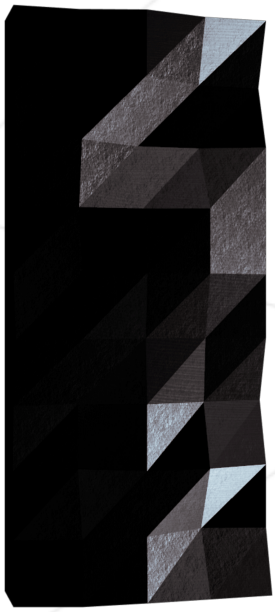

CodeCultivation

Object-Oriented Garden Systems

Summary: Build a comprehensive digital garden ecosystem while discovering advanced Python concepts. Create tools to manage community gardens efficiently through data-driven approaches.

Version: 1.0

Contents

Chapter I

Foreword

In the digital age, even gardens need smart systems.

A wise gardener once said: "You can't plant flowers and then pull them up every day to see how the roots are doing." The same applies to code—sometimes the most beautiful growth happens when we trust the process and let our programs develop naturally, layer by layer.

Behind every thriving community garden lies a network of relationships. Plants don't grow in isolation; they form communities where each species contributes something unique while sharing common needs. Some plants protect others from pests, some fix nitrogen in the soil, and others provide shade for delicate seedlings. This interconnected web of mutual support mirrors how well-designed software systems work—individual components with distinct roles, working together harmoniously.

Just as a master gardener knows that healthy soil produces healthy plants, experienced programmers understand that well-structured code grows into robust applications. You'll discover that organizing your code is like planning a garden: some elements need protection, others need room to grow, and the best systems emerge when everything has its proper place and purpose.

In this project, you'll cultivate your programming skills while building tools that could genuinely help community gardens flourish. Every line of code you write is a seed planted in the digital soil of possibility.

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 Code Cultivation!

Building on your Python fundamentals from the first activity, you'll now tackle more complex programming challenges through creating a comprehensive garden data management system. This project introduces advanced concepts that make Python a powerful tool for modeling real-world systems.

You'll work on:

- Understanding how Python programs are structured and executed
- Organizing complex data structures efficiently
- Creating reusable code components
- Building systems that can adapt and extend
- Protecting data integrity in collaborative environments
- Designing scalable software architectures

Each exercise builds upon previous ones, creating a complete digital garden ecosystem by the end.

IMPORTANT: This module starts with basic Python program structure, then progresses to Object-Oriented Programming. Each exercise should contain the requested definitions and any required code. You may include simple test code at the bottom of each file using `if __name__ == "__main__":` blocks for your own testing.

Chapter IV

General Instructions

- Your code must be written in Python 3.10+
- Your code must respect the flake8 linter standards
- Each exercise must be in its own file
- Use proper naming conventions: classes in PascalCase, functions and variables in snake_case
- Include docstrings for functions, classes and methods
- Type hints are encouraged for all functions and methods
- You don't need to handle input validation unless explicitly mentioned
- Focus on demonstrating programming concepts clearly
- Your programs must always run without errors

Digital Garden Ecosystem Note: This project focuses on Python programming concepts, starting from basic program structure and progressing to Object-Oriented Programming. Each exercise introduces new features while building a cohesive garden management system. Later exercises will reuse concepts from earlier ones.

Chapter V

Exercise 0: Planting Your First Seed

	Exercise0
ft_garden_intro	
Directory: *ex*0/	
Files to Submit: `ft_garden_intro.py`	
Authorized: `print(), if __name__ == "__main__"`	

Before we can build complex garden management systems, you need to understand how Python programs work.
Every Python program needs a starting point - a place where execution begins. Just like planting your first seed in a garden, this is where your programming journey begins!

Your task is to create your very first Python program that displays information about a plant in your garden.

Requirements:

- Create a program that runs when executed directly
- Use the special `if __name__ == "__main__":` pattern
- Store plant information in simple variables (name, height, age)
- Display the plant information using `print()`

Example:

```
$> python3 ft_garden_intro.py
=== Welcome to My Garden ===
Plant: Rose
Height: 25cm
Age: 30 days

=== End of Program ===
```

This is your first seed planted in the garden of Python programming! Understanding how programs start and execute is fundamental before moving to more complex concepts like functions and classes.

What happens if you remove the if __name__ == "__main__": line? Try it and observe the difference! Also, have you noticed that some Python files start with a special line beginning with #!? Research what this "shebang" line does and why it might be useful for making your scripts executable directly.

Chapter VI

Exercise 1: Garden Data Organizer

	Exercise1
ft_garden_data	
Directory: *ex*1/	
Files to Submit: `ft_garden_data.py`	
Authorized: `class, __init__, print()`	

The community garden needs to track multiple plants with their information. You need to store and display data for several plants efficiently.

Each plant has:

- A name
- Height in centimeters
- Age in days

Create a program that manages data for at least 3 different plants and displays their information in an organized way.

You must create a Plant class that serves as a blueprint for any plant, rather than handling each one individually. For example, every plant might have a name, height, and age - but you'll need a way to organize this data that makes sense
.
Required: Create a `Plant` blueprint that can represent any plant with its attributes.

Example:

```
$> python3 ft_garden_data.py
=== Garden Plant Registry ===
Rose: 25cm, 30 days old
Sunflower: 80cm, 45 days old
Cactus: 15cm, 120 days old
```

How are you currently storing plant information? What challenges might arise with more plants?

Chapter VII

Exercise 2: Plant Growth Simulator

	Exercise2
ft_plant_growth	
Directory: *ex2/*	
Files to Submit: `ft_plant_growth.py`	
Authorized: `class`, `__init__`, `print()`	

The garden manager wants to simulate plant growth over time. You need to track how plants change and provide *operations* to modify their state.

Requirements:

- Reuse your `Plant class` from Exercise 1
- Plants need to be able to `grow()` and `age()` - think about what actions a plant can perform
- You'll need a way to `get_info()` about the current plant status
- Simulate a week of growth for multiple plants
- Consider how plants should change over time through their own actions

Your program should show plants changing over time. Think about giving your plants *behaviors* they can perform on themselves.

Example:

```
$> python3 ft_plant_growth.py
=== Day 1 ===
Rose: 25cm, 30 days old
=== Day 7 ===
Rose: 31cm, 36 days old
Growth this week: +6cm
```

How are you handling the operations on plant data? Is there repetition in your code?

Chapter VIII

Exercise 3: Plant Factory

	Exercise3
ft_plant_factory	
Directory: *ex3/*	
Files to Submit: `ft_plant_factory.py`	
Authorized: `class, __init__, print()`	

The garden center needs to create many plants quickly using your `Plant class` with different starting values. You need to streamline the plant *creation* process and *initialize* them properly.

Requirements:

- Plants need to be created with their initial information (name, starting height, starting age)
- Each plant should be ready to use immediately after *construction*
- Create at least 5 different plants with varying characteristics
- Display all created plants in an organized format
- Think about how you can streamline the plant creation process

Consider how you might set up plants with their starting values efficiently. What would make creating many plants easier?

Example:

```
$> python3 ft_plant_factory.py
=== Plant Factory Output ===
Created: Rose (25cm, 30 days)
Created: Oak (200cm, 365 days)
Created: Cactus (5cm, 90 days)
Created: Sunflower (80cm, 45 days)
Created: Fern (15cm, 120 days)

Total plants created: 5
```

How are you currently initializing your plants? Is there a more efficient way to set them up?

Chapter IX

Exercise 4: Garden Security System

	Exercise4
ft_garden_security	
Directory: *ex4/*	
Files to Submit: `ft_garden_security.py`	
Authorized: `class, __init__, def, print(), setter/getter (custom)`	

The garden manager is concerned about data integrity. Some volunteers accidentally corrupted plant data by setting invalid values (negative heights, impossible ages). You need to create a secure system that *protects* and *encapsulates* sensitive data.

Requirements:

- Create a `SecurePlant` that protects its data from corruption
- Provide controlled ways to modify plant data: `set_height()`, `set_age()`
- Provide safe ways to access plant data: `get_height()`, `get_age()`
- Ensure plant height cannot be negative through validation
- Ensure plant age cannot be negative through validation
- Print error messages when invalid values are attempted
- Think about *encapsulation* - protecting important data from direct access

Consider how you might create a system that *validates* data before storing it, ensuring data integrity.

Example:

```
$> python3 ft_garden_security.py
=== Garden Security System ===
Plant created: Rose
Height updated: 25cm [OK]
Age updated: 30 days [OK]

Invalid operation attempted: height -5cm [REJECTED]
Security: Negative height rejected

Current plant: Rose (25cm, 30 days)
```

How can you protect your data from being accidentally corrupted?
What mechanisms could you put in place?

Chapter X

Exercise 5: Specialized Plant Types

	Exercise5
ft_plant_types	
Directory: *ex5/*	
Files to Submit: `ft_plant_types.py`	
Authorized: `class, __init__, super(), print()`	

The garden now needs to handle different types of plants: flowers, trees, and vegetables. Each type has unique characteristics but *inherits* common plant features from their *parent* category.

Requirements:

- Start with a base `Plant` that has common features (name, height, age)
- Create specialized types: `Flower`, `Tree`, and `Vegetable`
- Each specialized type should *inherit* the basic plant features
- `Flower` needs: color attribute and ability to `bloom()`
- `Tree` needs: trunk_diameter and ability to `produce_shade()`
- `Vegetable` needs: harvest_season and nutritional_value
- When creating specialized plants, call the parent setup with `super().__init__()`
- Create at least 2 instances of each plant type
- Avoid duplicating common plant code across different specialized types

Think about *family relationships* in nature - how do species share traits while having unique characteristics?

Example:

```
$> python3 ft_plant_types.py
=== Garden Plant Types ===

Rose (Flower): 25cm, 30 days, red color
Rose is blooming beautifully!

Oak (Tree): 500cm, 1825 days, 50cm diameter
Oak provides 78 square meters of shade

Tomato (Vegetable): 80cm, 90 days, summer harvest
Tomato is rich in vitamin C
```

Notice how much code you might be duplicating. Each plant type shares common features but has unique characteristics. Think about **family relationships** in nature-how do species share traits?

How are you handling the common features shared by all plant types? Is there a way to avoid repeating the same code by creating a family tree of related types?

Chapter XI

Exercise 6: Garden Analytics Platform

	Exercise6
ft_garden_analytics	
Directory: *ex6/*	
Files to Submit: `ft_garden_analytics.py`	
Authorized: `class`, `__init__`, `super()`, `print()`, `staticmethod()`, `classmethod()`	

Build a comprehensive garden data analytics platform that processes and analyzes garden data. This system needs to handle complex data relationships and provide detailed analytics using *nested* components and *inheritance chains*.

Requirements:

- Create a `GardenManager` that can handle multiple gardens
- Include a helper `GardenStats` inside your manager for calculating statistics
- Build a plant family tree: `Plant` → `FloweringPlant` → `PrizeFlower`
- Include a method `create_garden_network()` that works on the manager type itself
- Add utility functions that don't need specific garden data
- Show different types of methods: instance methods, class-level methods, and utility functions
- Each garden should track plant collections and statistics
- Use your nested statistics helper to calculate analytics
- Organize everything within appropriate structures - avoid scattered global functions

Consider how you might organize complex systems with multiple interacting components. What happens when you need different types of methods?

Example:

```
$> python3 ft_garden_analytics.py
=== Garden Management System Demo ===

Added Oak Tree to Alice's garden
Added Rose to Alice's garden
Added Sunflower to Alice's garden

Alice is helping all plants grow...
Oak Tree grew 1cm
Rose grew 1cm
Sunflower grew 1cm

=== Alice's Garden Report ===
Plants in garden:
- Oak Tree: 101cm
- Rose: 26cm, red flowers (blooming)
- Sunflower: 51cm, yellow flowers (blooming), Prize points: 10

Plants added: 3, Total growth: 3cm
Plant types: 1 regular, 1 flowering, 1 prize flowers

Height validation test: True
Garden scores - Alice: 218, Bob: 92
Total gardens managed: 2
```

This exercise brings together all the programming patterns you've learned throughout the project. You will be evaluated on your understanding of how different components interact and organize themselves within a complex system.

How do you organize complex systems with multiple interacting components? What happens when you need different types of methods that belong to the class itself rather than individual instances?

Chapter XII

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 programming concepts, demonstrate inheritance relationships, or extend your code with new functionality. Make sure you understand the principles behind your implementations.

You need to return only the files requested by the subject of this project. Focus on clean, well-documented code that clearly demonstrates programming principles.