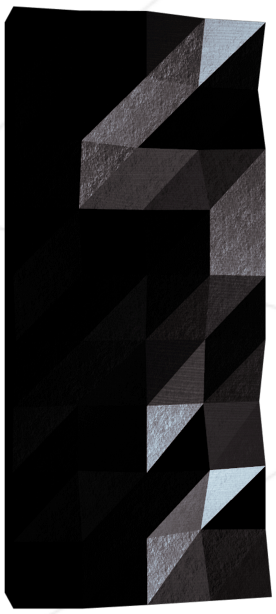

Growing Code

Python Fundamentals Through Garden Data

Summary: Discover Python's fundamental syntax and semantics through analyzing community garden data. Learn expressions, variables, functions, and control flow while contributing to sustainable community initiatives.

Version: 1.0

Contents

I	Foreword	2
II	AI Instructions	3
III	Introduction	5
IV	General Instructions	6
V	Exercise 0: Hello Garden	7
VI	Exercise 1: Garden Plot Area	8
VII	Exercise 2: Harvest Total	9
VIII	Exercise 3: Plant Age Check	10
IX	Exercise 4: Water Reminder	11
X	Exercise 5: Count to Harvest	12
XI	Exercise 6: Garden Summary	14
XII	Exercise 7: Seed Inventory with Type Annotations	15
XIII	Helper Resources	16
XIV	Turn in and Submission	17

Chapter I

Foreword

In community gardens across the world, data tells stories of growth, collaboration, and impact.

From tracking harvest yields that feed local families to monitoring water usage that preserves precious resources, every measurement matters. Each data point represents someone's dedication—a volunteer's weekend hours, a child's first tomato, an elderly neighbor sharing decades of gardening wisdom.

Python, like these gardens, grows from simple seeds into something powerful and nourishing. Named after Monty Python's Flying Circus, it reminds us that learning should be joyful, accessible, and inclusive. Today, Python helps scientists understand climate change, enables communities to optimize resource sharing, and empowers anyone to transform raw data into meaningful insights.

In this project, you'll discover Python's fundamental building blocks while analyzing real community garden data. You'll learn that programming, like gardening, is about nurturing growth—both in code and in the communities we serve.

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 Growing Code!

In this project, you'll discover Python's core concepts through community garden scenarios. You'll work with practical exercises that introduce fundamental programming concepts in an engaging, real-world context.

Each exercise builds upon the previous one, helping you develop essential programming skills while working on meaningful tasks.

`IMPORTANT: For each exercise, you must write ONLY a function (not a main program). Each file should contain only the requested function that handles input and output directly. Do not write if __name__ == "__main__": blocks or call the function directly in your files.`

`HELPER TOOL: To help you test your exercises, there is a main.py file provided in the attachments. Copy this file to your working directory and run python3 main.py to test your functions easily. This helper will import and test your functions automatically.`

Chapter IV

General Instructions

- Your functions must be written in Python 3.11+
- Your code must respect the flake8 linter standards
- Each exercise must be in its own file
- Each file should contain only the requested function
- Function names must match exactly what is requested
- You don't need to handle input validation or error cases unless explicitly mentioned
- For negative numbers or invalid inputs, the behavior is undefined (you don't need to handle these cases)
- Type hints are optional but recommended for learning purposes (will be introduced in Exercise 7)

Growing Code Specific Note: Since this is an introductory project focusing on basic Python syntax, you only need to submit your individual Python function files (e.g., ft_hello_garden.py, ft_plot_area.py, etc.). Advanced project management tools will be introduced in later projects.

Chapter V

Exercise 0: Hello Garden

Exercise0
ft_hello_garden
Directory: *ex0/*
Files to Submit: `ft_hello_garden.py`
Authorized: `print()`

Welcome to your first Python function! Write a function named `ft_hello_garden` that simply displays a welcome message to the garden community.

```
>>> ft_hello_garden()
Hello, Garden Community!
```

This is your first step into Python programming. Notice how functions can display messages.

Chapter VI

Exercise 1: Garden Plot Area

	Exercise1
ft_plot_area	
Directory: *ex1/*	
Files to Submit: `ft_plot_area.py`	
Authorized: `input(), int(), print()`	

A community garden needs to know the area of a rectangular plot. Write a function named `ft_plot_area` that asks for length and width, then calculates and displays the area.

```
>>> ft_plot_area()
Enter length: 5
Enter width: 3
Plot area: 15
```

Notice how you can store numbers and do calculations with them.

Chapter VII

Exercise 2: Harvest Total

	Exercise2
ft_harvest_total	
Directory: *ex2/*	
Files to Submit: `ft_harvest_total.py`	
Authorized: `input()`, `int()`, `print()`	

A gardener harvested vegetables on 3 different days. Write a function named `ft_harvest_total` that asks for the weight of each harvest and calculates the total.

```
>>> ft_harvest_total()
Day 1 harvest: 5
Day 2 harvest: 8
Day 3 harvest: 3
Total harvest: 16
```

Write only the function ft_harvest_total(), not a main program. The function should handle input and output directly.

See how you can add numbers together and store the result.

Chapter VIII

Exercise 3: Plant Age Check

	Exercise3
ft_plant_age	
Directory: *ex3/*	
Files to Submit: `ft_plant_age.py`	
Authorized: `input()`, `int()`, `print()`	

Write a function named `ft_plant_age` that asks for a plant's age in days and tells if it's ready to harvest (more than 60 days) or not.

```
>>> ft_plant_age()
Enter plant age in days: 75
Plant is ready to harvest!
>>> ft_plant_age()
Enter plant age in days: 45
Plant needs more time to grow.
```

Write only the function ft_plant_age(), not a main program. The function should handle input and output directly.

Programs can make decisions based on the values they receive.

Chapter IX

Exercise 4: Water Reminder

	Exercise4
ft_water_reminder	
Directory: *ex4/*	
Files to Submit: `ft_water_reminder.py`	
Authorized: `input()`, `int()`, `print()`	

Write a function named `ft_water_reminder` that asks for the number of days since last watering. If it's more than 2 days, print "Water the plants!", otherwise print "Plants are fine".

```
>>> ft_water_reminder()
Days since last watering: 4
Water the plants!
>>> ft_water_reminder()
Days since last watering: 1
Plants are fine
```

Write only the function ft_water_reminder(), not a main program. The function should handle input and output directly.

Notice how programs can give different responses based on conditions.

Chapter X

Exercise 5: Count to Harvest

	Exercise5
ft_count_harvest	
Directory: *ex5/*	
Files to Submit: `ft_count_harvest_iterative.py`, `ft_count_harvest_recursive.py`	
Authorized: `input()`, `int()`, `print()`, `range()`	

Write two functions named `ft_count_harvest_iterative` and `ft_count_harvest_recursive`. Both functions should count from 1 to a given number, printing each day until harvest time.

```
>>> ft_count_harvest_iterative()
Days until harvest: 5
Day 1
Day 2
Day 3
Day 4
Day 5
Harvest time!

>>> ft_count_harvest_recursive()
Days until harvest: 5
Day 1
Day 2
Day 3
Day 4
Day 5
Harvest time!
```

Write only the functions `ft_count_harvest_iterative()` and `ft_count_harvest_recursive()`, not a main program. The functions should handle input and output directly.

Sometimes you need to repeat actions. You can do this either with loops (iteration) or by calling the same function again (recursion). What happens when you do the same thing multiple times?

Chapter XI

Exercise 6: Garden Summary

	Exercise6
ft_garden_summary	
Directory: *ex6/*	
Files to Submit: `ft_garden_summary.py`	
Authorized: `input(), print()`	

Write a function named `ft_garden_summary` that asks for a garden name and the number of plants, then displays a simple summary with a fixed status message.

```
>>> ft_garden_summary()
Enter garden name: Community Garden
Enter number of plants: 25
Garden: Community Garden
Plants: 25
Status: Growing well!
```

Write only the function ft_garden_summary(), not a main program. The function should handle input and output directly.

See how you can combine different pieces of information to create something useful. Note that "Status: Growing well!" is a fixed message that should always be displayed exactly as shown.

Chapter XII

Exercise 7: Seed Inventory with Type Annotations

	Exercise7
ft_seed_inventory	
Directory: *ex7/*	
Files to Submit: `ft_seed_inventory.py`	
Authorized: `print()`	

The garden coordinator needs to track seed inventory with precise data types. Write a function named `ft_seed_inventory` that manages seed packets, displaying information about different seed types and quantities.

```
>>> ft_seed_inventory("tomato", 15, "packets")
Tomato seeds: 15 packets available
>>> ft_seed_inventory("carrot", 8, "grams")
Carrot seeds: 8 grams total
>>> ft_seed_inventory("lettuce", 12, "area")
Lettuce seeds: covers 12 square meters
```

Your function signature must look like this: `def ft_seed_inventory(seed_type: str, quantity: int, unit: str) -> None:`

Support these units: "packets" (packets available), "grams" (grams total), "area" (covers X square meters). For any other unit, print "Unknown unit type". Write only the function, not a main program.

Chapter XIII

Helper Resources

To support your learning journey, we've provided a `main.py` file that helps you test your exercises easily.
Since you're just starting with Python and don't know how to write main functions yet, this helper will:

- Import and run your exercise functions automatically
- Show clear error messages if something goes wrong
- Let you test individual exercises or all at once
- Help you understand how Python imports work

To use the helper, simply run `python3 main.py` in your terminal and choose which exercise to test. Make sure your exercise files (like `ft_plot_area.py`) are in the same folder as `main.py`.

The helper file is designed to be readable and educational. Feel free to look at the code to understand how it works, but focus on writing your own exercise solutions first.

The helper file is for learning support only. Your submitted solutions should be your own work and demonstrate your understanding of the concepts.

Chapter XIV

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 your code, trace
through execution, or modify your solutions.  Make sure you
understand every line you write.
```

```
You need to return only the files requested by the subject of this
project.  Focus on clean, readable code that demonstrates your
understanding of Python fundamentals.
```