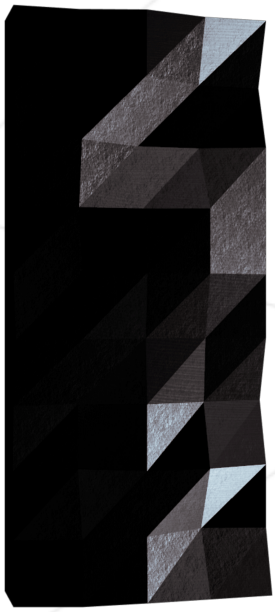

Garden Guardian

Data Engineering for Smart Agriculture

Summary: Build resilient data pipelines for your smart garden! Learn to handle sensor failures, process agricultural data streams, and create robust monitoring systems that keep your digital greenhouse thriving.

Version: 2.1

Contents

Chapter I

Foreword

Welcome to the world of agricultural data engineering!

Building on your Python foundations (Module 00) and garden monitoring classes (Module 01), you're now ready to tackle the real challenges of smart agriculture. In modern farming, data flows like water through irrigation systems—sensor readings stream in continuously, weather APIs provide forecasts, and IoT devices monitor everything from soil pH to greenhouse humidity.

But what happens when this data pipeline encounters turbulence? When sensors malfunction during harvest season? When network connections drop during critical monitoring periods? When corrupted data threatens to trigger false irrigation cycles?

Professional agricultural data engineers know that robust systems aren't built to avoid failures—they're designed to **gracefully handle the unexpected**. Your digital greenhouse needs to be as resilient as nature itself.

Python's **exception handling** system is your toolkit for building bulletproof agricultural data pipelines. You'll learn to **catch sensor anomalies**, **create custom agricultural alerts**, and **ensure data integrity** even when Mother Nature (or Murphy's Law) strikes.

In this project, you'll evolve from a basic programmer to an agricultural data engineer, building monitoring systems that keep growing even when the unexpected happens.

Chapter II

AI Instructions

Context

During your learning journey, AI can assist with many different tasks. Take the time to explore the various capabilities of AI tools and how they can support your work. However, always approach them with caution and critically assess the results. Whether it's code, documentation, ideas, or technical explanations, you can never be completely sure that your question was well-formed or that the generated content is accurate. Your peers are a valuable resource to help you avoid mistakes and blind spots.

Main message

- Use AI to reduce repetitive or tedious tasks.
- Develop prompting skills — both coding and non-coding — that will benefit your future career.
- Learn how AI systems work to better anticipate and avoid common risks, biases, and ethical issues.
- Continue building both technical and power skills by working with your peers.
- Only use AI-generated content that you fully understand and can take responsibility for.

Learner rules:

- You should take the time to explore AI tools and understand how they work, so you can use them ethically and reduce potential biases.
- You should reflect on your problem before prompting — this helps you write clearer, more detailed, and more relevant prompts using accurate vocabulary.
- You should develop the habit of systematically checking, reviewing, questioning, and testing anything generated by AI.
- You should always seek peer review — don't rely solely on your own validation.

Phase outcomes:

- Develop both general-purpose and domain-specific prompting skills.
- Boost your productivity with effective use of AI tools.
- Continue strengthening computational thinking, problem-solving, adaptability, and collaboration.

Comments and examples:

- You'll regularly encounter situations — exams, evaluations, and more — where you must demonstrate real understanding. Be prepared, keep building both your technical and interpersonal skills.
- Explaining your reasoning and debating with peers often reveals gaps in your understanding. Make peer learning a priority.
- AI tools often lack your specific context and tend to provide generic responses. Your peers, who share your environment, can offer more relevant and accurate insights.
- Where AI tends to generate the most likely answer, your peers can provide alternative perspectives and valuable nuance. Rely on them as a quality checkpoint.

✓ Good practice:

> I ask AI: "How do I test a sorting function?" It gives me a few ideas. I try them out and review the results with a peer. We refine the approach together.

✗ Bad practice:

> I ask AI to write a whole function, copy-paste it into my project. During peer-evaluation, I can't explain what it does or why. I lose credibility — and I fail my project.

✓ Good practice:

> I use AI to help design a parser. Then I walk through the logic with a peer. We catch two bugs and rewrite it together — better, cleaner, and fully understood.

✗ Bad practice:

> I let Copilot generate my code for a key part of my project. It compiles, but I can't explain how it handles pipes. During the evaluation, I fail to justify and I fail my project.

Chapter III

Introduction

Welcome to Garden Guardian: Data Engineering for Smart Agriculture!

Building on your garden monitoring foundation from previous projects, you'll now master the critical skills of **resilient data pipeline engineering** for agricultural systems.

You'll discover:

- How to **validate and clean agricultural data streams** in real-time
- What different **failure modes** exist in IoT sensor networks
- How to **create custom agricultural alerts** for crop-specific monitoring
- Essential techniques for **data pipeline fault tolerance** and recovery
- How to **ensure data integrity** in distributed farming systems

Each exercise builds a component of your smart agriculture data platform, progressing from basic sensor validation to comprehensive agricultural monitoring systems.

IMPORTANT: This project focuses on **agricultural data engineering**. Your programs should demonstrate how to build robust data pipelines that handle real-world farming scenarios gracefully.

Chapter IV

General Instructions

- Your programs must be written in Python 3.10+
- Your code must respect the flake8 linter standards
- All functions and methods must include type hints
- Each exercise must be in its own file
- Focus on demonstrating basic error handling concepts clearly
- Show both normal operations and error scenarios
- Use built-in exceptions appropriately
- Keep solutions simple and focused on learning
- Your programs must never crash

Data Engineering Note: This project teaches **resilient data pipeline design** for agricultural systems. Your code should demonstrate how to build fault-tolerant monitoring systems that maintain data integrity under real-world conditions.

Exception Handling: All exercises in this module require the use of try/except blocks for error handling. Python keywords such as 'try', 'except', 'finally', and 'raise' are fundamental language features and do not need to be listed in authorized functions.

You may use any built-in exception types necessary to complete the exercises, including but not limited to: ValueError, TypeError, ZeroDivisionError, FileNotFoundError, KeyError, IndexError, AttributeError, and the base Exception class. Each exercise description will indicate which exception types are most appropriate for that specific task.

Chapter V

Exercise 0: Agricultural Data Validation Pipeline

	Exercise0
ft_first_exception	
Directory: *ex*0/	
Files to Submit: `ft_first_exception.py`	
Authorized: `int(), print()`	

Your smart agriculture data pipeline receives temperature readings from field sensors. Sometimes sensors transmit corrupted data or farmers input invalid values through mobile apps. Your data validation layer must filter out bad data before it corrupts your agricultural analytics.

Write a function `check_temperature(temp_str)` that:

- Takes a string input from the user
- Tries to convert it to a number
- Checks if the temperature is reasonable for plants (0 to 40 degrees)
- Returns the temperature if it's valid
- Handles the case when the input isn't a number
- Handles the case when the temperature is too high or too low

Create a `test_temperature_input()` function that demonstrates:

- Testing with good input ("25")
- Testing with bad input ("abc")
- Testing with extreme values ("100", "-50")
- Showing how your program keeps running despite errors

Example:

```
$> 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!
```

What happens when your program tries to convert "abc" to a number?
How can you **catch** this error and handle it gracefully?

Chapter VI

Exercise 1: Different Types of Problems

	Exercise1
ft_different_errors	
Directory: *ex*1/	
Files to Submit: `ft_different_errors.py`	
Authorized: `print(), open(), close(), int()`	

Your garden program might encounter different types of problems. Python has different types of errors for different situations, and you can catch them separately or together.

Write a function `garden_operations()` that demonstrates these common errors:

- `ValueError` - when someone gives bad data (like "abc" instead of a number)
- `ZeroDivisionError` - when you try to divide by zero
- `FileNotFoundError` - when you try to open a file that doesn't exist
- `KeyError` - when you look for something that isn't in a dictionary

Create a `test_error_types()` function that:

- Shows each type of error happening
- Catches each error and explains what went wrong
- Demonstrates that your program continues running after each error
- Shows how to catch multiple error types with one except block

Example:

```
$> 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!
```

Why does Python have different types of errors? How can you catch **multiple types** of errors with one piece of code?

Chapter VII

Exercise 2: Making Your Own Error Types

	Exercise2
ft_custom_errors	
Directory: *ex*2/	
Files to Submit: `ft_custom_errors.py`	
Authorized: `print(), int(), input()`	

Sometimes the built-in Python errors aren't specific enough for your garden program. You can create your own error types to make your code clearer and more helpful.

Create these simple custom exception classes:

- `GardenError` - A basic error for garden problems
- `PlantError` - For problems with plants (inherits from GardenError)
- `WaterError` - For problems with watering (inherits from GardenError)

Each custom exception should:

- Be a simple class that inherits from Exception (or GardenError)
- Have a helpful error message
- Be easy to catch and handle

Create functions that:

- Raise your custom errors in different situations
- Show how to catch your specific error types
- Demonstrate that catching `GardenError` catches all garden-related errors

Example:

```
$> 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!
```

When should you create your own error types instead of using Python's built-in ones? How does **inheritance** help organize different types of errors?

Chapter VIII

Exercise 3: Finally Block - Always Clean Up

	Exercise3
ft_finally_block	
Directory: *ex*3/	
Files to Submit: `ft_finally_block.py`	
Authorized: `print(), int()`	

Sometimes your garden program needs to clean up after itself, even if an error happens. The `finally` block is perfect for this - it always runs, whether there was an error or not.

Write a function `water_plants(plant_list)` that:

- Opens a "watering system" (just print a message)
- Goes through each plant in the list
- Waters each plant (print a message)
- Always closes the watering system in a `finally` block
- Handles errors if a plant name is invalid

Create a `test_watering_system()` function that demonstrates:

- Normal watering with a good plant list
- Watering with a bad plant list (causes an error)
- Shows that cleanup always happens, even when there's an error
- Uses try/except/finally structure

Example:

```
$> 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!
```

Why is it important to clean up resources even when errors happen?
How does the **finally** block help ensure cleanup always occurs?

Chapter IX

Exercise 4: Raising Your Own Errors

	Exercise4
ft_raise_errors	
Directory: *ex*4/	
Files to Submit: `ft_raise_errors.py`	
Authorized: `print(), int()`	

Sometimes your garden program needs to create its own errors when it detects a problem. You can use the `raise` keyword to signal that something is wrong.

Write a function `check_plant_health(plant_name, water_level, sunlight_hours)` that:

- Checks if the plant name is valid (not empty)
- Checks if water level is reasonable (between 1 and 10)
- Checks if sunlight hours are reasonable (between 2 and 12)
- Raises appropriate errors with helpful messages when something is wrong
- Returns a success message if everything is okay

Create a `test_plant_checks()` function that demonstrates:

- Testing with good values (should work fine)
- Testing with bad plant name (should raise ValueError)
- Testing with bad water level (should raise ValueError)
- Testing with bad sunlight hours (should raise ValueError)
- Catching and handling each error appropriately

Example:

```
$> 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!
```

When should your program **raise** its own errors? How do you create helpful error messages that tell users exactly what went wrong?

Chapter X

Exercise 5: Garden Management System

	Exercise5
ft_garden_management	
Directory: *ex*5/	
Files to Submit: `ft_garden_management.py`	
Authorized: `print(), int()`	

Now put everything together!
Create a simple garden management system that uses all the error handling techniques you've learned.

Create a `GardenManager` class that:

- Has methods to add plants, water plants, and check plant health
- Uses your custom error types from previous exercises
- Handles different types of errors appropriately
- Uses try/except/finally blocks where needed
- Raises its own errors when something is wrong
- Keeps working even when some operations fail

Your garden manager should:

- Handle bad input gracefully
- Use custom exceptions for garden-specific problems
- Always clean up resources (use finally blocks)
- Provide helpful error messages to users
- Demonstrate all the error handling concepts from this project

Create a `test_garden_management()` function that demonstrates:

- Adding plants with both valid and invalid inputs
- Watering plants with proper cleanup (using finally)
- Checking plant health and handling validation errors
- Error recovery - showing the system continues working after errors
- Integration of all error handling techniques learned

Example:

```
$> 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!
```

This exercise combines all the error handling concepts from the project. You'll be evaluated on how well you use try/except blocks, custom exceptions, finally blocks, and error raising together.

How do all these error handling techniques work together to make a robust garden program? What makes a program **reliable** when things go wrong?

Chapter XI

Turn in and Submission

Turn in your assignment in your `Git` repository as usual. Only the work inside your repository will be evaluated during the defense. Don't hesitate to double check the names of your files to ensure they are correct.

During evaluation, you may be asked to explain error handling concepts, demonstrate how exceptions work in your garden programs, or show how your system handles different types of problems. Make sure you understand the principles behind your code.

You need to return only the files requested by the subject of this project. Focus on clean, readable code that clearly demonstrates error handling and defensive programming concepts.