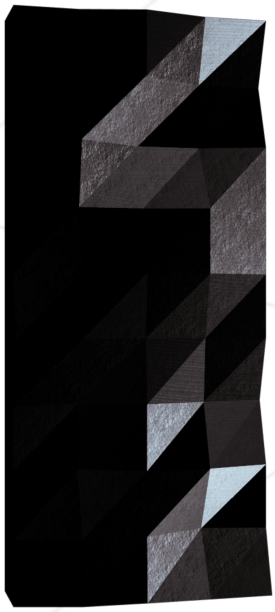

Guarda del Huerto

Ingeniería de Datos para Agricultura Inteligente

Resumen: ¡Construye pipelines de datos resistentes para tu huerto inteligente! Aprende a manejar fallos de sensores, procesar flujos de datos agrícolas y crear sistemas de monitorización robustos que mantengan tu invernadero digital en plena forma.

Versión: 1.0

Índice general

Capítulo I

Prólogo

¡Te damos la bienvenida al mundo de la ingeniería de datos agrícolas!

Sobre la base de tus fundamentos de Python (Módulo 00) y las clases de monitorización del huerto (Módulo 01), ahora cuentas con las competencias para afrontar los retos reales de la agricultura inteligente. En la agricultura moderna, los datos fluyen como el agua a través de los sistemas de riego: las lecturas de sensores llegan de forma continua, las APIs meteorológicas ofrecen pronósticos y los dispositivos IoT monitorizan todo, desde el pH del suelo hasta la humedad del invernadero.

Pero ¿qué sucede cuando esta canalización de datos encuentra turbulencias? ¿Cuando los sensores fallan durante la temporada de cosecha? ¿Cuando las conexiones de red se caen en periodos críticos de monitorización? ¿Cuando datos corruptos amenazan con disparar riegos falsos?

Quien se dedica profesionalmente a la ingeniería de datos agrícolas sabe que los sistemas robustos no se construyen para evitar fallos: se diseñan para **gestionar de forma adecuada lo inesperado**. Tu invernadero digital debe ser tan resiliente como la propia naturaleza.

El sistema de **gestión de excepciones** de Python es tu caja de herramientas para construir pipelines de datos a prueba de fallos en agricultura. Aprenderás a **capturar anomalías de sensores**, **crear alertas agrícolas personalizadas** y **garantizar la integridad de los datos** incluso cuando la Madre Naturaleza (o la Ley de Murphy) haga de las suyas.

En este proyecto, evolucionarás de una programación básica a la ingeniería de datos agrícolas, construyendo sistemas de monitorización que siguen creciendo incluso cuando ocurre lo inesperado.

Capítulo II

Instrucciones sobre la IA

Contexto

Durante tu proceso de aprendizaje, la IA puede ayudarte con muchas tareas diferentes. Tómate el tiempo necesario para explorar las diversas capacidades de las herramientas de IA y cómo pueden apoyarte con tu trabajo. Sin embargo, siempre debes abordarlas con precaución y evaluar de forma crítica los resultados. Ya sea código, documentación, ideas o explicaciones técnicas, nunca podrás saber con total certeza si tu pregunta está bien formulada o si el contenido generado es el adecuado. Las personas que te rodean son tu recurso más valioso para ayudarte a evitar errores y puntos ciegos.

Mensaje principal:

- Utiliza la IA para reducir las tareas repetitivas o tediosas.
- Desarrolla habilidades de prompting, ya sea para programacion o para otros temas, que beneficiarán tu futura carrera.
- Aprende cómo funcionan los sistemas de IA para anticipar de forma eficiente y evitar los riesgos comunes, sesgos y problemas éticos.
- Sigue trabajando con tus compañeros para desarrollar tanto habilidades técnicas como habilidades transversales.
- Utiliza únicamente contenido generado por IA que entiendas completamente y del cual puedas responsabilizarte.

Reglas para estudiantes:

- Debes tomarte el tiempo necesario para explorar las herramientas de IA y comprender cómo funcionan, para poder utilizarlas de manera ética y reducir los sesgos potenciales.
- Debes reflexionar sobre tu problema antes de dar instrucciones a la IA. Esto te ayuda a escribir preguntas, instrucciones o conjuntos de datos más claros, detalladas y relevantes utilizando un vocabulario preciso.

- Debes desarrollar el hábito de revisar, cuestionar y probar sistemáticamente cualquier contenido generado por la IA.
- Debes buscar siempre la revisión de otras personas, no te limites a confiar en tu propia validación.

Resultados de esta etapa:

- Desarrollar habilidades de prompting tanto generales como de ámbito especifico.
- Aumentar tu productividad con un uso eficaz de las herramientas de IA.
- Seguir fortaleciendo el pensamiento computacional, la resolución de problemas, la adaptabilidad y la colaboración.

Comentarios y ejemplos:

- Ten en cuenta que la IA puede no tener la respuesta correcta porque esa respuesta no esté ni siquiera en Internet. Además, si te da soluciones incorrectas, intenta no insistir y busca ayuda entre las personas que te rodean. Vas a ahorrarte tiempo y vas a sumar en compresión.
- Vas a enfretarte con frecuencia a situaciones (como exámenes o evaluaciones) donde debes demostrar una comprensión real. Prepárate, sigue construyendo tanto tus habilidades técnicas como transversales.
- Explicar tu razonamiento y debatir con otras personas suele revelar lagunas en tu comprensión de un concepto. Prioriza el aprendizaje entre pares.
- Lo normal es que la herramienta de IA que utilices no conozca tu contexto específico (a menos que se lo indiques), así que te dará respuestas genéricas. Si buscas información más adecuada y más precisa en relación a tu entorno cercano, confía en el resto de estudiantes.
- Donde la IA tiende a generar la respuesta más probable, el resto de estudiantes puede proporcionar perspectivas alternativas y matices valiosos. Confía en la comunidad de 42 como un punto de control de calidad.

✓ Buenas prácticas:

> Le pregunto a la IA: "¿Cómo pruebo una función de ordenación?"Me da algunas ideas. Las pruebo y reviso los resultados con otra persona. Refinamos el enfoque de manera conjunta.

✗ Bad practice:

> Le pido a la IA que escriba una función completa, la copio y la pego en mi proyecto. Durante la evaluación entre pares, no puedo explicar qué hace ni por qué. Pierdo credibilidad. Suspendo mi proyecto.

✓ Good practice:

Utilizo la IA para ayudarme a diseñar un parser. Luego, reviso la lógica con otra persona. Encontramos dos errores y lo reescribimos juntos: mejor, más limpio y comprendiendo al 100

✗ Bad practice:

Dejo que Copilot genere mi código para una parte clave de mi proyecto. Compila, pero no puedo explicar cómo maneja los pipes. Durante la evaluación, no puedo justificarlo y suspendo mi proyecto.

Capítulo III

Introducción

¡Te damos la bienvenida a Guarda del Huerto: Ingeniería de Datos para Agricultura Inteligente!

Sobre la base de tu experiencia previa en monitoreo del huerto, ahora dominarás las habilidades críticas de **ingeniería sólida de pipelines de datos** para sistemas agrícolas.

Descubrirás:

- Cómo **validar y limpiar flujos de datos agrícolas** en tiempo real.
- Qué **modos de fallo** existen en redes de sensores IoT.
- Cómo **crear alertas agrícolas personalizadas** para la supervisión específica de cultivos.
- Técnicas esenciales para la **tolerancia a fallos** y la recuperación en pipelines de datos.
- Cómo **garantizar la integridad de los datos** en sistemas agrícolas distribuidos.

Cada ejercicio construye un componente en tu plataforma de datos para agricultura inteligente, avanzando desde la validación básica de sensores hasta sistemas integrales de monitorización agrícola.

IMPORTANTE: Este proyecto se centra en la **ingeniería de datos agrícolas**. Tus programas deben demostrar cómo construir pipelines de datos robustas que gestionen de forma adecuada escenarios reales en agricultura.

Capítulo IV

Instrucciones Generales

- Tus programas deben estar escritos en Python 3.10+
- Tu código debe respetar los estándares de linter flake8
- Cada ejercicio debe estar en su propio archivo
- Incluye docstrings sencillos para funciones y clases
- Enfócate en demostrar claramente conceptos básicos de manejo de errores
- Muestra tanto operaciones normales como escenarios de error
- Usa apropiadamente las excepciones incorporadas
- Mantén las soluciones simples y orientadas al aprendizaje
- Tus programas nunca deben cerrarse de forma inesperada

Nota de Ingeniería de Datos: Este proyecto enseña **diseño resiliente de pipelines de datos** para sistemas agrícolas. Tu código debe demostrar cómo construir sistemas de monitorización tolerantes a fallos que mantengan la integridad de los datos en condiciones reales.

Capítulo V

Ejercicio 0: Pipeline de Validación de Datos Agrícolas

	Ejercicio: 0
ft_first_exception	
Directorio de entrega: $ex0/$	
Archivos a entregar: `ft_first_exception.py`	
Funciones autorizadas: `try, except, int(), print()`	

Tu flujo de datos, o pipeline, de agricultura inteligente recibe lecturas de temperatura de los sensores del campo. A veces los sensores transmiten datos corruptos o se introducen valores no válidos desde aplicaciones móviles. Tu capa de validación de datos debe filtrar los datos erróneos antes de que corrompan tu analítica agrícola.

Escribe una función `check_temperature(temp_str)` que:

- Reciba una entrada de texto de la persona usuaria.
- Intente convertirla a un número.
- Verifique si la temperatura es razonable para las plantas (0 a 40 grados).
- Devuelva la temperatura si es válida.
- Gestione el caso en que la entrada no sea un número.
- Gestione el caso en que la temperatura sea demasiado alta o demasiado baja.

Crea una función `test_temperature_input()` que demuestre:

- Pruebas con entrada válida (“25”).
- Pruebas con entrada no válida (“abc”).
- Pruebas con valores extremos (“100”, “-50”).
- Cómo tu programa sigue funcionando a pesar de los errores.

Ejemplo:

```
$> python3 ft_first_exception.py
=== Garden Temperature Checker ===

Testing temperature: 25
Temperature 25°C is perfect for plants!

Testing temperature: abc
Error: 'abc' is not a valid number

Testing temperature: 100
Error: 100°C is too hot for plants (max 40°C)

Testing temperature: -50
Error: -50°C is too cold for plants (min 0°C)

All tests completed - program didn't crash!
```

¿Qué sucede cuando tu programa intenta convertir "abc" a un número?
¿Cómo puedes **capturar** este error y gestionarlo de forma adecuada?

Capítulo VI

Ejercicio 1: Diferentes Tipos de Problemas

	Ejercicio: 1
ft_different_errors	
Directorio de entrega: $ex1/$	
Archivos a entregar: `ft_different_errors.py`	
Funciones autorizadas: `try, except, ValueError, ZeroDivisionError, FileNotFoundError, KeyError, print()`	

Tu programa del huerto puede encontrarse con diferentes tipos de problemas. Python tiene distintos tipos de errores para situaciones diversas, y puedes capturarlos por separado o juntos.

Escribe una función `garden_operations()` que demuestre estos errores comunes:

- `ValueError` - cuando se proporciona un dato erróneo (como "abc" en lugar de un número).
- `ZeroDivisionError` - cuando intentas dividir entre cero.
- `FileNotFoundError` - cuando intentas abrir un archivo que no existe.
- `KeyError` - cuando buscas algo que no está en un diccionario.

Crea una función `test_error_types()` que:

- Muestre la aparición de cada tipo de error.
- Capture cada error y explique qué salió mal.
- Demuestre que tu programa continúa ejecutándose tras cada error.
- Muestre cómo capturar múltiples tipos de error con un único bloque `except`.

Ejemplo:

```
$> python3 ft_different_errors.py
=== Garden Error Types Demo ===

Testing ValueError...
Caught ValueError: invalid literal for int()

Testing ZeroDivisionError...
Caught ZeroDivisionError: division by zero

Testing FileNotFoundError...
Caught FileNotFoundError: No such file 'missing.txt'

Testing KeyError...
Caught KeyError: 'missing\_plant'

Testing multiple errors together...
Caught an error, but program continues!

All error types tested successfully!
```

¿Por qué Python tiene diferentes tipos de errores? ¿Cómo puedes capturar **múltiples tipos** de errores con un solo fragmento de código?

Capítulo VII

Ejercicio 2: Crear tus Propios Tipos de Error

	Ejercicio: 2
ft_custom_errors	
Directorio de entrega: *ex2/*	
Archivos a entregar: `ft_custom_errors.py`	
Funciones autorizadas: `class, Exception, try, except, raise, print()`	

A veces los errores ya existentes en Python no son lo suficientemente específicos para tu programa del huerto. Puedes crear tus propios tipos de error para que tu código sea más claro y útil.

Crea estas clases simples de excepciones personalizadas:

- `GardenError` - Error básico para problemas del huerto.
- `PlantError` - Para problemas con plantas (hereda de GardenError).
- `WaterError` - Para problemas de riego (hereda de GardenError).

Cada excepción personalizada debe:

- Ser una clase simple que herede de Exception (o GardenError).
- Tener un mensaje de error ilustrativo y útil.
- Ser fácil de capturar y gestionar.

Crea funciones que:

- Generen tus errores personalizados en distintas situaciones.
- Muestren cómo capturar tus tipos de error específicos.
- Demuestren que al capturar `GardenError` se capturan todos los errores relacionados con el huerto.

Ejemplo:

```
$> python3 ft_custom_errors.py
=== Custom Garden Errors Demo ===

Testing PlantError...
Caught PlantError: The tomato plant is wilting!

Testing WaterError...
Caught WaterError: Not enough water in the tank!

Testing catching all garden errors...
Caught a garden error: The tomato plant is wilting!
Caught a garden error: Not enough water in the tank!

All custom error types work correctly!
```

¿Cuándo deberías crear tus propios tipos de error en lugar de usar los ya presentes en Python? ¿Cómo ayuda la **herencia** a organizar diferentes tipos de errores?

Capítulo VIII

Ejercicio 3: Bloque Finally - Siempre Limpia

	Ejercicio: 3
ft_finally_block	
Directorio de entrega: $ex3/$	
Archivos a entregar: `ft_finally_block.py`	
Funciones autorizadas: `try, except, finally, print()`	

A veces, tu programa del huerto necesita limpiar tras su ejecución, incluso si ocurre algún error. El bloque `finally` es perfecto para esto: siempre se ejecuta, haya o no un error.

Escribe una función `water_plants(plant_list)` que:

- Abra un "sistema de riego" (simplemente imprime un mensaje).
- Recorra cada planta de la lista.
- Riegue cada planta (imprime un mensaje).
- Siempre cierre el sistema de riego en un bloque `finally`.
- Gestione errores si un nombre de planta no es válido.

Crea una función `test_watering_system()` que demuestre:

- Riego normal con una lista de plantas válida.
- Riego con una lista de plantas no válida (provoca un error).
- Que la limpieza siempre ocurre, incluso cuando hay un error.
- Uso de la estructura try/except/finally.

Ejemplo:

```
$> python3 ft_finally_block.py
=== Garden Watering System ===

Testing normal watering...
Opening watering system
Watering tomato
Watering lettuce
Watering carrots
Closing watering system (cleanup)
Watering completed successfully!

Testing with error...
Opening watering system
Watering tomato
Error: Cannot water None - invalid plant!
Closing watering system (cleanup)

Cleanup always happens, even with errors!
```

¿Por qué es importante limpiar los recursos incluso cuando ocurren errores? ¿Cómo ayuda el bloque **finally** a asegurar que la limpieza siempre ocurra?

Capítulo IX

Ejercicio 4: Lanzar tus Propios Errores

	Ejercicio: 4
ft_raise_errors	
Directorio de entrega: *ex*4/	
Archivos a entregar: `ft_raise_errors.py`	
Funciones autorizadas: `try, except, raise, ValueError, print()`	

A veces tu programa del huerto necesita crear sus propios errores cuando detecta un problema. Puedes usar la palabra clave `raise` para indicar que algo va mal.

Escribe una función `check_plant_health(plant_name, water_level, sunlight_hours)` que:

- Verifique que el nombre de la planta sea válido (no vacío).
- Verifique que el nivel de agua sea razonable (entre 1 y 10).
- Verifique que las horas de luz sean razonables (entre 2 y 12).
- Lance errores apropiados con mensajes útiles cuando algo esté mal.
- Devuelva un mensaje de éxito si todo está bien.

Crea una función `test_plant_checks()` que demuestre:

- Pruebas con valores correctos (debería funcionar bien).
- Pruebas con nombre de planta incorrecto (debería lanzar ValueError).
- Pruebas con nivel de agua incorrecto (debería lanzar ValueError).
- Pruebas con horas de luz incorrectas (debería lanzar ValueError).
- Captura y gestión adecuada de cada error.

Ejemplo:

```
$> python3 ft_raise_errors.py
=== Garden Plant Health Checker ===

Testing good values...
Plant 'tomato' is healthy!

Testing empty plant name...
Error: Plant name cannot be empty!

Testing bad water level...
Error: Water level 15 is too high (max 10)

Testing bad sunlight hours...
Error: Sunlight hours 0 is too low (min 2)

All error raising tests completed!
```

¿Cuándo debería tu programa **lanzar** sus propios errores? ¿Cómo crearías mensajes de error útiles que indiquen exactamente qué salió mal?

Capítulo X

Ejercicio 5: Sistema de Gestión del Huerto

	Ejercicio: 5
ft_garden_management	
Directorio de entrega: $ex5/$	
Archivos a entregar: `ft_garden_management.py`	
Funciones autorizadas: `class, Exception, try, except, finally, raise, print()`	

¡Es momento de juntarlo todo!
Crea un sistema sencillo de gestión del huerto que use todas las técnicas de manejo de errores que has aprendido.

Crea una clase `GardenManager` que:

- Tenga métodos para añadir plantas, regarlas y comprobar su salud.
- Utilice tus tipos de error personalizados de los ejercicios anteriores.
- Gestione adecuadamente diferentes tipos de errores.
- Use bloques try/except/finally donde sea necesario.
- Lance sus propios errores cuando algo esté mal.
- Siga funcionando incluso cuando algunas operaciones fallen.

Tu gestor del huerto debe:

- Manejar correctamente entradas de datos incorrectas.
- Usar excepciones personalizadas para problemas específicos del huerto.
- Limpiar siempre los recursos (usa bloques `finally`).
- Proporcionar mensajes de error útiles a quien usa el sistema.
- Demostrar todos los conceptos de manejo de errores de este proyecto.

Ejemplo:

```
$> python3 ft_garden_management.py
=== Garden Management System ===

Adding plants to garden...
Added tomato successfully
Added lettuce successfully
Error adding plant: Plant name cannot be empty!

Watering plants...
Opening watering system
Watering tomato - success
Watering lettuce - success
Closing watering system (cleanup)

Checking plant health...
tomato: healthy (water: 5, sun: 8)
Error checking lettuce: Water level 15 is too high (max 10)

Testing error recovery...
Caught GardenError: Not enough water in tank
System recovered and continuing...

Garden management system test complete!
```

Este ejercicio combina todos los conceptos de manejo de errores de este proyecto. Se evaluará conjuntamente cómo usas los bloques try/except, las excepciones personalizadas, los bloques finally y el lanzamiento de errores.

¿Cómo funcionan juntas estas técnicas de manejo de errores para crear un programa de huerto robusto? ¿Qué hace que un programa sea **fiable** cuando las cosas salen mal?

Capítulo XI

Entrega y Evaluación

Entrega tu trabajo en tu repositorio `Git` como de costumbre. Solo se evaluará el contenido que esté dentro de tu repositorio durante la defensa. No dudes en comprobar que los nombres de tus archivos sean los correctos.

Durante la evaluación, se te puede pedir que expliques tu código, sigas el flujo ejecución o modifiques tus soluciones. Asegúrate de comprender cada línea que escribes.

Debes entregar únicamente los archivos solicitados por el enunciado de este proyecto. Concéntrate en escribir un código limpio, legible y que demuestre tu comprensión de los fundamentos de Python.