

Task 1: Book Class Generation

The screenshot shows a VS Code interface with a Python file named 1079.py open. The code defines a Book class with __init__ and summary methods, and a student_commentary function. A Chat sidebar provides AI-generated student commentary and todo items.

```
1079.py
1  class Book:
2      def __init__(self, title, author):
3          self.title = title
4          self.author = author
5
6      def summary(self):
7          return f"This book '{self.title}' is written by {self.author}."
8
9
10     def student_commentary() -> str:
11         return (
12             "Student commentary on code quality:\n"
13             "- Correctness: The `summary()` method returns the expected string.\n"
14             "- Readability: Clear and concise; uses an f-string. Add docstrings and type hints for clarity.\n"
15             "- Efficiency: Efficient; previous implementation used unnecessary loops.\n"
16             "- Maintainability: Consider adding input validation and `__repr__` for d"
17         )
18
19
20 if __name__ == "__main__":
21     demo = Book("The Little Prince", "Antoine de Saint-Exupéry")
22     print(demo.summary())
23     print()

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

PS C:\Users\tloke\OneDrive\Documents\AI_79_Assignments> & C:/Users/tloke/AppData/Local/Python/pythoncore-3.14-64/python.exe c:/Users/tloke/OneDrive/Documents/AI_79_Assignments/1079.py
This book 'The Little Prince' is written by Antoine de Saint-Exupéry.

Student commentary on code quality:
- Correctness: The `summary()` method returns the expected string.
- Readability: Clear and concise; uses an f-string. Add docstrings and type hints for clarity.

Student commentary on code quality:
- Correctness: The `summary()` method returns the expected string.
- Readability: Clear and concise; uses an f-string. Add docstrings and type hints for clarity.

CHAT

Using Cursor AI to generate a Python class Book with attributes title, author, and a summary() method.
Expected Output is Student commentary on code quality

✓ Reading file content
I'll add a short plan for the task, then update 1079.py with a Book class and student commentary.

✓ Created 3 todos
Reading the current 1079.py to avoid overwriting existing content.

✓ Updated and simplified Book class implementation
Now I'll simplify the Book class implementation and add a concise student commentary block.

✓ Edited 1079.py +14 -23
✓ Updating todo list statuses
Marking the todo items as completed.

✓ Completed: Add Book class (1/3)
I'll run the script to show the summary and student commentary

> Todos (3/3)

Describe what to build next

Agent Auto

Task 2: Sorting Dictionaries with AI

The screenshot shows a VS Code interface with a Python file named Task02.py open. The code demonstrates sorting a list of dictionaries by key using sorted() with lambda. A Chat sidebar provides AI-generated code options and execution details.

```
Task02.py
1 """
2 Sorting a List of Dictionaries by Key - Multiple Approaches
3 Comparison of Clarity and Performance
4 """
5
6 import timeit
7 from operator import itemgetter
8
9 # Sample data
10 students = [
11     {"name": "Alice", "grade": 85, "age": 20},
12     {"name": "Bob", "grade": 92, "age": 19},
13     {"name": "Charlie", "grade": 78, "age": 21},
14     {"name": "Diana", "grade": 88, "age": 20},
15 ]
16
17 print("=" * 70)
18 print("APPROACH 1: Using sorted() with lambda (Most Readable)")
19 print("=" * 70)
20
21 # Simple, readable approach using lambda
22 sorted_by_grade_lambda = sorted(students, key=lambda x: x["grade"])
23 print("Sorted by grade (ascending):")
24 print(sorted_by_grade_lambda)

PS C:\Users\tloke\OneDrive\Documents\AI_79_Assignments> & C:/Users/tloke/AppData/Local/Python/pythoncore-3.14-64/python.exe c:/Users/tloke/OneDrive/Documents/AI_79_Assignments/Task02.py  
This book 'The Little Prince' is written by Antoine de Saint-Exupéry.

Student commentary on code quality:  
- Correctness: The `summary()` method returns the expected string.  
- Readability: Clear and concise; uses an f-string. Add docstrings and type hints for clarity.

Student commentary on code quality:  
- Correctness: The `summary()` method returns the expected string.  
- Readability: Clear and concise; uses an f-string. Add docstrings and type hints for clarity.  
- Correctness: The `summary()` method returns the expected string.  
- Readability: Clear and concise; uses an f-string. Add docstrings and type hints for clarity.  
- Readability: Clear and concise; uses an f-string. Add docstrings and type hints for clarity.



CHAT



Use Gemini and Cursor AI to generate code that sorts ...  
Use Gemini and Cursor AI to generate code that sorts a list of dictionaries by key.  
Output  
Both AI outputs  
Comparison of clarity and performance



✓ Clarified limitations and offered code generation options.  
I don't have access to Gemini or Cursor AI directly, but I can create a comprehensive solution in your current file showing multiple approaches to sorting a list of dictionaries by a key, with comparisons. Let me generate this for you:



✓ Read Task02.py  
Now I'll create comprehensive code showing multiple approaches to sort dictionaries with comparisons:



✓ Created Task02.py  
Now let's run the code to show you the output:



Run pwsh command?  
python Task02.py  
Allow Skip



> 1 file changed +139 -1



Describe what to build next



Agent Auto


```

The screenshot shows the VS Code interface with the following details:

- Code Editor:** Task02.py file open, showing Python code for sorting a list of dictionaries by key using various methods.
- Terminal:** Shows the output of the code execution, displaying sorted lists of products based on price and stock.
- AI Interface (right side):**
 - Header: USE GEMINI AND CURSOR AI TO GENERATE CODE THAT SORTS ...
 - Text input: Use Gemini and Cursor AI to generate code that sorts a list of dictionaries by a key.
 - Output: Both AI outputs
 - Comparison of clarity and performance
 - Checklist: Clarity and performance options checked.
 - Text: I don't have access to Gemini or Cursor AI directly, but I can create a comprehensive solution in your current file showing multiple approaches to sorting a list of dictionaries by a key, with comparisons. Let me generate this for you:
 - Buttons: Read (Task02.py), Created (Task02.py)
 - Text: Now I'll create comprehensive code showing multiple approaches to sort dictionaries with comparisons:
 - Buttons: Allow (Task02.py), Skip
 - Text: Now let's run the code to show you the output:
 - Run command: Run psh command?
 - Command: python Task02.py
 - Buttons: Allow (Task02.py), Skip
 - Text: Task02.py +
 - Text: Describe what to build next
 - Agent dropdown: Agent Auto
 - Buttons: > >

Task 3: Calculator Using Functions

The screenshot shows the VS Code interface with the following details:

- Code Editor:** Task03.py file open, showing Python code for a calculator application using functions.
- Terminal:** Shows the output of the code execution, displaying sorted lists of products based on price and stock.
- AI Interface (right side):**
 - Header: SORTING A LIST OF DICTIONARIES BY KEY
 - Text input: Ask Gemini to generate a calculator using functions and explain how it works.
 - Output: Outputs, Calculator code, AI explanation, Screenshot
 - Checklist: Created Python file.
 - Text: Now let me run the calculator with some sample calculations to show the output:
 - Command: echo -e "1\n10\n5\n2\n20\n4\n5\n2\n7\n9\n8" | python Task03.py
 - Text: Enable shell integration to improve command detection
 - Text: Let me create a demonstration script instead:
 - Created: calculator_demo.py
 - Command: python calculator_demo.py
 - Text: Enable shell integration to improve command detection
 - Text: Perfect! I've created a comprehensive function-based calculator. Here's what I've delivered:
 - Calculator Code - Task03.py
 - Operations: 7 Operations
 - Task03.py +
 - Describe what to build next
 - Agent dropdown: Agent Auto
 - Buttons: > >

```

calculator_demo.py > ...
1 """
2 CALCULATOR DEMO - Running multiple calculations
3 """
4
5 # Import the calculator functions
6 def add(x, y):
7     return x + y
8
9 def subtract(x, y):
10    return x - y
11
12 def multiply(x, y):
13    return x * y
14
15 def divide(x, y):
16     if y == 0:
17         raise ValueError("Cannot divide by zero!")
18     return x / y
19
20 def power(x, y):
21     return x ** y
22
23 def modulo(x, y):
24
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
1. Addition      (+)
2. Subtraction   (-)
3. Multiplication (*)
4. Division      (/)
5. Power          (**)
6. Modulo         (%)
7. Square Root    (sqrt)
8. Exit           (exit)
=====
Select an operation (1-8): 8
✓ Thank you for using the calculator. Goodbye!
PS C:\Users\tloke\OneDrive\Documents\AI_79_Assignments>

```

Ask Gemini to generate a calculator using functions and explain how it works.
outputs
Calculator code
AI explanation
Screenshot

✓ Created Python file
Now let me run the calculator with some sample calculations to show the output:

- echo -e "1\n10\n5\n2\n20\n4\n5\n2\n7\n9\n8" | python Task03.py
Enable shell integration to improve command detection

Let me create a demonstration script instead:

- Created calculator_demo.py
python calculator_demo.py
Enable shell integration to improve command detection

Perfect! I've created a comprehensive function-based calculator. Here's what I've delivered:

Calculator Code - Task03.py

7 Operations:

calculator_demo.py +
Describe what to build next
Agent Auto

Task 4: Armstrong Number Optimization

```

Task04.py > ...
1 """
2 ARMSTRONG NUMBER PROGRAM
3 Version Comparison: Basic vs Improved
4 """
5
6 print("=" * 80)
7 print(" " * 20 + "ARMSTRONG NUMBER PROGRAM - TWO VERSIONS")
8 print("=" * 80)
9
10 print("\n" + "=" * 80)
11 print("VERSION 1: BASIC IMPLEMENTATION (Gemini-style)")
12 print("=" * 80)
13
14 print(""""
15 What is an Armstrong Number?
16 An Armstrong number (narcissistic number) is a number that equals the sum of its
17 own digits each raised to the power of the number of digits.
18
19 Examples:
20     153 = 1³ + 5³ + 3³ = 1 + 125 + 27 = 153 ✓ (Armstrong)
21     9474 = 9⁴ + 4⁴ + 7⁴ + 4⁴ = 6561 + 256 + 2401 + 256 = 9474 ✓ (Armstrong)
22     123 = 1³ + 2³ + 3³ = 1 + 8 + 27 = 36 X (Not Armstrong)
23 """)

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
+ ... | ☒ X
Python powershell

```

KEY TAKEAWAYS:

- ✓ Version 1: Works, but not production-ready
- ✓ Version 2: Professional-grade with robustness and extensibility
- ✓ Real-world applications need error handling, documentation, and validation
- ✓ Generators and type hints are Python best practices
- ✓ Good code is about maintainability, not just functionality

=====

PS C:\Users\tloke\OneDrive\Documents\AI_79_Assignments>

← SORTING A LIST OF DICTIONARIES BY KEY

Generate an Armstrong number program using Gemini, then improve it using Cursor AI.
output
Two versions
Summary of improvements

✓ Created Python file
Now let's run it to show the complete output:

- python Task04.py
Enable shell integration to improve command detection

Perfect! I've created a comprehensive Armstrong number program in Task04.py with two versions and detailed improvements. Here's what's included:

Two Versions Side-by-Side

Version 1: Basic (Gemini-style)

- Simple, straightforward implementation
- Returns True/False only
- No error handling
- Works but not production-ready

Version 2: Improved (Cursor AI improvements)

- Full type hints (`int` → `Tuple[bool, Dict]`)
- Comprehensive docstrings with Args>Returns/Raises

Task04.py +
Describe what to build next
Agent Auto

The screenshot shows the Microsoft Visual Studio Code interface with the following components:

- Editor Area:** Displays Python code for finding Armstrong numbers, comparing Version 1 and Version 2. The code includes docstrings, type hints, and error handling.
- Terminal:** Shows the command `PS C:\Users\tloke\OneDrive\Documents\AI_79_Assignments>` indicating the current working directory.
- Bottom Status Bar:** Shows file details: `Task04.py`, line count `Ln 337, Col 1`, character count `Spaces: 4`, encoding `UTF-8`, line endings `CRLF`, language `Python`, version `3.14.2`, and a Go Live button.
- Right Sidebar:**
 - SORTING A LIST OF DICTIONARIES BY KEY**: A list of bullet points: "Statistical analysis (density, average, distribution)" and "Production-grade code".
 - Key Improvements Summary**: A table comparing V1 and V2 across various features.

Feature	V1	V2
Type Hints	X	✓
Documentation	Basic	Comprehensive
Error Handling	X	✓
Calculation Breakdown	X	✓ Shows equation
Memory Efficiency	Loads all data	Generator pattern
Statistics	X	✓ Density analysis
Validation	X	✓ Full validation
Maintainability	Medium	High
 - Results**: A list of findings:
 - Found 19 Armstrong numbers in range 1-100,000
 - Density: 0.019% (1 in 5,263 numbers)
 - Examples: 1, 153, 370, 371, 407, 1634, 8208, 9474, 54748, 92727, 93084
 - Build Status:** Shows a build status bar with icons for file operations.