Data Quest

Mastering Python Collections for Data Engineering

Summary: Journey through the digital realm as a data engineer! Master Python's powerful data structures while building game analytics systems, processing player statistics, and creating streaming data pipelines in the Pixel Dimension.

Version: 2.0

Contents

Chapter I

Foreword

Welcome back to the digital realm, data engineer!

Your journey through Python's foundations has prepared you well. You've mastered the basic syntax that powers digital gardens, built robust class hierarchies that model real-world systems, and learned to handle the unexpected with graceful exception management. Now, as you step into the **Pixel Dimension**, you're ready to tackle the heart of data engineering: **collections and data structures**.

Picture this: In 1980, Pac-Man's entire game state—every dot, ghost position, and score—fit in just 16KB of RAM. The programmers had to be wizards of efficiency! They discovered that organizing data wasn't just about saving memory—it was about unlocking gaming magic. Fast-forward to today: Fortnite processes over 10 million concurrent players, each generating thousands of data points per second. Same principles, bigger playground!

Python's collection types are like your gaming inventory: **lists** are your trusty backpack (ordered, expandable), **tuples** are your equipped armor (immutable, reliable), **sets** are your achievement collection (unique, no duplicates), and **dictionaries** are your spell book (instant lookups by name). Each has its superpower!

But here's where it gets epic: **generators** and **comprehensions** are Python's ultimate combo moves. Imagine processing a million player records without breaking a sweat—that's generator magic! Or transforming complex data with one elegant line—that's comprehension power!

In this quest, you'll build the "PixelMetrics 3000"—a game analytics platform that would make even the developers of Minecraft jealous. Every exercise unlocks a new data superpower, and by the end, you'll be wielding Python collections like a data engineering legend!

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 Data Quest: The Pixel Dimension!

You've conquered Python basics, mastered classes, and tamed exceptions. Now it's time for the fun part: becoming a data wizard! You're about to build the "PixelMetrics 3000"—the most epic game analytics platform this side of the digital universe.

Think of this as your Pokemon journey, but instead of catching creatures, you're collecting data superpowers:

- **Level 0**: Command Quest - Master command-line communication
- **Level 1**: Score Cruncher - Master lists by analyzing player scores
- **Level 2**: Position Tracker - Use tuples to navigate game worlds
- **Level 3**: Achievement Hunter - Leverage sets to track unique accomplishments
- **Level 4**: Inventory Master - Build complex systems with dictionaries
- **Level 5**: Stream Wizard - Process infinite data with generators
- **Level 6**: Data Alchemist - Transform everything with comprehensions

Each exercise is like unlocking a new ability in your favorite RPG. Start simple, level up gradually, and by the end you'll be the Neo of Python data structures!

⚠ `IMPORTANT: This project focuses on` **`data structure mastery`**`. Your programs should demonstrate practical data engineering scenarios while showcasing the unique strengths of each collection type.`

Chapter IV

Common Instructions

IV.1 General Rules

- Your project must be written in **Python 3.10 or later**.
- Your project must adhere to the **flake8** coding standard.
- All functions and methods must include type hints.
- Your functions should handle exceptions gracefully to avoid crashes.
- For this project, only **sys** import is allowed for command-line processing.
- No file I/O operations are allowed - all data must be processed in-memory or via command-line arguments.
- Focus on demonstrating collection usage patterns clearly.
- Show both basic operations and advanced techniques for each data structure.

IV.2 Additional Guidelines

- Create test programs to verify project functionality (not submitted or graded).
- Submit your work to the assigned Git repository.
- Only the content in this repository will be graded.

IV.3 Testing Resources

To help you test your implementations and generate sample data, we've provided helpful tools in a testing archive:

- **Testing Archive**: Extract `data_quest_tools.tar.gz` (provided alongside the project) to access testing utilities

- **Main Data Generator**: `data_generator.py` - Generate command-line ready data for all exercises (0-6)
- **Exercise 0 Helper**: `exercise_0_help.py` - Discover sys.argv and command-line arguments
- **Exercise 1 Helper**: `exercise_1_helper.py` - Elegant score analytics with realistic data patterns
- **Advanced Helper**: `advanced_data_helper.py` - Complex data scenarios and performance testing

Extract the tools archive with: `tar -xzf data_quest_tools.tar.gz`

Quick Start Examples:

```
# Generate test commands for all exercises
python3 data_generator.py

# Get specific exercise help
python3 exercise_0_help.py
python3 exercise_1_helper.py

# Generate command-line ready data
python3 data_generator.py 1 --count 10 --format argv
```

These tools generate command-line ready data that you can copy and paste directly into your terminal. No file operations needed - everything works with the concepts you already know!

The testing tools demonstrate elegant Python patterns and data generation techniques. They're designed to use only concepts appropriate for your current learning level - no advanced file I/O or JSON processing required.

Chapter V

Exercise 0: Command Quest

	Exercise0
ft_command_quest	
Directory: *ex*0/	
Files to Submit: `ft_command_quest.py`	
Authorized: `sys, sys.argv, len(), print()`	

Welcome, Data Adventurer! Every epic quest begins with understanding your tools. In the digital realm, programs need to receive instructions from the outside world. Your first mission: discover how programs can receive messages from their users!

Your Quest: Build a simple command interpreter that shows you've mastered the art of receiving external data. Think of it like learning to read quest scrolls that players send to your game!

What makes this magical:

- Discover how programs can receive information from the command line
- Learn to process different types of input data
- Handle cases where no information is provided
- Display information in a user-friendly way

Power-up tip: Every program is like a character in an RPG—it needs to know what the player wants it to do. The command line is how players communicate their wishes!

```
$> python3 ft_command_quest.py
=== Command Quest ===
No arguments provided!
Program name: ft_command_quest.py
Total arguments: 1

$> python3 ft_command_quest.py hello world 42
=== Command Quest ===
Program name: ft\_command\_quest.py
Arguments received: 3
Argument 1: hello
Argument 2: world
Argument 3: 42
Total arguments: 4

$> python3 ft_command_quest.py "Data Quest"
=== Command Quest ===
Program name: ft_command_quest.py
Arguments received: 1
Argument 1: Data Quest
Total arguments: 2
```

How does your program know what the user wants it to do? What's the difference between the program name and the arguments?

Chapter VI

Exercise 1: Score Cruncher

	Exercise1
ft_score_analytics	
Directory: *ex1/*	
Files to Submit: `ft_score_analytics.py`	
Authorized: `sys.argv, len(), sum(), max(), min(), int(), print()`	

`This exercise requires the use of lists to store scores and try/except blocks to handle invalid input gracefully (e.g., when a user provides non-numeric values).`

Mission Briefing: Now that you've mastered command communication, time for your first real data quest! The PixelMetrics 3000 needs a Score Cruncher module. Think of it like the leaderboard system in your favorite game—but you're building the engine that powers it!

Your mission (should you choose to accept it):

- Accept player scores from the command line (like cheat codes, but legal!)
- Use **lists** to store and organize the scores
- Calculate some basic stats that would make any game dev happy
- Handle the "oops, I typed 'banana' instead of '1000'" scenarios gracefully
- Make the output look cool enough to impress your gaming buddies

Power-up tip: Lists are like your inventory in an RPG—you can add items, count them, find the best one, and organize them however you want!

```
python3 ft_score_analytics.py 1500 2300 1800 2100 1950
```

```
$> python3 ft_score_analytics.py 1500 2300 1800 2100 1950
=== Player Score Analytics ===
Scores processed: [1500, 2300, 1800, 2100, 1950]
Total players: 5
Total score: 9650
Average score: 1930.0
High score: 2300
Low score: 1500
Score range: 800

$> python3 ft_score_analytics.py
=== Player Score Analytics ===
No scores provided. Usage: python3 ft_score_analytics.py <score1> <score2> ...
```

How do **lists** help you process sequential data? What makes them perfect for command-line data processing?

Chapter VII

Exercise 2: Position Tracker

	Exercise2
ft_coordinate_system	
Directory: *ex2/*	
Files to Submit: `ft_coordinate_system.py`	
Authorized: `sys, sys.argv, math, math.sqrt(), tuple(), int(), float(), print(), split()`	

`This exercise requires the use of tuples to store 3D coordinates (x, y, z) and try/except blocks to handle parsing errors when converting string input to numbers.`

Level Up! Time to master 3D coordinates! Remember playing games where you teleport to specific locations in a 3D world? Or when you need to find the distance between two points in 3D space? That's exactly what we're building!

Your Quest: Build a 3D coordinate system using **tuples**. Think of tuples as GPS coordinates that can't be accidentally changed—perfect for game positions!

What makes this fun:

- Create 3D positions like a game's spawn points: `(x, y, z)`
- Calculate distances using the 3D Euclidean distance formula: `sqrt((x2-x1)² + (y2-y1)² + (z2-z1)²)`
- Parse coordinate strings (like teleport commands!)
- Show off tuple unpacking magic (it's like unwrapping a present!)

Power-up tip: Tuples are like coordinates written in stone—once created, they won't change. Perfect for 3D positions, colors, or any data that should stay put!

Distance Formula Explained: To calculate the distance between two 3D points, we use the Euclidean distance formula. For points `(x1, y1, z1)` and `(x2, y2, z2)`, the distance is `math.sqrt((x2-x1)**2 + (y2-y1)**2 + (z2-z1)**2)`. This is just the 3D extension of the Pythagorean theorem!

Remember to handle the classic "I typed 'abc' instead of '123'" error gracefully!

```
$> python3 ft_coordinate_system.py
=== Game Coordinate System ===

Position created: (10, 20, 5)
Distance between (0, 0, 0) and (10, 20, 5): 22.91

Parsing coordinates: "3,4,0"
Parsed position: (3, 4, 0)
Distance between (0, 0, 0) and (3, 4, 0): 5.0

Parsing invalid coordinates: "abc,def,ghi"
Error parsing coordinates: invalid literal for int() with base 10: 'abc'
Error details - Type: ValueError, Args: ("invalid literal for int() with base 10: 'abc'",)

Unpacking demonstration:
Player at x=3, y=4, z=0
Coordinates: X=3, Y=4, Z=0
```

Why are **tuples** perfect for 3D coordinate data? How does **unpacking** make your code more readable and powerful?

Chapter VIII

Exercise 3: Achievement Hunter

	Exercise3
ft_achievement_tracker	
Directory: *ex3/*	
Files to Submit: `ft_achievement_tracker.py`	
Authorized: `set()`, `len()`, `print()`, `union()`, `intersection()`, `difference()`	

`This exercise requires the use of sets to store unique achievements and set operations (union, intersection, difference) to analyze achievement collections across players.`

Achievement Unlocked! Time to build the coolest achievement system ever! You know how satisfying it is when you unlock that rare achievement? Now you're building the system that tracks them all!

Your Mission: Create an Achievement Hunter using **sets**—the perfect tool for handling unique collections. No duplicates allowed in the hall of fame!

What makes this epic:

- Track unique achievements (no "First Kill" counted twice!)
- Find achievements shared by multiple players (the "common ground")
- Spot the ultra-rare achievements (bragging rights material!)
- See who's missing what achievements (gotta catch 'em all!)
- Build player communities based on shared accomplishments

Power-up tip: Sets are like your trophy case—each achievement appears exactly once, and you can instantly check if you have it or compare collections with friends!

```
$> python3 ft_achievement_tracker.py
=== Achievement Tracker System ===

Player alice achievements: {'first_kill', 'level_10', 'treasure_hunter', 'speed_demon'}
Player bob achievements: {'first_kill', 'level_10', 'boss_slayer', 'collector'}
Player charlie achievements: {'level_10', 'treasure_hunter', 'boss_slayer', 'speed_demon', '
    perfectionist'}

=== Achievement Analytics ===
All unique achievements: {'boss_slayer', 'collector', 'first_kill', 'level_10', 'perfectionist', '
    speed_demon', 'treasure_hunter'}
Total unique achievements: 7

Common to all players: {'level_10'}
Rare achievements (1 player): {'collector', 'perfectionist'}

Alice vs Bob common: {'first_kill', 'level_10'}
Alice unique: {'speed_demon', 'treasure_hunter'}
Bob unique: {'boss_slayer', 'collector'}
```

How do **sets** make data deduplication effortless? What makes set operations perfect for analytics?

Chapter IX

Exercise 4: Inventory Master

	Exercise4
ft_inventory_system	
Directory: *ex4/*	
Files to Submit: `ft_inventory_system.py`	
Authorized: `dict()`, `len()`, `print()`, `keys()`, `values()`, `items()`, `get()`, `update()`	

`This exercise requires the use of dictionaries to store inventory data with nested structures (items containing name, type, quantity, value). You must use dict methods like keys(), values(), items(), get(), and update() to manage the inventory.`

Loot Time! Remember organizing your inventory in RPGs? Checking if you have that legendary sword? Time to build the ultimate inventory system!

Your Quest: Create an Inventory Master using **dictionaries**; your magical storage system where you can instantly find any item by name!

What makes this awesome:

- Manage game inventory using dictionaries (like your personal treasure chest!)
- Track item quantities with key-value pairs
- Calculate inventory statistics (most/least abundant items)
- Organize items by abundance categories using nested dictionaries
- Generate inventory reports with dictionary operations

Power-up tip: Dictionaries are like having a super-organized backpack where you can instantly grab any item by saying its name. No more digging through everything to find that health potion!

```
$> python3 ft_inventory_system.py sword:1 potion:5 shield:2 armor:3 helmet:1
=== Inventory System Analysis ===
Total items in inventory: 12
Unique item types: 5

=== Current Inventory ===
potion: 5 units (41.7%)
armor: 3 units (25.0%)
shield: 2 units (16.7%)
sword: 1 unit (8.3%)
helmet: 1 unit (8.3%)

=== Inventory Statistics ===
Most abundant: potion (5 units)
Least abundant: sword (1 unit)

=== Item Categories ===
Moderate: {'potion': 5}
Scarce: {'sword': 1, 'shield': 2, 'armor': 3, 'helmet': 1}

=== Management Suggestions ===
Restock needed: ['sword', 'helmet']

=== Dictionary Properties Demo ===
Dictionary keys: ['sword', 'potion', 'shield', 'armor', 'helmet']
Dictionary values: [1, 5, 2, 3, 1]
Sample lookup - 'sword' in inventory: True
```

Why are **dictionaries** essential for game data? How do nested dictionaries model complex relationships?

Chapter X

Exercise 5: Stream Wizard

	Exercise5
ft_data_stream	
Directory: *ex5/*	
Files to Submit: `ft_data_stream.py`	
Authorized: `next(), iter(), range(), len(), print()`	

`This exercise requires the use of generators with the 'yield' keyword to create memory-efficient data streams. You must implement generator functions that produce values on-demand rather than storing everything in memory.`

Magic Time! Ever wondered how games handle millions of events without crashing? Welcome to the world of **generators**—Python's memory-saving superpower!

Your Quest: Build a Stream Wizard that processes data like a pro! Think of generators as magic spells that create data on-demand instead of storing everything at once.

What makes this magical:

- Create data streams that flow like a river (not a lake!)
- Process events one by one using **for-in loops** (like reading a book page by page)
- Filter interesting events (only the good stuff!)
- Keep track of statistics without storing everything (memory magic!)
- Show the difference between "store everything" vs "stream everything"

Power-up tip: Generators are like having a magical data fountain—they create exactly what you need, when you need it, without wasting memory on stuff you don't need yet!

```
$> python3 ft_data_stream.py
=== Game Data Stream Processor ===

Processing 1000 game events...

Event 1: Player alice (level 5) killed monster
Event 2: Player bob (level 12) found treasure
Event 3: Player charlie (level 8) leveled up
...

=== Stream Analytics ===
Total events processed: 1000
High-level players (10+): 342
Treasure events: 89
Level-up events: 156

Memory usage: Constant (streaming)
Processing time: 0.045 seconds

=== Generator Demonstration ===
Fibonacci sequence (first 10): 0, 1, 1, 2, 3, 5, 8, 13, 21, 34
Prime numbers (first 5): 2, 3, 5, 7, 11
```

How do **generators** enable memory-efficient processing? What makes **for-in loops** perfect for streaming data?

Chapter XI

Exercise 6: Data Alchemist

	Exercise6
ft_analytics_dashboard	
Directory: *ex6/*	
Files to Submit: `ft_analytics_dashboard.py`	
Authorized: `len()`, `print()`, `sum()`, `max()`, `min()`, `sorted()`	

`This exercise requires the use of list comprehensions, dict comprehensions, and set comprehensions to transform and filter data efficiently. These are fundamental Python features for elegant data processing.`

Final Boss Time! You've mastered all the data structures—now it's time to combine them into the ultimate analytics dashboard! This is where you become a true Data Alchemist!

Your Epic Quest: an analytics dashboard using **comprehensions** — Python's most elegant way to transform data. Think of comprehensions as magic spells that turn raw data into pure insights!

Core Requirements:

- Demonstrate **list comprehensions** for filtering and transforming data
- Demonstrate **dict comprehensions** for creating mappings and grouping data
- Demonstrate **set comprehensions** for finding unique values
- Process sample gaming data (scores, players, achievements, etc.)
- Show clear examples of each comprehension type in action

What to Focus On:

- **List comprehensions**: Filter high scores, transform data, create new lists
- **Dict comprehensions**: Group players by category, count occurrences, create mappings
- **Set comprehensions**: Find unique players, unique achievements, deduplicate data
- Combine comprehensions with the data structures from previous exercises
- Keep it simple—focus on demonstrating comprehension mastery, not building complex analytics

Power-up tip: Comprehensions are like having a magic wand—one elegant line of code can transform entire datasets. It's the difference between writing 10 lines of loops vs 1 line of pure Python magic!

Example Output (showing possible analytics - your implementation may vary):

```
$> python3 ft_analytics_dashboard.py
=== Game Analytics Dashboard ===

=== List Comprehension Examples ===
High scorers (>2000): ['alice', 'charlie', 'diana']
Scores doubled: [4600, 3600, 4300, 4100]
Active players: ['alice', 'bob', 'charlie']

=== Dict Comprehension Examples ===
Player scores: {'alice': 2300, 'bob': 1800, 'charlie': 2150}
Score categories: {'high': 3, 'medium': 2, 'low': 1}
Achievement counts: {'alice': 5, 'bob': 3, 'charlie': 7}

=== Set Comprehension Examples ===
Unique players: {'alice', 'bob', 'charlie', 'diana'}
Unique achievements: {'first_kill', 'level_10', 'boss_slayer'}
Active regions: {'north', 'east', 'central'}

=== Combined Analysis ===
Total players: 4
Total unique achievements: 12
Average score: 2062.5
Top performer: alice (2300 points, 5 achievements)
```

This exercise combines all the data structures and techniques you've learned. Focus on demonstrating **all three types of comprehensions** clearly. The example shows possibilities-your implementation should demonstrate comprehension mastery, not replicate the exact output.

Keep it simple! The goal is to master comprehensions, not build a complex analytics system. Use simple, hardcoded sample data (lists, dicts, sets) to demonstrate each comprehension type. Don't overcomplicate-clarity and comprehension mastery are what matter!

How do **comprehensions** make complex data transformations readable? What makes them essential for data engineering workflows?

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 data structure choices, demonstrate collection operations, or extend your analytics systems with new functionality. Make sure you understand the principles behind each data structure.`

`You need to return only the files requested by the subject of this project. Focus on clean, well-documented code that clearly demonstrates mastery of Python's collection types and data processing techniques.`