

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: 3.0

Contents

I	Foreword	2
II	AI Instructions	3
III	Introduction	5
IV	General Instructions	6
V	Exercise 0: Agricultural Data Validation	7
VI	Exercise 1: Agricultural Data Validation Pipeline	9
VII	Exercise 2: Different Types of Problems	11
VIII	Exercise 3: Making Your Own Error Types	14
IX	Exercise 4: Finally Block - Always Clean Up	16
X	Turn in and Submission	18

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
- Which 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: use `mypy` to check your code
- 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 may include which exception types are most appropriate for that specific task.

Chapter V

Exercise 0: Agricultural Data Validation

	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 simple function `input_temperature(temp_str)` that:

- Takes an input string as a parameter
- Converts it to a number
- Returns the temperature as an integer

Then, write a function `test_temperature()` that will perform the following tests on `input_temperature()`:

- Use a valid input ("25")
- Use an invalid input ("abc")
- Handle the case when `input_temperature()` fails; print an error message
- Show that your program keeps running despite the error

Example:

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

Input data is '25'
Temperature is now 25°C

Input data is 'abc'
Caught input_temperature error: invalid literal for int() with base 10: 'abc'

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



You can use the base Exception class, or find out which exceptions can be raised by input_temperature().



It's up to you to properly adjust the type hints for these two functions. Consider this information valid for all the exercises and upcoming projects.

Chapter VI

Exercise 1: Agricultural Data Validation Pipeline

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

You actually need extra control over your data. Even if the temperature is a valid number, your plants cannot grow if it's too cold or too hot.

Use your code from Exercise 0 and improve `input_temperature()` and `test_temperature()` as follows:

- In `input_temperature`:
  - Check if the temperature is reasonable for plants (0 to 40 degrees Celsius, limits included)
  - Return the temperature if it's valid; otherwise, *raise* an exception.
- In `test_temperature`:
  - Add new tests with extreme values ("100", "-50")

The `main` part of your file will call `test_temperature()`.

Example:

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

Input data is '25'
Temperature is now 25°C

Input data is 'abc'
Caught input_temperature error: invalid literal for int() with base 10: 'abc'

Input data is '100'
Caught input_temperature error: 100°C is too hot for plants (max 40°C)

Input data is '-50'
Caught input_temperature error: -50°C is too cold for plants (min 0°C)

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

Chapter VII

Exercise 2: Different Types of Problems

	Exercise2
ft_different_errors	
Directory: *ex2/*	
Files to Submit: `ft_different_errors.py`	
Authorized: `print(), open(), 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(operation_number)` that contains faulty code. For each value of `operation_number` between 0 and 3, a different piece of faulty code will raise one of the following exception:

- `ValueError` - when bad data is provided (like "abc" instead of a number to `int()`)
- `ZeroDivisionError` - when you try to divide by zero
- `FileNotFoundError` - when you try to open a file that does not exist (and if it's not open, no need to close() it)
- `TypeError` - when you try to mix different types that cannot be mixed (did you try to add a string and a number?)

Other values of `operation_number` will not contain faulty code and simply return.

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 `try:` block

Example:

```
$> python3 ft_different_errors.py
=== Garden Error Types Demo ===
Testing operation 0...
Caught ValueError: invalid literal for int() with base 10: 'abc'
Testing operation 1...
Caught ZeroDivisionError: division by zero
Testing operation 2...
Caught FileNotFoundError: [Errno 2] No such file or directory: '/non/existent/file'
Testing operation 3...
Caught TypeError: can only concatenate str (not "int") to str
Testing operation 4...
Operation completed successfully

All error types tested successfully!
```



Why does Python have different types of errors? How can you catch **multiple types** of errors with a single try: only? Note that you can't use type().



mypy will display an error for the faulty code that raises the TypeError. That's its job! So, to test this exception, we need to keep this error on purpose.



You have already encountered open() in C. Using it in Python is pretty straightforward.

Chapter VIII

Exercise 3: Making Your Own Error Types

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

Sometimes the built-in Python errors are not 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 specific default error message (e.g., “Unknown plant error”) if none is provided

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 GardenError: The tomato plant is wilting!
Caught GardenError: 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 IX

Exercise 4: Finally Block - Always Clean Up

	Exercise4
ft_finally_block	
Directory: *ex*4/	
Files to Submit: `ft_finally_block.py`	
Authorized: `print(), str.capitalize()`	

Your garden program needs to clean up resources, even if an error has occurred. The `finally` block is perfect for this - it always runs, whether there was an error or not.

Write a function `water_plant(plant_name)` that:

- Tries to water the plant
- Succeeds if the plant name is capitalized
- Raises an error if the plant name is not capitalized (use your PlantError exception from the previous exercise)
- Prints a message on success

Create a `test_watering_system()` function that:

- Opens the watering system (just print a message)
- Waters several plants using `water_plant()`
- Uses try/except/finally structure
- Handles errors if a plant name is invalid and can't be watered. In that case, stop the test and immediately return to `main`.
- Always closes the watering system in a `finally` block
- Shows that cleanup always happens, even when there's an error

Example:

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

Testing valid plants...
Opening watering system
Watering Tomato: [OK]
Watering Lettuce: [OK]
Watering Carrots: [OK]
Closing watering system

Testing invalid plants...
Opening watering system
Watering Tomato: [OK]
Caught PlantError: Invalid plant name to water: 'lettuce'
.. ending tests and returning to main
Closing watering system

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 X

Turn in and Submission

Turn in your assignment to your `Git` repository as usual. Only the work inside your repository will be evaluated during the defense. Do not 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 submit only the files requested by the subject of this project. Focus on clean, readable code that clearly demonstrates error handling and defensive programming concepts.