

# **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**

<b>1 De Python básico a NumPy (el puente conceptual)</b>	<b>3</b>
1.1 Qué es un programa en Python (y qué significa “ejecutar”) . . . . .	3
1.2 Tipos de datos: números, texto y booleanos . . . . .	3
<b>2 Verificar y convertir tipos de datos en Python</b>	<b>4</b>
2.1 Tipos básicos: int, float, str . . . . .	4
2.2 Cómo saber de qué tipo es una variable . . . . .	4
2.3 Convertir entre tipos (casting) . . . . .	4
2.4 Ejemplo aplicado al agro . . . . .	5
<b>3 Control de flujo: decisiones y reglas</b>	<b>6</b>
3.1 Repetición: recorrer muchos datos . . . . .	6
3.2 Estructuras clave: listas y diccionarios . . . . .	6
<b>4 Funciones: empaquetar lógica reutilizable</b>	<b>7</b>
4.1 Errores y excepciones: fail fast en la práctica . . . . .	7
4.2 Archivos: leer texto y CSV sin librerías . . . . .	7
<b>5 Por qué NumPy: cuando el tamaño importa</b>	<b>8</b>
5.1 Máscaras booleanas: decisiones sobre mapas . . . . .	8
5.2 np.where: decidir por celda sin if por celda . . . . .	8
<b>6 Profundización: NumPy y su potencial</b>	<b>9</b>
6.1 El objeto central: ndarray (shape y dtype) . . . . .	9
6.2 Creación eficiente de arrays . . . . .	9
6.3 Indexing y slicing (leer sub-zonas del lote) . . . . .	9
<b>7 Vectorización con ufuncs: “sin for”</b>	<b>11</b>
7.1 Broadcasting: el superpoder . . . . .	11
7.2 Reducciones y el parámetro axis . . . . .	11
7.3 Máscaras booleanas (boolean indexing) . . . . .	12
7.4 np.where: decisiones vectorizadas . . . . .	12
<b>8 Visualización esencial con Matplotlib (desde NumPy)</b>	<b>13</b>
8.1 pyplot: la interfaz rápida . . . . .	13
8.2 Gráfico de línea (1D) . . . . .	13
8.3 Scatter (dispersión): detectar relación y outliers . . . . .	13
8.4 Heatmap (2D) con imshow + colorbar . . . . .	14

8.5 Guardar gráficos con savefig . . . . .	14
8.6 Ejercicios . . . . .	15
<b>9 Programación defensiva (Fail Fast)</b>	<b>18</b>
9.1 Por qué esto es ingeniería . . . . .	18
9.2 Programa 1: Validador con guard clauses (comentado) . . . . .	18
<b>10 CSV sucio → CSV limpio (sin pandas)</b>	<b>20</b>
10.1 Por qué sin pandas . . . . .	20
10.2 Programa 2: Generar un CSV sucio (comentado) . . . . .	20
10.3 Programa 3: Limpie el CSV (comentado) . . . . .	20
<b>11 HPC con NumPy — benchmark (loop vs vectorización)</b>	<b>23</b>
11.1 Por qué medir . . . . .	23
11.2 Programa 4: Benchmark con time.perf_counter() (comentado) . . . . .	23
<b>12 Trabajar con Archivos .npy: El Formato Binario de NumPy</b>	<b>25</b>
12.1 ¿Qué es un archivo .npy? . . . . .	25
12.2 Comparación: CSV vs .npy . . . . .	25
12.3 Guardar y cargar arrays . . . . .	25
12.3.1 Guardar datos en .npy . . . . .	25
12.3.2 Cargar datos desde .npy . . . . .	26
12.4 Caso de uso: Sensor agrícola con 1 año de datos . . . . .	26
12.5 Comando rápido: Generar dataset de prueba . . . . .	26
12.6 Convertir .npy y .npz de vuelta a CSV . . . . .	26
12.6.1 Conversión básica con np.savetxt() . . . . .	27
12.6.2 Agregar nombres de columnas (header) . . . . .	27
12.6.3 Convertir archivos .npz (múltiples arrays) . . . . .	27
12.6.4 Script reutilizable de conversión . . . . .	28
12.7 Bonus: Guardar múltiples arrays en un solo archivo . . . . .	29
12.8 Cuándo usar cada formato . . . . .	29
<b>13 Lógica espacial con matrices (máscaras + np.where)</b>	<b>30</b>
13.1 Programa 5: Mapa de lote + acciones con np.where (comentado) . . . . .	30
<b>14 Introducción a LaTeX: Documentos Profesionales en Ingeniería</b>	<b>32</b>
14.1 ¿Qué es LaTeX? . . . . .	32
14.2 LaTeX vs. Editores tradicionales . . . . .	32
14.3 ¿Por qué LaTeX en este curso? . . . . .	32
14.4 Tu primer documento LaTeX . . . . .	33
14.5 Compilación en Codespaces . . . . .	33
14.6 Ventajas para ingenieros de datos . . . . .	33

## 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

```

1 def main():
2     # aquí va la lógica principal
3     print("Hola")
4
5 if __name__ == "__main__":
6     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

```

1 temp = 24.2      # float
2 hum = 55        # int
3 sensor_id = "SENSOR_01"  # str
4 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

```
1 temperatura = 25.0
2 parcela_id = "P001"
3 cantidad_sacos = 30
4
5 print(type(temperatura))    # <class 'float'>
6 print(type(parcela_id))     # <class 'str'>
7 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

```
1 valor_bruto = "28.5"  # Esto es texto, no un número
2
3 temperatura = float(valor_bruto)  # str -> float
4 sacos_str = "30"
5 sacos = int(sacos_str)           # str -> int
6
7 print(temperatura + 1.5)        # 30.0
8 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

```
1 fila = {  
2     "parcela_id": "P010",  
3     "temperatura_c": "29.7",  
4     "humedad_suelo_pct": "38",  
5     "ph": "5.4"  
6 }  
7  
8 parcela_id = fila["parcela_id"]          # str  
9 temperatura = float(fila["temperatura_c"]) # str -> float  
10 humedad = int(fila["humedad_suelo_pct"])  # str -> int  
11 ph = float(fila["ph"])                  # str -> float  
12  
13 print(type(parcela_id))    # <class 'str'>  
14 print(type(temperatura))   # <class 'float'>  
15 print(type(humedad))       # <class 'int'>  
16 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

```
1 if hum < 40:  
2     print("Riego")  
3 elif hum > 80:  
4     print("Drenaje")  
5 else:  
6     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

```
1 humedades = [55, 60, 38, 92]  
2 for h in humedades:  
3     if h < 40:  
4         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)

```
1 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

```

1 def necesita_riego(hum):
2     return hum < 40
3
4 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

```

1 def dividir(a, b):
2     if b == 0:
3         raise ValueError("No se puede dividir por cero")
4     return a / b
5
6 try:
7     print(dividir(10, 0))
8 except ValueError as e:
9     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)

```

1 with open("sensores.csv", "r", encoding="utf-8") as f:
2     header = f.readline() # primera línea: nombres de columnas
3     for linea in f:
4         partes = linea.strip().split(",")
5         # 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

```
1 import numpy as np
2 humedad = np.random.rand(5, 5)
3 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

```
1 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

```

1 import numpy as np
2
3 a = np.array([1, 2, 3])           # 1D
4 b = np.array([[1.0, 2.0], [3.0, 4.0]]) # 2D
5
6 print(a.shape, a.dtype)
7 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

```

1 import numpy as np
2
3 Z = np.zeros((3, 4))      # matriz 3x4 llena de 0
4 O = np.ones((2, 2))       # matriz 2x2 llena de 1
5 x = np.arange(0, 10)      # [0..9]
6 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

```
1 import numpy as np
2
3 humedad = np.random.rand(100, 100)
4
5 zona_centro = humedad[40:60, 40:60]      # 20x20
6 primera_fila = humedad[0, :]                # 100 valores
7 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

```
1 import numpy as np
2
3 x = np.random.rand(5)
4 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

```
1 import numpy as np
2
3 A = np.random.rand(3, 4) # 3x4
4 b = np.array([10, 20, 30, 40]) # (4,)
5
6 # b se aplica a cada fila de A por broadcasting
7 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

```
1 import numpy as np
2
3 M = np.random.rand(3, 4)
4
5 total = M.sum()           # escalar: suma todo
6 por_fila = M.mean(axis=1)  # (3,) promedio por fila
7 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

```
1 import numpy as np
2
3 humedad = np.random.rand(100, 100)
4 sequia = humedad < 0.2
5
6 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

```
1 import numpy as np
2
3 humedad = np.random.rand(10, 10)
4 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

```

1 import numpy as np # importa la libreria numpy, con alias np
2 import matplotlib.pyplot as plt # importa la libreria matplotlib con su sublibreria pyplot y
   alias plt
3
4 # Simulación: 24 mediciones de humedad (una por hora)
5 # Evalúa la función en una línea que está en el intervalo [0,2pi)
6 # Con 24 puntos, no se incluye el extremo derecho
7
8 hum = 40 + 10*np.sin(np.linspace(0, 2*np.pi, 24))
9
10
11 plt.plot(hum, label="Humedad (%)") # Grafica la curva
12 plt.title("Humedad por hora")      # Título
13 plt.xlabel("Hora")                # Nombre del eje x
14 plt.ylabel("Humedad (%)")        # Nombre del eje y
15 plt.legend() # leyenda
16 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

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 np.random.seed(7) # Utiliza una semilla, para que los datos no cambien
5 temp = 20 + 10*np.random.rand(100) # Genera una serie aleatoria de temperaturas
6 hum = 30 + 40*np.random.rand(100) # Genera una serie aleatoria de humedades
7
8 plt.scatter(temp, hum, s=18) # Utiliza el tipo de gráfica scatter

```

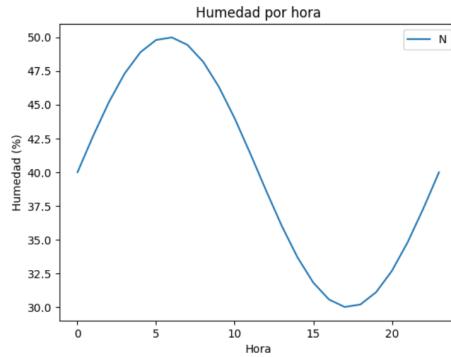


Figura 1: Gráfica generada con los comandos de python

```

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

```

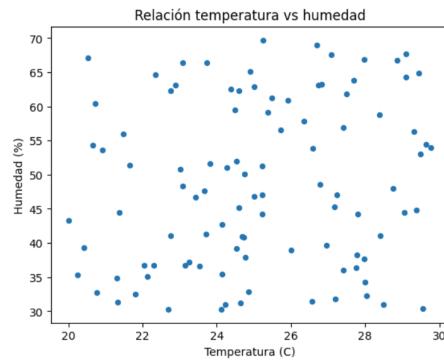


Figura 2: Diagrama de dispersión, generada con numpy

## 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`.

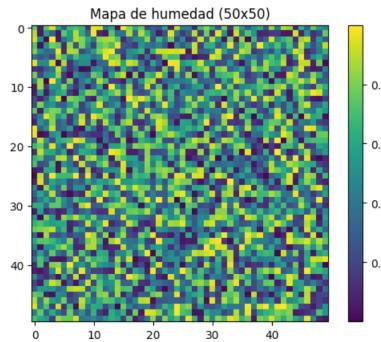


Figura 3: Mapa de calor con numpy

Listing 25: Guardar figura a mapa\_humedad.png

```

1 import os # Importa comandos del sistema
2 import numpy as np
3 import matplotlib.pyplot as plt
4
5 os.makedirs("reports", exist_ok=True) # Verifica que exista la carpeta reports
6
7 humedad = np.random.rand(50, 50)
8
9 plt.imshow(humedad, cmap="viridis", interpolation="nearest")
10 plt.colorbar()
11 plt.title("Mapa de humedad (50x50)")
12
13 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 → 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  $\text{\LaTeX}$  (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

```

1 # Programa 1:
2 # Este script enseña a validar datos antes de procesarlos.
3 # La idea es que el sistema "falle rápido" (fail fast) con mensajes claros,
4 # en lugar de continuar con datos inválidos y generar resultados incorrectos.
5
6 from __future__ import annotations
7
8
9 def validar_sensor_row(row: dict) -> dict:
10     """
11     Recibe un registro tipo dict y lo valida.
12     Si algo está mal: lanza ValueError (falla rápido).
13     Si todo está bien: devuelve un dict limpio/normalizado.
14
15     Formato esperado:
16         {"id": "SENSOR_01", "temp": 24.2, "hum": 55}
17     """
18
19     # 1) Validar existencia y calidad del ID
20     # Guard clause: si no hay id, no seguimos.
21     if "id" not in row or not str(row["id"]).strip():
22         raise ValueError("Falta 'id' o está vacío.")
23
24     # Normalización: quitar espacios y usar mayúsculas
25     sensor_id = str(row["id"]).strip().upper()
26
27     # 2) Validar temperatura
28     # Guard clause: debe existir la llave
29     if "temp" not in row:
30         raise ValueError("Falta 'temp' (temperatura).")
31
32     # Guard clause: debe ser numérica
33     if not isinstance(row["temp"], (int, float)):
34         raise ValueError("'temp' debe ser numérica (int/float).")
35
36     temp = float(row["temp"])
37
38     # Guard clause: rango razonable para sensores agrícolas
39     if temp < -10 or temp > 60:
40         raise ValueError("'temp' fuera de rango [-10, 60].")
41
42     # 3) Validar humedad
43     if "hum" not in row:
44         raise ValueError("Falta 'hum' (humedad).")
45
46     if not isinstance(row["hum"], (int, float)):
```

```
47     raise ValueError("'hum' debe ser numérica (int/float).")
48
49     hum = float(row["hum"])
50
51     if hum < 0 or hum > 100:
52         raise ValueError("'hum' fuera de rango [0, 100].")
53
54     # Si llegamos aquí: el registro es válido.
55     # Devolvemos una versión limpia.
56     return {"id": sensor_id, "temp": temp, "hum": hum}
57
58
59 def main() -> None:
60     # Dataset simulado: incluye registros válidos e inválidos a propósito.
61     datos = [
62         {"id": "sensor_01", "temp": 24.2, "hum": 55}, # válido
63         {"id": "SENSOR_02", "temp": 25.1, "hum": 56}, # válido
64         {"id": "SENSOR_XX", "temp": 999, "hum": 10}, # inválido (temp)
65         {"id": "", "temp": 23.0, "hum": 50}, # inválido (id)
66         {"id": "SENSOR_03", "temp": "N/A", "hum": 40}, # inválido (tipo temp)
67     ]
68
69     validos = 0
70     motivos = {}
71
72     # Recorremos el dataset, validamos y contamos errores por motivo.
73     for row in datos:
74         try:
75             _ = validar_sensor_row(row)
76             validos += 1
77         except ValueError as e:
78             motivo = str(e)
79             motivos[motivo] = motivos.get(motivo, 0) + 1
80
81     print(f"Validos: {validos}/{len(datos)}")
82     print("Motivos de descarte:")
83     for k, v in sorted(motivos.items(), key=lambda x: x[1], reverse=True):
84         print(f" - {k}: {v}")
85
86
87 if __name__ == "__main__":
88     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)

```

1 # Programa 2:
2 # Genera un archivo CSV con datos "sucios" (errores intencionales).
3 # Esto simula el mundo real: sensores que envían N/A, vacíos o valores imposibles.
4
5 from __future__ import annotations
6
7 import random
8
9
10 def generar_csv_sucio(path: str, n: int = 40) -> None:
11     # Semilla para reproducibilidad: mismos datos en cada ejecución
12     random.seed(7)
13
14     sensores = ["SENSOR_01", "SENSOR_02", "SENSOR_03", "SENSOR_04"]
15
16     # Abrimos el archivo en modo escritura y ponemos header
17     with open(path, "w", encoding="utf-8") as f:
18         f.write("id_sensor,fecha,temperatura,humedad\n")
19
20         for i in range(n):
21             s = random.choice(sensores)
22             fecha = f"2026-01-01 00:{i:02d}:00"
23
24             # Inyectamos errores en ciertas filas (por índice)
25             if i in (5, 17):
26                 temp = "N/A"      # error: no numérico
27             elif i in (8,):
28                 temp = 999       # error: imposible
29             else:
30                 temp = round(random.uniform(18, 35), 1)
31
32             if i in (3, 22):
33                 hum = ""        # error: faltante
34             elif i in (12,):
35                 hum = 200       # error: fuera de rango
36             else:
37                 hum = random.randint(30, 90)
38
39             # Escribimos la fila al CSV
40             f.write(f"{s},{fecha},{temp},{hum}\n")
41
42
43 if __name__ == "__main__":
44     generar_csv_sucio("data/raw/sensores_sucio.csv", n=40)
45     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

```
1 # Programa 3:
```

```

2 # Lee un CSV sucio línea por línea, valida y escribe un CSV limpio.
3 # Además genera un reporte de limpieza (cuántos descartó y por qué).
4
5 from __future__ import annotations
6
7
8 def validar_sensor_row(row: dict) -> dict:
9     # Reutilizamos el validador (versión corta, misma lógica del Programa 1)
10    if "id" not in row or not str(row["id"]).strip():
11        raise ValueError("Falta 'id' o está vacío.")
12
13    sensor_id = str(row["id"]).strip().upper()
14
15    if "temp" not in row:
16        raise ValueError("Falta 'temp'.")
17    if not isinstance(row["temp"], (int, float)):
18        raise ValueError("'temp' debe ser numérica.")
19    temp = float(row["temp"])
20    if temp < -10 or temp > 60:
21        raise ValueError("'temp' fuera de rango [-10, 60].")
22
23    if "hum" not in row:
24        raise ValueError("Falta 'hum'.")
25    if not isinstance(row["hum"], (int, float)):
26        raise ValueError("'hum' debe ser numérica.")
27    hum = float(row["hum"])
28    if hum < 0 or hum > 100:
29        raise ValueError("'hum' fuera de rango [0, 100].")
30
31    return {"id": sensor_id, "temp": temp, "hum": hum}
32
33
34 def parse_row(linea: str) -> dict:
35     # Convertimos una línea CSV en un dict con tipos correctos
36     parts = linea.strip().split(",")
37     if len(parts) != 4:
38         raise ValueError("Fila corrupta: columnas != 4")
39
40     id_sensor, fecha, temp_str, hum_str = parts
41
42     # Temperatura: detectar faltantes o N/A
43     if temp_str == "" or temp_str.upper() == "N/A":
44         raise ValueError("Temperatura faltante o N/A")
45     temp = float(temp_str)
46
47     # Humedad: detectar faltantes
48     if hum_str == "":
49         raise ValueError("Humedad faltante")
50     hum = float(hum_str)
51
52     # Validación de rangos (fail fast)
53     _ = validar_sensor_row({"id": id_sensor, "temp": temp, "hum": hum})
54
55     # Si pasa: devolvemos un registro con fecha incluida
56     return {"id": id_sensor.strip().upper(), "fecha": fecha, "temp": temp, "hum": hum}
57
58
59 def limpiar_csv(in_path: str, out_path: str, report_path: str) -> None:
60     total = 0
61     validos = 0
62     motivos = []
63
64     # Leemos el CSV de entrada y escribimos uno nuevo (limpio)

```

```
65  with open(in_path, "r", encoding="utf-8") as fin, open(out_path, "w", encoding="utf-8") as
66      fout:
67          header = fin.readline().strip()
68          fout.write(header + "\n")
69
70      for linea in fin:
71          total += 1
72          try:
73              row = parse_row(linea)
74              fout.write(f"{row['id']},{row['fecha']},{row['temp']},{row['hum']}\n")
75              validos += 1
76          except Exception as e:
77              m = str(e)
78              motivos[m] = motivos.get(m, 0) + 1
79
80      # Escribir reporte de limpieza
81      top = sorted(motivos.items(), key=lambda x: x[1], reverse=True)[:3]
82      with open(report_path, "w", encoding="utf-8") as frep:
83          frep.write("==> REPORTE LIMPIEZA CSV ==>\n")
84          frep.write(f"Total filas (sin header): {total}\n")
85          frep.write(f"Validadas: {validos}\n")
86          frep.write(f"Descartadas: {total - validos}\n\n")
87          frep.write("Top 3 motivos:\n")
88          for motivo, c in top:
89              frep.write(f"- {motivo}: {c}\n")
90
91 if __name__ == "__main__":
92     # Nota: este script asume que el CSV sucio ya existe.
93     # En el taller, primero se ejecuta Programa 2 para generararlo.
94     limpiar_csv(
95         in_path="data/raw/sensores_sucio.csv",
96         out_path="data/processed/sensores_limpio.csv",
97         report_path="reports/reporte_limpieza.txt",
98     )
99     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()

```

1 # Programa 4:
2 # Compara el rendimiento de un bucle for vs una operación vectorizada en NumPy.
3 # Se usa time.perf_counter() porque ofrece alta resolución para medir tiempos.
4
5 from __future__ import annotations
6
7 import time
8 import numpy as np
9
10
11 def loop_for(x: np.ndarray) -> np.ndarray:
12     # Este enfoque hace 1 millón de iteraciones en Python.
13     # Cada iteración tiene overhead del intérprete.
14     y = np.empty_like(x)
15     for i in range(x.shape[0]):
16         y[i] = np.sin(x[i]) + np.cos(x[i])
17     return y
18
19
20 def vectorizado(x: np.ndarray) -> np.ndarray:
21     # Este enfoque delega el trabajo a NumPy (internamente C optimizado).
22     # Opera sobre todo el array sin iterar explícitamente en Python.
23     return np.sin(x) + np.cos(x)
24
25
26 def main() -> None:
27     n = 1_000_000
28     x = np.random.rand(n).astype(np.float64)
29
30     # Medición for-loop
31     t0 = time.perf_counter()
32     y1 = loop_for(x)
33     t_for = time.perf_counter() - t0
34
35     # Medición vectorizada
36     t0 = time.perf_counter()
37     y2 = vectorizado(x)
38     t_vec = time.perf_counter() - t0
39
40     # Validación numérica: deben ser muy parecidos
41     diff = float(np.max(np.abs(y1 - y2)))
42
43     # Speedup: cuántas veces es más rápido lo vectorizado
44     speedup = t_for / t_vec if t_vec > 0 else float("inf")
45
46     print(f"Tiempo for-loop: {t_for:.4f} s")
47     print(f"Tiempo vectorizado: {t_vec:.4f} s")
48     print(f"Speedup: {speedup:.2f}x")
49     print(f"Max diff: {diff:.6e}")
50
51

```

```
52 | if __name__ == "__main__":
53 |     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	<b>40x más rápido</b>
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

```

1 import numpy as np
2
3 # Ejemplo 1: Guardar datos simulados
4 humedad = np.random.rand(365, 100) * 100 # 1 año, 100 zonas
5 np.save('humedad_finca.npy', humedad)
6
7 # Ejemplo 2: Guardar datos reales de un CSV procesado
8 datos_sensor = []
9 with open('sensor_2025.csv', 'r') as f:
10     reader = csv.DictReader(f)
11     for fila in reader:
12         datos_sensor.append(float(fila['humedad']))
13
14 # Convertir a array y guardar
15 datos_np = np.array(datos_sensor)
16 np.save('humedad_real_2025.npy', datos_np)

```

### 12.3.2 Cargar datos desde .npy

Listing 31: Cargar y verificar datos guardados

```

1 import numpy as np
2
3 # Cargar el array
4 humedad = np.load('humedad_finca.npy')
5
6 # Verificar metadatos
7 print(f"Shape: {humedad.shape}")      # Dimensiones
8 print(f"Dtype: {humedad.dtype}")       # Tipo de dato
9 print(f"Tamaño: {humedad.size:,}")     # Total de elementos
10
11 # Ejemplo de salida:
12 # Shape: (365, 100)
13 # Dtype: float64
14 # 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:

```

1 # Generar matriz 365x100 con valores [0,100]
2 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

```

1 import numpy as np
2
3 # Cargar el archivo .npy
4 humedad = np.load('humedad_finca.npy')
5
6 # Guardar como CSV
7 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

```

1 import numpy as np
2
3 humedad = np.load('humedad_finca.npy')
4
5 # Crear header con nombres personalizados
6 header = ','.join([f'zona_{i+1}' for i in range(humedad.shape[1])])
7
8 np.savetxt('humedad_finca.csv',
9             humedad,
10            delimiter=',',
11            fmt='%.2f',
12            header=header,
13            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

```

1 import numpy as np
2
3 # Cargar el archivo .npz
4 datos = np.load('datos_finca.npz')
5
6 # Guardar cada array por separado
7 np.savetxt('humedad.csv', datos['humedad'], delimiter=',', fmt='%.2f')
8 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

```

1 import numpy as np
2
3 datos = np.load('datos_finca.npz')

```

```

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

```

#### 12.6.4 Script reutilizable de conversión

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

Listing 36: Conversor .npy a CSV reutilizable

```

"""
Convertir archivos .npy a CSV para compartir o visualizar
"""

import numpy as np
import os

def npy_to_csv(npy_path, csv_path, decimales=2):
    """
    Convierte un archivo .npy a CSV

    Args:
        npy_path: Ruta del archivo .npy
        csv_path: Ruta de salida .csv
        decimales: Número de decimales a guardar
    """

    datos = np.load(npy_path)

    # Formato según decimales
    fmt = f'%.{decimales}f'

    np.savetxt(csv_path, datos, delimiter=',', fmt=fmt)

    print(f"OK: Convertido {npy_path} -> {csv_path}")
    print(f"  Shape: {datos.shape}")
    print(f"  Tamaño CSV: {os.path.getsize(csv_path) / 1024:.1f} KB")

# Uso
if __name__ == "__main__":
    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

```
1 import numpy as np
2
3 humedad = np.random.rand(365, 100)
4 temperatura = np.random.rand(365, 100)
5 timestamps = np.arange(365)
6
7 # Guardar todo junto
8 np.savez('datos_finca.npz',
9          humedad=humedad,
10         temperatura=temperatura,
11         dias=timestamps)
12
13 # Cargar
14 datos = np.load('datos_finca.npz')
15 print(datos['humedad'].shape)
16 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

```

1 # Programa 5:
2 # Simula un lote como una matriz de humedad (200x200).
3 # Calcula sequía/inundación usando máscaras booleanas.
4 # Luego decide acciones por celda con np.where().
5
6 from __future__ import annotations
7
8 import numpy as np
9
10
11 def render_ascii(mapa: np.ndarray, step: int = 10) -> str:
12     """
13     Render ASCII para ver un mapa grande en consola.
14     step=10 significa que "muestreamos" 1 de cada 10 celdas (reduce tamaño).
15     """
16     lines = []
17     for r in range(0, mapa.shape[0], step):
18         lines.append("".join(mapa[r, ::step].tolist()))
19     return "\n".join(lines)
20
21
22 def main() -> None:
23     np.random.seed(7)
24
25     # Matriz 200x200 con valores en [0,1]: simula humedad normalizada
26     humedad = np.random.rand(200, 200)
27
28     # Umbral de decisión
29     umbral_sequia = 0.2
30     umbral_inund = 0.85
31
32     # Máscaras: arrays booleanos (True/False) del mismo tamaño
33     sequia = humedad < umbral_sequia
34     inundacion = humedad > umbral_inund
35
36     # Porcentajes: mean() de booleanos equivale a proporción de True
37     pct_sequia = 100.0 * float(sequia.mean())
38     pct_inund = 100.0 * float(inundacion.mean())
39
40     # Presupuesto hídrico:
41     # Queremos subir humedad a 0.5 solo donde esté por debajo de 0.5.
42     objetivo = 0.5
43     agua_por_celda = np.where(humedad < objetivo, (objetivo - humedad), 0.0)
44     agua_total = float(np.sum(agua_por_celda))
45
46     # Acciones por celda (caracter):
47     # R = regar (sequia)
48     # D = drenaje (inundación)
49     # . = normal
50     acciones = np.where(sequia, "R", np.where(inundacion, "D", "."))
51
52     print("== Resumen lote (200x200) ==")
53     print(f"% sequía (<{umbral_sequia}): {pct_sequia:.2f}%")
54     print(f"% inundación (>{umbral_inund}): {pct_inund:.2f}%")

```

```
55 print(f"Agua total requerida (objetivo={objetivo}): {agua_total:.2f}")  
56  
57 print("\nMapa ASCII (muestreo cada 10 celdas):")  
58 print(render_ascii(acciones, step=10))  
59  
60  
61 if __name__ == "__main__":  
62     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.

## 14 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.

### 14.1 ¿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.

### 14.2 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

### 14.3 ¿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.

## 14.4 Tu primer documento LaTeX

Un documento mínimo tiene esta estructura:

Listing 39: Estructura básica de LaTeX

```
1 \documentclass{article}
2 \usepackage[utf8]{inputenc}
3 \usepackage[spanish]{babel}
4
5 \title{Reporte de Análisis de Sensores}
6 \author{Tu Nombre}
7 \date{\today}
8
9 \begin{document}
10
11 \maketitle
12
13 \section{Introducción}
14 Este reporte presenta el análisis de 1440 lecturas de sensores agrícolas.
15
16 \section{Resultados}
17 Se detectaron 3 períodos críticos de temperatura superior a 30°C.
18
19 \end{document}
```

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

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

## 14.5 Compilación en Codespaces

En tu entorno de desarrollo ejecutas:

```
1 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].

## 14.6 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.

## 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.