Semana 02 — Computación Científica y HPC para el Agro

Desarrollo + ejemplos comentados + ejercicios

Curso de IA Aplicada al Agro

Enero 2026

Índice general

1 Capítulo 1: De Python básico a NumPy (el puente conceptual)

Este capítulo 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.

Meta del capítulo

Al finalizar este capítulo 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
```

1.3 Control de flujo: decisiones y reglas

Las reglas agronómicas se expresan como condiciones:

Listing 3: Decisión simple con if/elif/else

```python
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.

1.4 Repetición: recorrer muchos datos

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

Listing 4: Recorrer una lista con for

```python
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.

1.5 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 5: Registro tipo diccionario (una fila de sensor)

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

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

1.6 Funciones: empaquetar lógica reutilizable

Cuando una regla se repite, se convierte en función:

Listing 6: 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.

1.7 Errores y excepciones: fail fast en la práctica

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

Listing 7: 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.

1.8 Archivos: leer texto y CSV sin librerías

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

Listing 8: 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.

1.9 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)`

1.10 Máscaras booleanas: decisiones sobre mapas

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

Listing 9: Máscara booleana en matriz

```python
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.

1.11 `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 10: 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.

2 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**.

2.1 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 11: 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.

2.2 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 12: Creación rápida de matrices

```python
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
```

2.3 Indexing y slicing (leer sub-zonas del lote)

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

Listing 13: 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]`.

2.4 Vectorización con ufuncs: “sin for”

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

Listing 14: 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.

2.5 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 15: 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`.

2.6 Reducciones y el parámetro `axis`

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

Listing 16: 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).

2.7 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 17: 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
```

2.8 `np.where`: decisiones vectorizadas

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

Listing 18: 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.

3 Capítulo: 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`.

3.1 `pyplot`: la interfaz rápida

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

3.2 Gráfico de línea (1D)

Listing 19: 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 3.2

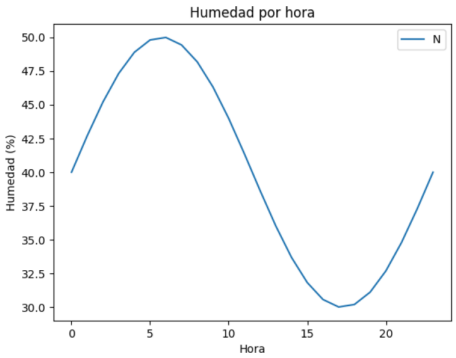

Figura 1: Gráfica generada con los comandos de python

3.3 Scatter (dispersión): detectar relación y outliers

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

Listing 20: 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
plt.title("Relación temperatura vs humedad")
plt.xlabel("Temperatura (C)")
plt.ylabel("Humedad (%)")
plt.show()
```

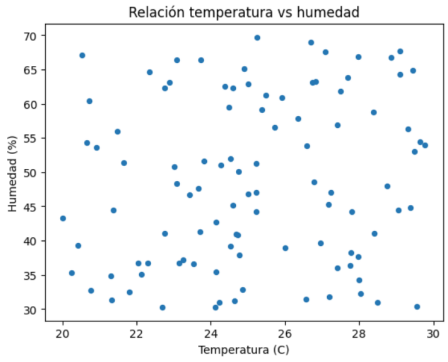

Figura 2: Diagrama de dispersión, generada con numpy

3.4 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 21: Heatmap: humedad 2D con colorbar

```
import numpy as np
```

```python
import matplotlib.pyplot as plt

humedad = np.random.rand(50, 50) # Genera una matriz de valores aleatorios

plt.imshow(humedad, cmap="viridis", interpolation="nearest")  # Muestra el mapa de calor
plt.colorbar()                                                # Colorea
plt.title("Mapa de humedad (50x50)")
plt.show()
```

Se puede apreciar el mapa de calor en la figura 3

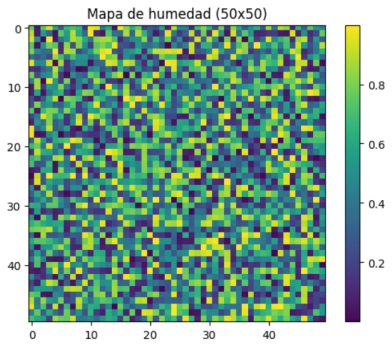

Figura 3: Mapa de calor con numpy

3.5 Guardar gráficos con `savefig`

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

Listing 22: 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.

3.6 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

En Semana 01 construiste hábitos: terminal, organización y Git. En Semana 02 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).

4 Capítulo I: Programación defensiva (Fail Fast)

4.1 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.

4.2 Programa 1: Validador con guard clauses (comentado)

Listing 23: 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"".

5 Capítulo II: CSV sucio → CSV limpio (sin pandas)

5.1 Por qué sin pandas

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

5.2 Programa 2: Generar un CSV sucio (comentado)

Listing 24: 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")
```

5.3 Programa 3: Limpiar el CSV (comentado)

Listing 25: 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`.

6 Capítulo III: HPC con NumPy — benchmark (loop vs vectorización)

6.1 Por qué medir

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

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

Listing 26: 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}")
```

```
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.

7 Capítulo IV: Lógica espacial con matrices (máscaras + np.where)

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

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

Listing 27: 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.

8 Capítulo V: Reporte en LaTeX (manual en Semana 2)

Qué debe incluir tu PDF (1 página)

- Resumen (5–8 líneas).
- Resultados del benchmark (tiempos + speedup).
- Resultados del lote (porcentajes + agua total).
- Qué validaciones aplicaste y por qué.

Ejercicio 5 (entrega)

En `workspace/report/main.tex` completa tu reporte y compila:

```
latexmk -xelatex -interaction=nonstopmode main.tex
```

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.