreciar en la figura 8.2

## 8.3 Scatter (dispersión): detectar relación y outliers

Diagrama de dispersión generado por Scatter de python, ver figura 2

Listing 23: Scatter: temperatura vs humedad

```
import numpy as np
import matplotlib.pyplot as plt

np.random.seed(7) # Utiliza una semilla, para que los datos no cambien
temp = 20 + 10*np.random.rand(100) # Genera una serie aleatoria de temperaturas
hum  = 30 + 40*np.random.rand(100) # Genera una serie aleatoria de humedades

plt.scatter(temp, hum, s=18) # Utiliza el tipo de gráfica scatter
```

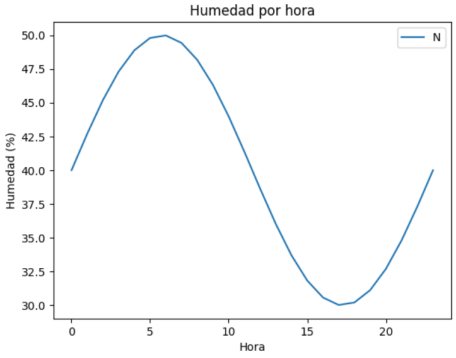


Figura 1: Gráfica generada con los comandos de python

```
9  plt.title("Relación temperatura vs humedad")
10 plt.xlabel("Temperatura (C)")
11 plt.ylabel("Humedad (%)")
12 plt.show()
```

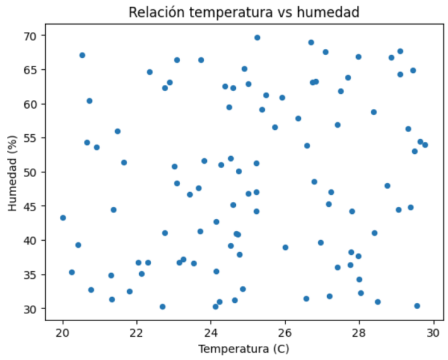


Figura 2: Diagrama de dispersión, generada con numpy

## 8.4 Heatmap (2D) con `imshow` + `colorbar`

Para una matriz (por ejemplo, mapa de humedad), `imshow` la dibuja como imagen y `colorbar` añade la escala de colores.

Listing 24: Heatmap: humedad 2D con colorbar

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 humedad = np.random.rand(50, 50) # Genera una matriz de valores aleatorios
5
6 plt.imshow(humedad, cmap="viridis", interpolation="nearest")  # Muestra el mapa de calor
7 plt.colorbar()                                                # Colorea
8 plt.title("Mapa de humedad (50x50)")
9 plt.show()
```

Se puede apreciar el mapa de calor en la figura 3

## 8.5 Guardar gráficos con `savefig`

Para incluir resultados en el reporte, lo ideal es guardar figuras en archivos (por ejemplo PNG) usando `savefig`.

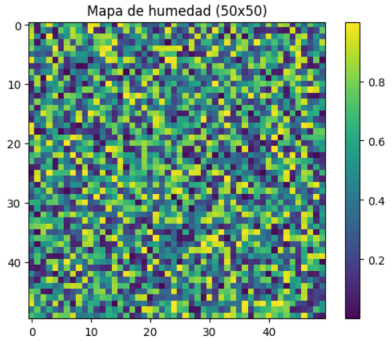


Figura 3: Mapa de calor con numpy

Listing 25: Guardar figura a mapa_humedad.png

```python
import os # Importa comandos del sistema
import numpy as np
import matplotlib.pyplot as plt

os.makedirs("reports", exist_ok=True) # Verifica que exista la carpeta reports

humedad = np.random.rand(50, 50)

plt.imshow(humedad, cmap="viridis", interpolation="nearest")
plt.colorbar()
plt.title("Mapa de humedad (50x50)")

plt.savefig("reports/mapa_humedad.png", dpi=150, bbox_inches="tight")  # Guarda la figura con
    el nombre mapa_humedad.png en la carpeta reports
```

**Tip práctico**

Si vas a usar `plt.show()`, guarda antes con `plt.savefig(...)` para evitar perder la figura en algunos flujos de ejecución.

## 8.6 Ejercicios

**Ejercicio (básico): dos parcelas**

Grafica dos series (dos parcelas) en la misma figura usando `label=` y `plt.legend()`.

**Ejercicio (medio): mapa de sequía**

Dada una matriz `humedad`, crea `sequia = humedad < 0.2` y grafica esa máscara con `imshow`.

**Ejercicio (difícil): diagnóstico 2x2**

Construye una figura con 4 subgráficos:

- mapa de humedad,
- máscara de sequía,
- máscara de inundación,
- mapa de acciones (convertido a números).

Guárdala como `reports/diagnostico.png`.

## Prefacio: de scripts a sistemas que escalan

Vamos a profesionalizar el código: validar datos, medir rendimiento y usar NumPy para operar por bloques.

**Definition of Done (DoD)**

Al terminar, debes tener:

- Un validador *fail fast* para registros de sensores.
- Un pipeline CSV sucio $\rightarrow$ CSV limpio (sin pandas).
- Un benchmark reproducible con `time.perf_counter()` (loop vs vectorizado).
- Un mapa de lote (matriz) con máscaras booleanas y `np.where`.
- Un reporte en LaTeX (manual en Semana 2).

# 9 Programación defensiva (Fail Fast)

## 9.1 Por qué esto es ingeniería

Un sensor puede fallar. Un archivo puede venir mal. Un dato puede ser imposible. Si el sistema no valida, el error se propaga.

**Regla**

Valida temprano, falla claro y registra qué descartaste.

## 9.2 Programa 1: Validador con guard clauses (comentado)

Listing 26: Programa 1: Validación fail fast con guard clauses

```python
# Programa 1:
# Este script enseña a validar datos antes de procesarlos.
# La idea es que el sistema "falle rápido" (fail fast) con mensajes claros,
# en lugar de continuar con datos inválidos y generar resultados incorrectos.

from __future__ import annotations


def validar_sensor_row(row: dict) -> dict:
    """
    Recibe un registro tipo dict y lo valida.
    Si algo está mal: lanza ValueError (falla rápido).
    Si todo está bien: devuelve un dict limpio/normalizado.

    Formato esperado:
      {"id": "SENSOR_01", "temp": 24.2, "hum": 55}
    """

    # 1) Validar existencia y calidad del ID
    # Guard clause: si no hay id, no seguimos.
    if "id" not in row or not str(row["id"]).strip():
        raise ValueError("Falta 'id' o está vacío.")

    # Normalización: quitar espacios y usar mayúsculas
    sensor_id = str(row["id"]).strip().upper()

    # 2) Validar temperatura
    # Guard clause: debe existir la llave
    if "temp" not in row:
        raise ValueError("Falta 'temp' (temperatura).")

    # Guard clause: debe ser numérica
    if not isinstance(row["temp"], (int, float)):
        raise ValueError("'temp' debe ser numérica (int/float).")

    temp = float(row["temp"])

    # Guard clause: rango razonable para sensores agrícolas
    if temp < -10 or temp > 60:
        raise ValueError("'temp' fuera de rango [-10, 60].")

    # 3) Validar humedad
    if "hum" not in row:
        raise ValueError("Falta 'hum' (humedad).")

    if not isinstance(row["hum"], (int, float)):
```

```python
        raise ValueError("'hum' debe ser numérica (int/float).")

    hum = float(row["hum"])

    if hum < 0 or hum > 100:
        raise ValueError("'hum' fuera de rango [0, 100].")

    # Si llegamos aquí: el registro es válido.
    # Devolvemos una versión limpia.
    return {"id": sensor_id, "temp": temp, "hum": hum}


def main() -> None:
    # Dataset simulado: incluye registros válidos e inválidos a propósito.
    datos = [
        {"id": "sensor_01", "temp": 24.2, "hum": 55},  # válido
        {"id": "SENSOR_02", "temp": 25.1, "hum": 56},  # válido
        {"id": "SENSOR_XX", "temp": 999, "hum": 10},   # inválido (temp)
        {"id": "", "temp": 23.0, "hum": 50},           # inválido (id)
        {"id": "SENSOR_03", "temp": "N/A", "hum": 40}, # inválido (tipo temp)
    ]

    validos = 0
    motivos = {}

    # Recorremos el dataset, validamos y contamos errores por motivo.
    for row in datos:
        try:
            _ = validar_sensor_row(row)
            validos += 1
        except ValueError as e:
            motivo = str(e)
            motivos[motivo] = motivos.get(motivo, 0) + 1

    print(f"Validos: {validos}/{len(datos)}")
    print("Motivos de descarte:")
    for k, v in sorted(motivos.items(), key=lambda x: x[1], reverse=True):
        print(f"  - {k}: {v}")


if __name__ == "__main__":
    main()
```

**Ejercicio 1**

Modifica el validador para aceptar `hum` como string numérico (por ejemplo ““45““) convirtiéndolo a float, pero seguir rechazando ““N/A““.

# 10 CSV sucio → CSV limpio (sin pandas)

## 10.1 Por qué sin pandas

En esta semana, el objetivo es entender el pipeline y las validaciones sin “magia”.

## 10.2 Programa 2: Generar un CSV sucio (comentado)

Listing 27: Programa 2: Generar un CSV sucio (con errores intencionales)

```python
# Programa 2:
# Genera un archivo CSV con datos "sucios" (errores intencionales).
# Esto simula el mundo real: sensores que envían N/A, vacíos o valores imposibles.

from __future__ import annotations

import random


def generar_csv_sucio(path: str, n: int = 40) -> None:
    # Semilla para reproducibilidad: mismos datos en cada ejecución
    random.seed(7)

    sensores = ["SENSOR_01", "SENSOR_02", "SENSOR_03", "SENSOR_04"]

    # Abrimos el archivo en modo escritura y ponemos header
    with open(path, "w", encoding="utf-8") as f:
        f.write("id_sensor,fecha,temperatura,humedad\n")

        for i in range(n):
            s = random.choice(sensores)
            fecha = f"2026-01-01 00:{i:02d}:00"

            # Inyectamos errores en ciertas filas (por índice)
            if i in (5, 17):
                temp = "N/A"      # error: no numérico
            elif i in (8,):
                temp = 999        # error: imposible
            else:
                temp = round(random.uniform(18, 35), 1)

            if i in (3, 22):
                hum = ""          # error: faltante
            elif i in (12,):
                hum = 200         # error: fuera de rango
            else:
                hum = random.randint(30, 90)

            # Escribimos la fila al CSV
            f.write(f"{s},{fecha},{temp},{hum}\n")


if __name__ == "__main__":
    generar_csv_sucio("data/raw/sensores_sucio.csv", n=40)
    print("OK: generado data/raw/sensores_sucio.csv")
```

## 10.3 Programa 3: Limpiar el CSV (comentado)

Listing 28: Programa 3: Limpieza CSV sin pandas + reporte de errores

```python
# Programa 3:
```

```python
# Lee un CSV sucio línea por línea, valida y escribe un CSV limpio.
# Además genera un reporte de limpieza (cuántos descartó y por qué).

from __future__ import annotations


def validar_sensor_row(row: dict) -> dict:
    # Reutilizamos el validador (versión corta, misma lógica del Programa 1)
    if "id" not in row or not str(row["id"]).strip():
        raise ValueError("Falta 'id' o está vacío.")

    sensor_id = str(row["id"]).strip().upper()

    if "temp" not in row:
        raise ValueError("Falta 'temp'.")
    if not isinstance(row["temp"], (int, float)):
        raise ValueError("'temp' debe ser numérica.")
    temp = float(row["temp"])
    if temp < -10 or temp > 60:
        raise ValueError("'temp' fuera de rango [-10, 60].")

    if "hum" not in row:
        raise ValueError("Falta 'hum'.")
    if not isinstance(row["hum"], (int, float)):
        raise ValueError("'hum' debe ser numérica.")
    hum = float(row["hum"])
    if hum < 0 or hum > 100:
        raise ValueError("'hum' fuera de rango [0, 100].")

    return {"id": sensor_id, "temp": temp, "hum": hum}


def parse_row(linea: str) -> dict:
    # Convertimos una línea CSV en un dict con tipos correctos
    parts = linea.strip().split(",")
    if len(parts) != 4:
        raise ValueError("Fila corrupta: columnas != 4")

    id_sensor, fecha, temp_str, hum_str = parts

    # Temperatura: detectar faltantes o N/A
    if temp_str == "" or temp_str.upper() == "N/A":
        raise ValueError("Temperatura faltante o N/A")
    temp = float(temp_str)

    # Humedad: detectar faltantes
    if hum_str == "":
        raise ValueError("Humedad faltante")
    hum = float(hum_str)

    # Validación de rangos (fail fast)
    _ = validar_sensor_row({"id": id_sensor, "temp": temp, "hum": hum})

    # Si pasa: devolvemos un registro con fecha incluida
    return {"id": id_sensor.strip().upper(), "fecha": fecha, "temp": temp, "hum": hum}


def limpiar_csv(in_path: str, out_path: str, report_path: str) -> None:
    total = 0
    validos = 0
    motivos = {}

    # Leemos el CSV de entrada y escribimos uno nuevo (limpio)
```

```
    with open(in_path, "r", encoding="utf-8") as fin, open(out_path, "w", encoding="utf-8") as fout:
        header = fin.readline().strip()
        fout.write(header + "\n")

        for linea in fin:
            total += 1
            try:
                row = parse_row(linea)
                fout.write(f"{row['id']},{row['fecha']},{row['temp']},{row['hum']}\n")
                validos += 1
            except Exception as e:
                m = str(e)
                motivos[m] = motivos.get(m, 0) + 1

    # Escribir reporte de limpieza
    top = sorted(motivos.items(), key=lambda x: x[1], reverse=True)[:3]
    with open(report_path, "w", encoding="utf-8") as frep:
        frep.write("=== REPORTE LIMPIEZA CSV ===\n")
        frep.write(f"Total filas (sin header): {total}\n")
        frep.write(f"Validas: {validos}\n")
        frep.write(f"Descartadas: {total - validos}\n\n")
        frep.write("Top 3 motivos:\n")
        for motivo, c in top:
            frep.write(f"- {motivo}: {c}\n")


if __name__ == "__main__":
    # Nota: este script asume que el CSV sucio ya existe.
    # En el taller, primero se ejecuta Programa 2 para generarlo.
    limpiar_csv(
        in_path="data/raw/sensores_sucio.csv",
        out_path="data/processed/sensores_limpio.csv",
        report_path="reports/reporte_limpieza.txt",
    )
    print("OK: generado data/processed/sensores_limpio.csv y reports/reporte_limpieza.txt")
```

**Ejercicio 2**

Extiende `parse_row()` para registrar también un error cuando la fecha esté vacía o no tenga el formato `YYYY-MM-DD HH:MM:SS`.

# 11 HPC con NumPy — benchmark (loop vs vectorización)

## 11.1 Por qué medir

Si no mides, solo estás adivinando. Para comparar dos enfoques, necesitas un benchmark repetible.

## 11.2 Programa 4: Benchmark con `time.perf_counter()` (comentado)

Listing 29: Programa 4: Benchmark con `time.perf_counter()`

```python
# Programa 4:
# Compara el rendimiento de un bucle for vs una operación vectorizada en NumPy.
# Se usa time.perf_counter() porque ofrece alta resolución para medir tiempos.

from __future__ import annotations

import time
import numpy as np


def loop_for(x: np.ndarray) -> np.ndarray:
    # Este enfoque hace 1 millón de iteraciones en Python.
    # Cada iteración tiene overhead del intérprete.
    y = np.empty_like(x)
    for i in range(x.shape[0]):
        y[i] = np.sin(x[i]) + np.cos(x[i])
    return y


def vectorizado(x: np.ndarray) -> np.ndarray:
    # Este enfoque delega el trabajo a NumPy (internamente C optimizado).
    # Opera sobre todo el array sin iterar explícitamente en Python.
    return np.sin(x) + np.cos(x)


def main() -> None:
    n = 1_000_000
    x = np.random.rand(n).astype(np.float64)

    # Medición for-loop
    t0 = time.perf_counter()
    y1 = loop_for(x)
    t_for = time.perf_counter() - t0

    # Medición vectorizada
    t0 = time.perf_counter()
    y2 = vectorizado(x)
    t_vec = time.perf_counter() - t0

    # Validación numérica: deben ser muy parecidos
    diff = float(np.max(np.abs(y1 - y2)))

    # Speedup: cuántas veces es más rápido lo vectorizado
    speedup = t_for / t_vec if t_vec > 0 else float("inf")

    print(f"Tiempo for-loop:    {t_for:.4f} s")
    print(f"Tiempo vectorizado: {t_vec:.4f} s")
    print(f"Speedup:            {speedup:.2f}x")
    print(f"Max diff:           {diff:.6e}")


```

```python
if __name__ == "__main__":
    main()
```

**Ejercicio 3**

Ejecuta el benchmark 3 veces. Anota los 3 speedups y reporta el promedio en tu PDF.

# 12 Trabajar con Archivos .npy: El Formato Binario de NumPy

Cuando trabajas con datos numéricos de sensores agrícolas (miles o millones de mediciones), el formato CSV puede volverse ineficiente. NumPy ofrece un formato binario optimizado: `.npy`.

## 12.1 ¿Qué es un archivo .npy?

Un archivo `.npy` es el formato nativo de NumPy para almacenar arrays en disco de forma binaria. A diferencia de CSV (texto plano), `.npy` guarda los datos en el formato exacto que NumPy usa en memoria, lo que resulta en:

- **Velocidad**: Carga 10-50x más rápido que CSV.
- **Tamaño**: Archivos más pequeños (sin necesidad de convertir números a texto).
- **Precisión**: Mantiene la precisión numérica exacta (float64, int32, etc.).
- **Metadatos**: Guarda automáticamente el shape y dtype del array.

## 12.2 Comparación: CSV vs .npy

| Característica | CSV | .npy |
|---|---|---|
| Formato | Texto plano | Binario |
| Tamaño (1M valores) | ∼10 MB | ∼4 MB |
| Tiempo de carga | ∼2 segundos | ∼0.05 segundos |
| Velocidad relativa | 1x | **40x más rápido** |
| Legible en Excel | Sí | No |
| Uso típico | Compartir datos | Procesamiento científico |

## 12.3 Guardar y cargar arrays

### 12.3.1 Guardar datos en .npy

Listing 30: Guardar datos de sensor en formato .npy

```
import numpy as np

# Ejemplo 1: Guardar datos simulados
humedad = np.random.rand(365, 100) * 100  # 1 año, 100 zonas
np.save('humedad_finca.npy', humedad)

# Ejemplo 2: Guardar datos reales de un CSV procesado
datos_sensor = []
with open('sensor_2025.csv', 'r') as f:
    reader = csv.DictReader(f)
    for fila in reader:
        datos_sensor.append(float(fila['humedad']))

# Convertir a array y guardar
datos_np = np.array(datos_sensor)
np.save('humedad_real_2025.npy', datos_np)
```

### 12.3.2 Cargar datos desde .npy

Listing 31: Cargar y verificar datos guardados

```python
import numpy as np

# Cargar el array
humedad = np.load('humedad_finca.npy')

# Verificar metadatos
print(f"Shape: {humedad.shape}")      # Dimensiones
print(f"Dtype: {humedad.dtype}")      # Tipo de dato
print(f"Tamaño: {humedad.size:,}")    # Total de elementos

# Ejemplo de salida:
# Shape: (365, 100)
# Dtype: float64
# Tamaño: 36,500
```

El archivo se carga en milisegundos, sin necesidad de parsear texto ni convertir tipos.

## 12.4 Caso de uso: Sensor agrícola con 1 año de datos

Imagina un sensor IoT que mide humedad del suelo cada 5 minutos durante un año completo:

- Frecuencia: 12 lecturas/hora × 24 horas × 365 días = **105,120 mediciones**.
- CSV: Archivo de ∼2.5 MB, carga en ∼1.2 segundos.
- NPY: Archivo de ∼820 KB, carga en ∼0.03 segundos.

Si tu pipeline procesa estos datos 100 veces al día (análisis, validaciones, visualizaciones), ahorras **2 minutos diarios** solo en tiempo de lectura.

## 12.5 Comando rápido: Generar dataset de prueba

Para los talleres, puedes generar datasets simulados directamente desde la terminal:

```bash
# Generar matriz 365x100 con valores [0,100]
python -c "import numpy as np; np.save('humedad_finca.npy', np.random.rand(365, 100) * 100)"
```

Desglose del comando:

- `python -c "..."`: Ejecuta código Python en una sola línea.
- `np.random.rand(365, 100)`: Crea matriz 365×100 con valores aleatorios [0,1].
- `* 100`: Escala a rango [0,100] (porcentajes).
- `np.save(...)`: Guarda en formato binario .npy.

## 12.6 Convertir .npy y .npz de vuelta a CSV

Una ventaja del formato .npy es que siempre puedes convertirlo de vuelta a CSV cuando necesites compartir los datos o abrirlos en Excel. NumPy incluye la función `np.savetxt()` para esta tarea.

### 12.6.1 Conversión básica con np.savetxt()

Listing 32: Convertir .npy a CSV

```python
import numpy as np

# Cargar el archivo .npy
humedad = np.load('humedad_finca.npy')

# Guardar como CSV
np.savetxt('humedad_finca.csv', humedad, delimiter=',', fmt='%.2f')
```

**Parámetros importantes:**

- `delimiter=','`: Define el separador de columnas (coma para CSV).
- `fmt='%.2f'`: Formato numérico con 2 decimales. Ajusta según precisión necesaria.

### 12.6.2 Agregar nombres de columnas (header)

Para que el CSV sea más legible, puedes agregar un encabezado:

Listing 33: CSV con nombres de columnas

```python
import numpy as np

humedad = np.load('humedad_finca.npy')

# Crear header con nombres personalizados
header = ','.join([f'zona_{i+1}' for i in range(humedad.shape[1])])

np.savetxt('humedad_finca.csv',
           humedad,
           delimiter=',',
           fmt='%.2f',
           header=header,
           comments='')  # Evita el simbolo # en el header
```

El CSV resultante tendrá la primera fila con: `zona_1,zona_2,zona_3,...`

### 12.6.3 Convertir archivos .npz (múltiples arrays)

Si guardaste varios arrays en un archivo `.npz`, puedes extraerlos y convertir cada uno:

Listing 34: Convertir .npz a múltiples CSV

```python
import numpy as np

# Cargar el archivo .npz
datos = np.load('datos_finca.npz')

# Guardar cada array por separado
np.savetxt('humedad.csv', datos['humedad'], delimiter=',', fmt='%.2f')
np.savetxt('temperatura.csv', datos['temperatura'], delimiter=',', fmt='%.2f')
```

O combinarlos en un solo CSV con múltiples columnas:

Listing 35: Combinar arrays en un solo CSV

```python
import numpy as np

datos = np.load('datos_finca.npz')
```

```
4
5  # Combinar columnas horizontalmente
6  combinado = np.column_stack([datos['humedad'], datos['temperatura']])
7
8  np.savetxt('datos_completos.csv',
9             combinado,
10            delimiter=',',
11            fmt='%.2f',
12            header='humedad,temperatura',
13            comments='')
```

### 12.6.4 Script reutilizable de conversión

Para facilitar la conversión frecuente, puedes crear un script genérico:

Listing 36: Conversor .npy a CSV reutilizable

```
1  """
2  Convertir archivos .npy a CSV para compartir o visualizar
3  """
4  import numpy as np
5  import os
6
7  def npy_to_csv(npy_path, csv_path, decimales=2):
8      """
9      Convierte un archivo .npy a CSV
10
11     Args:
12         npy_path: Ruta del archivo .npy
13         csv_path: Ruta de salida .csv
14         decimales: Numero de decimales a guardar
15     """
16     datos = np.load(npy_path)
17
18     # Formato segun decimales
19     fmt = f'%.{decimales}f'
20
21     np.savetxt(csv_path, datos, delimiter=',', fmt=fmt)
22
23     print(f"OK: Convertido {npy_path} -> {csv_path}")
24     print(f"  Shape: {datos.shape}")
25     print(f"  Tamano CSV: {os.path.getsize(csv_path) / 1024:.1f} KB")
26
27 # Uso
28 if __name__ == "__main__":
29     npy_to_csv('humedad_finca.npy', 'humedad_finca.csv', decimales=1)
```

**Flujo de trabajo típico en proyectos agrícolas**

1. **Recolección**: Sensores IoT envían datos cada hora (guardados en CSV por el sistema).
2. **Procesamiento**: Conviertes CSV a .npy para análisis rápido con NumPy.
3. **Análisis**: Todo el pipeline científico trabaja con .npy (rápido).
4. **Entrega**: Conviertes resultados finales a CSV para compartir con agrónomos.

**Regla práctica**: Usa .npy internamente en tu código, y exporta a CSV solo cuando necesites compartir o visualizar en herramientas externas.

## 12.7 Bonus: Guardar múltiples arrays en un solo archivo

NumPy también ofrece `.npz` (comprimido) para guardar varios arrays relacionados:

Listing 37: Guardar múltiples variables en .npz

```python
import numpy as np

humedad = np.random.rand(365, 100)
temperatura = np.random.rand(365, 100)
timestamps = np.arange(365)

# Guardar todo junto
np.savez('datos_finca.npz',
         humedad=humedad,
         temperatura=temperatura,
         dias=timestamps)

# Cargar
datos = np.load('datos_finca.npz')
print(datos['humedad'].shape)
print(datos['temperatura'].shape)
```

Esto es ideal para datasets complejos donde múltiples sensores están relacionados temporalmente.

## 12.8 Cuándo usar cada formato

**Regla práctica**

- **Usa CSV** cuando necesites:
  - Abrir los datos en Excel o Google Sheets.
  - Compartir con personas sin Python.
  - Datos pequeños ($<$10,000 filas).
- **Usa .npy** cuando necesites:
  - Procesar datos repetidamente en Python.
  - Trabajar con arrays grandes ($>$100,000 valores).
  - Máxima velocidad y precisión numérica.

# 13 Lógica espacial con matrices (máscaras + np.where)

Las máscaras booleanas permiten seleccionar celdas sin usar `if` por celda.

## 13.1 Programa 5: Mapa de lote + acciones con `np.where` (comentado)

Listing 38: Programa 5: Mapa de lote (200x200) con máscaras y `np.where`

```python
# Programa 5:
# Simula un lote como una matriz de humedad (200x200).
# Calcula sequía/inundación usando máscaras booleanas.
# Luego decide acciones por celda con np.where().

from __future__ import annotations

import numpy as np


def render_ascii(mapa: np.ndarray, step: int = 10) -> str:
    """
    Render ASCII para ver un mapa grande en consola.
    step=10 significa que "muestreamos" 1 de cada 10 celdas (reduce tamaño).
    """
    lines = []
    for r in range(0, mapa.shape[0], step):
        lines.append("".join(mapa[r, ::step].tolist()))
    return "\n".join(lines)


def main() -> None:
    np.random.seed(7)

    # Matriz 200x200 con valores en [0,1): simula humedad normalizada
    humedad = np.random.rand(200, 200)

    # Umbrales de decisión
    umbral_sequia = 0.2
    umbral_inund = 0.85

    # Máscaras: arrays booleanos (True/False) del mismo tamaño
    sequia = humedad < umbral_sequia
    inundacion = humedad > umbral_inund

    # Porcentajes: mean() de booleanos equivale a proporción de True
    pct_sequia = 100.0 * float(sequia.mean())
    pct_inund = 100.0 * float(inundacion.mean())

    # Presupuesto hídrico:
    # Queremos subir humedad a 0.5 solo donde esté por debajo de 0.5.
    objetivo = 0.5
    agua_por_celda = np.where(humedad < objetivo, (objetivo - humedad), 0.0)
    agua_total = float(np.sum(agua_por_celda))

    # Acciones por celda (caracter):
    # R = regar (sequia)
    # D = drenaje (inundación)
    # . = normal
    acciones = np.where(sequia, "R", np.where(inundacion, "D", "."))

    print("=== Resumen lote (200x200) ===")
    print(f"% sequía (<{umbral_sequia}):     {pct_sequia:.2f}%")
    print(f"% inundación (>{umbral_inund}): {pct_inund:.2f}%")
```

```
    print(f"Agua total requerida (objetivo={objetivo}): {agua_total:.2f}")

    print("\nMapa ASCII (muestreo cada 10 celdas):")
    print(render_ascii(acciones, step=10))


if __name__ == "__main__":
    main()
```

**Ejercicio 4 (medio)**

Cambia los umbrales y analiza cómo cambia:

- % de sequía
- % de inundación
- agua total requerida

Registra 2 configuraciones y compáralas en tu PDF.

## Reto Final Integrador (Semana 2)

**Artefactos mínimos**

Debes dejar estos archivos en el repositorio:

- `data/raw/sensores_sucio.csv`
- `data/processed/sensores_limpio.csv`
- `reports/reporte_limpieza.txt`
- `reports/decision_hidrica.txt` (resumen del lote)
- `report/main.pdf`

**Checklist Git**

- Mínimo 3 commits claros.
- Los scripts se pueden ejecutar en orden sin errores.
- El PDF existe y resume resultados.

# 14 Fundamentos Algorítmicos para Series Temporales

Para desarrollar el taller de análisis de sensores IoT, es necesario dominar cuatro patrones fundamentales de manipulación de datos en Python estándar. A continuación, se describen las estrategias recomendadas para abordar cada fase del procesamiento.

## 14.1 Patrón de Agrupación (Group By)

Los datos de sensores suelen llegar en formato ”plano” (una lista de filas donde los IDs de los sensores se repiten). Para analizar cada sensor individualmente, debemos transformar esta lista plana en una estructura jerárquica: un diccionario de listas.

La lógica consiste en iterar sobre cada lectura y decidir dinámicamente:

- Si el ID del sensor **no existe** en nuestro diccionario, creamos una entrada con una lista vacía.
- Independientemente de lo anterior, agregamos la lectura actual a la lista de ese ID.

Listing 39: Pseudocódigo para agrupar datos

```
datos_agrupados = {}

for fila in datos_crudos:
    clave = fila['id']

    # Inicializacion perezosa (Lazy Initialization)
    if clave not in datos_agrupados:
        datos_agrupados[clave] = []

    datos_agrupados[clave].append(fila)
```

## 14.2 Limpieza y Conversión de Tipos (Casting)

El módulo `csv` de Python lee todos los datos como cadenas de texto (`str`). Es un error común intentar realizar operaciones matemáticas (suma, promedio, comparaciones) sin convertir previamente los datos.

La forma más eficiente (”Pythonic”) de extraer y convertir una columna completa de datos es utilizando *List Comprehensions*:

Listing 40: Extracción y conversión eficiente

```
# Incorrecto (generará TypeError):
# promedio = sum(lecturas) / len(lecturas)

# Correcto:
valores_float = [float(x['temperatura']) for x in lista_lecturas]
promedio = sum(valores_float) / len(valores_float)
```

## 14.3 Lógica de Estado para Eventos Temporales

Detectar un ”pico” de temperatura es trivial (un simple `if`). Sin embargo, detectar un evento sostenido (ej. ”temperatura alta por más de 1 hora”) requiere memoria.

Como procesamos las lecturas en orden cronológico, necesitamos una variable de estado o ”contador de racha”.

El algoritmo funciona como una máquina de estados simple:

1. Si la condición se cumple (ej. $T > 30$), incrementamos el contador.
2. Si la condición **no** se cumple, el evento se rompió: reiniciamos el contador a cero.
3. Verificamos si el contador alcanzó el umbral temporal deseado (ej. si cada lectura es de 5 min, 1 hora equivale a un contador de 12).

Listing 41: Algoritmo de detección de rachas

```
contador = 0
umbral_lecturas = 12  # Equivalente a 1 hora

for lectura in historial:
    if lectura > 30:
        contador += 1
    else:
        # La racha se rompio, verificar si fue critica antes de reiniciar
        if contador >= umbral_lecturas:
            registrar_alerta()
        contador = 0 # Reset
```

## 14.4 Persistencia de Resultados

Finalmente, para exportar estructuras complejas (como diccionarios) a un archivo CSV, utilizamos la clase `csv.DictWriter`. Es crucial definir previamente los nombres de las columnas (`fieldnames`) y asegurarse de escribir el encabezado antes de volcar los datos.

Listing 42: Escritura de diccionarios a CSV

```
columnas = ['id', 'promedio', 'maximo']

with open('resumen.csv', 'w') as f:
    writer = csv.DictWriter(f, fieldnames=columnas)
    writer.writeheader()
    # writerows espera una lista de diccionarios
    writer.writerows(lista_de_resultados)
```

# 15 Caso de Estudio Integrador: El Monitor de Voltaje

Para aterrizar los conceptos anteriores, analicemos un problema simplificado que simula la estructura del taller real.

## 15.1 El Problema

Supongamos que recibimos datos crudos de baterías de drones. Necesitamos:

1. Agrupar las lecturas por `id_dron`.
2. Calcular el voltaje promedio.
3. Detectar si el voltaje cae por debajo de 10V durante **3 lecturas consecutivas** (indicador de falla inminente).

## 15.2 Solución Completa Comentada

A continuación, se presenta un script que integra los cuatro patrones fundamentales explicados anteriormente. Observe cómo los datos fluyen desde la lista cruda hasta el reporte final.

Listing 43: Integración de conceptos: Análisis de Baterías (Detallado)

```
import csv # Necesario para leer/escribir archivos CSV (aunque aqui simulamos)

# 1. DATOS SIMULADOS: Imaginemos que esto es lo que leimos del archivo CSV
# Notese que los valores numericos vienen como cadenas de texto ('12.5')
datos_crudos = [
    {'id': 'D01', 'voltaje': '12.5'}, {'id': 'D02', 'voltaje': '11.0'},
    {'id': 'D01', 'voltaje': '9.0'},  {'id': 'D01', 'voltaje': '8.5'},
    {'id': 'D01', 'voltaje': '9.5'},  {'id': 'D02', 'voltaje': '10.8'}
]

def main():
    # --- PASO 1: AGROUPACION (De lista plana a diccionario organizado) ---
    grupos = {} # Diccionario vacio donde guardaremos { 'ID': [lista_de_filas] }

    for lectura in datos_crudos:       # Recorremos cada fila del CSV original
        d_id = lectura['id']           # Extraemos el ID para usarlo como clave

        if d_id not in grupos:         # Si es la primera vez que vemos este ID...
            grupos[d_id] = []          # ...inicializamos su lista vacia

        grupos[d_id].append(lectura)   # Agregamos la fila actual a la lista de ese ID

    reporte_final = [] # Aqui acumularemos los resultados procesados

    # --- PASO 2: PROCESAMIENTO (Iterar por cada grupo/Dron individualmente) ---
    # .items() nos da la clave (d_id) y el valor (lista_lecturas) al mismo tiempo
    for d_id, lista_lecturas in grupos.items():

        # --- PASO 3: CASTING (Limpieza de datos) ---
        # Usamos List Comprehension para crear una lista limpia solo con numeros float
        # Esto nos permite hacer calculos matematicos despues
        voltajes = [float(x['voltaje']) for x in lista_lecturas]

        # Calculo estadistico: suma total dividida por cantidad de elementos
        promedio = sum(voltajes) / len(voltajes)

        # --- PASO 4: LOGICA DE ESTADO (Detectar eventos en el tiempo) ---
```

```
        falla_detectada = False  # Bandera (flag) que inicia asumiento que todo esta bien
        contador_bajo = 0        # Variable de estado: cuenta lecturas seguidas problematicas

        for v in voltajes:       # Recorremos cronologicamente los voltajes de ESTE dron
            if v < 10.0:         # ¿El voltaje es bajo?
                contador_bajo += 1  # Si, aumentamos la racha
            else:
                contador_bajo = 0   # No, el voltaje es bueno. La racha se rompe (reset)

            # Verificamos si la racha alcanzo el limite critico (3 veces seguidas)
            if contador_bajo >= 3:
                falla_detectada = True  # Activamos la bandera de alerta
                break  # Opcional: Salimos del bucle porque ya confirmamos el fallo

        # Empaquetamos los resultados de este dron en un diccionario limpio
        reporte_final.append({
            'id': d_id,
            'voltaje_promedio': round(promedio, 2), # Redondeamos a 2 decimales
            'estado': 'CRITICO' if falla_detectada else 'OK' # Operador ternario para texto
        })

    # --- PASO 5: EXPORTACION (Mostrar resultados) ---
    print("Reporte Generado:")
    for fila in reporte_final:
        print(fila) # En el taller, aqui usarian csv.DictWriter

if __name__ == '__main__':
    main()
```

## 15.3 Análisis del Flujo

Si ejecutamos el código anterior, para el dron `D01` la secuencia de voltajes es: $[12.5, 9.0, 8.5, 9.5]$.

- $12.5$ (Contador = 0)
- $9.0$ (Contador = 1)
- $8.5$ (Contador = 2)
- $9.5$ (Contador = 3) $\rightarrow$ **¡Alerta Activada!**

Este mismo esquema mental es el que utilizará para el Taller 02, sustituyendo los voltajes por *temperatura* y ajustando el umbral de tiempo (que en su caso serán 12 lecturas equivalentes a 1 hora).

# 16 Introducción a LaTeX: Documentos Profesionales en Ingeniería

A partir de esta semana, entregarás tus reportes en **LaTeX** en lugar de editores visuales como Word o Google Docs. ¿Por qué? Porque LaTeX es el estándar en publicaciones científicas, ingeniería y documentación técnica.

## 16.1 ¿Qué es LaTeX?

LaTeX es un *sistema de composición de documentos* basado en texto plano. A diferencia de los editores "WYSIWYG" (What You See Is What You Get), en LaTeX **describes** cómo debe verse el documento mediante comandos, y el sistema lo compila automáticamente en un PDF profesional.

Piensa en LaTeX como **código para documentos**:

- Escribes instrucciones en un archivo `.tex` (texto plano).
- El compilador `xelatex` o `pdflatex` las convierte en un PDF final.
- El resultado es consistente, elegante y reproducible.

## 16.2 LaTeX vs. Editores tradicionales

| **Editor visual (Word/Docs)** | **LaTeX** |
|---|---|
| Formato manual (click en cada elemento) | Formato automático por comandos |
| Las fórmulas se descuadran al editar | Las fórmulas siempre quedan perfectas |
| Referencias a figuras hay que actualizarlas a mano | Referencias se actualizan automáticamente |
| Difícil versionar cambios | Compatible con Git (texto plano) |
| Cada colaborador puede cambiar el formato | El formato es consistente por diseño |

## 16.3 ¿Por qué LaTeX en este curso?

En ingeniería de software e inteligencia artificial, necesitas documentar:

- **Código fuente** con sintaxis resaltada.
- **Fórmulas matemáticas** (como funciones de costo en Machine Learning).
- **Tablas con resultados** de experimentos o datasets.
- **Gráficos y diagramas** referenciados automáticamente.

LaTeX maneja todo esto de forma profesional y estandarizada. Además, tus reportes quedan **versionados en Git** junto con tu código, permitiendo rastrear cambios históricos.

## 16.4 Tu primer documento LaTeX

Un documento mínimo tiene esta estructura:

Listing 44: Estructura básica de LaTeX

```latex
\documentclass{article}
\usepackage[utf8]{inputenc}
\usepackage[spanish]{babel}

\title{Reporte de Análisis de Sensores}
\author{Tu Nombre}
\date{\today}

\begin{document}

\maketitle

\section{Introducción}
Este reporte presenta el análisis de 1440 lecturas de sensores agrícolas.

\section{Resultados}
Se detectaron 3 periodos críticos de temperatura superior a 30°C.

\end{document}
```

Al compilar con `xelatex reporte.tex`, obtienes un PDF profesional con:

- Portada automática.
- Numeración de secciones.
- Márgenes, tipografía y espaciado estandarizados.

## 16.5 Compilación en Codespaces

En tu entorno de desarrollo ejecutas:

```
xelatex reporte.tex
```

Esto genera `reporte.pdf` junto con archivos auxiliares (`.aux`, `.log`). Solo el `.tex` y el `.pdf` se versiona en Git [web:41].

## 16.6 Ventajas para ingenieros de datos

- **Reproducibilidad**: El mismo `.tex` genera el mismo PDF en cualquier sistema.
- **Versionamiento**: Git muestra cambios línea por línea en el código LaTeX.
- **Automatización**: Puedes generar reportes desde scripts Python insertando resultados dinámicamente.
- **Profesionalismo**: Papers académicos, tesis y documentación técnica usan LaTeX como estándar.

En las próximas semanas aprenderás a incluir tablas de resultados, ecuaciones de modelos y fragmentos de código Python en tus reportes LaTeX.