**ASSIGMENT:6.1**

**Name**: N.Shiva

**Batch:**12

**Htno:** 2403A51298

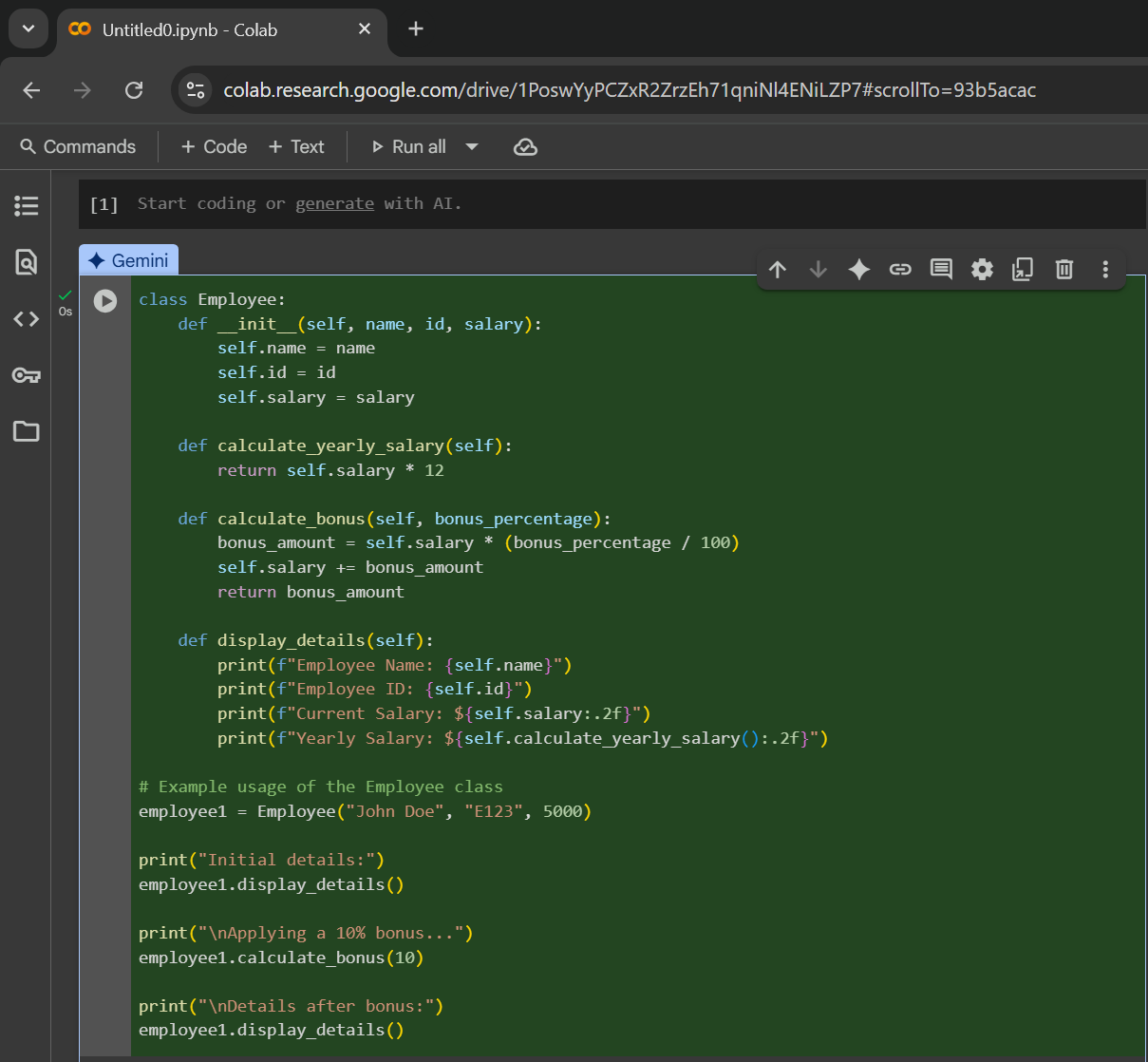
**Task Description #1 (Classes – Employee Management)**

* **Task: Use AI to create an Employee class with attributes (name, id, salary) and a method to calculate yearly salary.**
* **Instructions:**
  + **Prompt AI to generate the Employee class.**
  + **Analyze the generated code for correctness and structure.**
  + **Ask AI to add a method to give a bonus and recalculate salary.**

**Prompt:**

**#Create a Python class Employee with attributes: name, id, and salary. Add methods to calculate yearly salary, apply bonus via calculate\_bonus(), and display\_details() to show employee info and updated salary.**

**Code:**

****

**Output:**

****

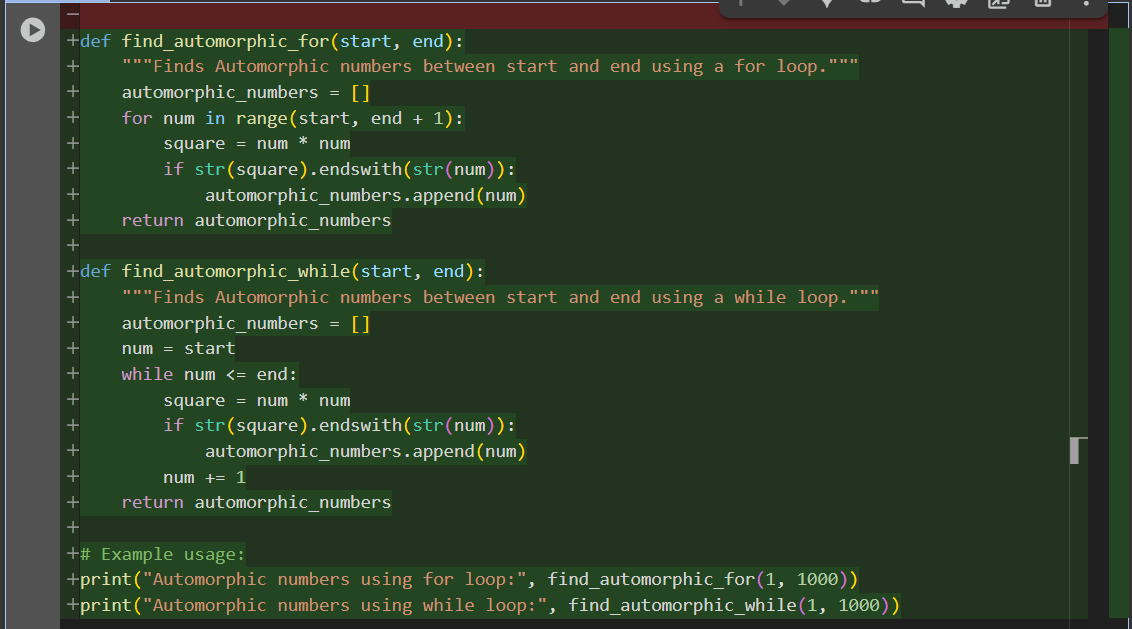
**Task Description #2 (Loops – Automorphic Numbers in a Range)**

* **Task: Prompt AI to generate a function that displays all Automorphic numbers between 1 and 1000 using a for loop.**
* **Instructions:**
  + **Get AI-generated code to list Automorphic numbers using a for loop.**
  + **Analyze the correctness and efficiency of the generated logic.**
  + **Ask AI to regenerate using a while loop and compare both implementations.**

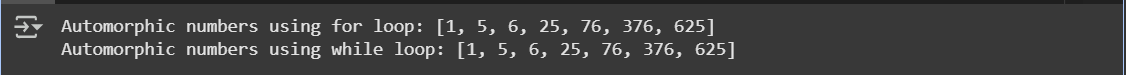
**Prompt:**

#Write a Python function to find Automorphic numbers between 1 and 1000 using a for loop. Rewrite it using a while loop and compare both versions for logic and performance.

**Code:**

****

**Output:**

****

**Comparison:**

**1. Execution Speed**

* Both versions perform the same number of iterations and operations.
* The difference in performance is negligible for small to moderate ranges (like 1 to 1000).
* In Python for loops are slightly faster because they use internal iterator optimizations**.**

**2. Memory Usage**

* Both store results in a list, so memory usage is identical.
* No recursion or heavy data structures involved.

**3. Scalability**

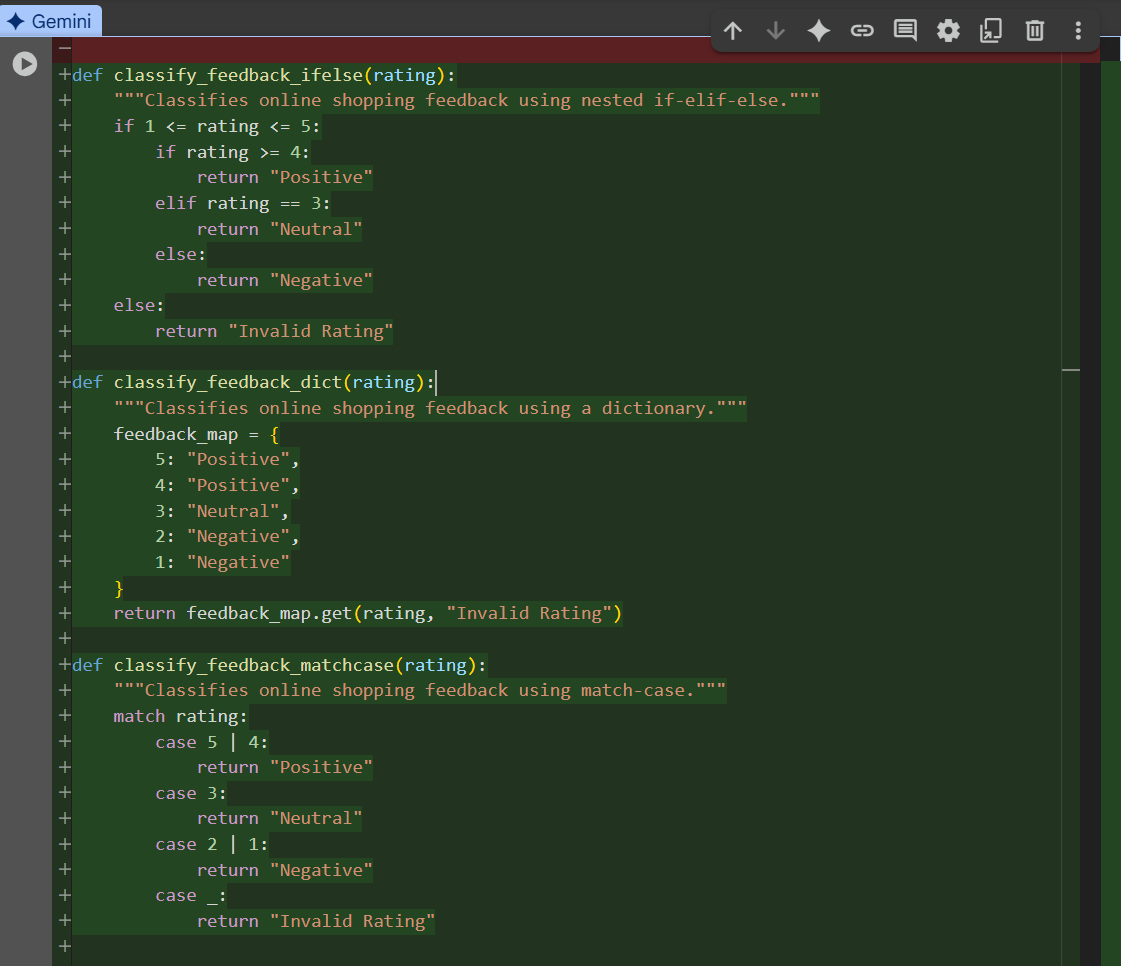
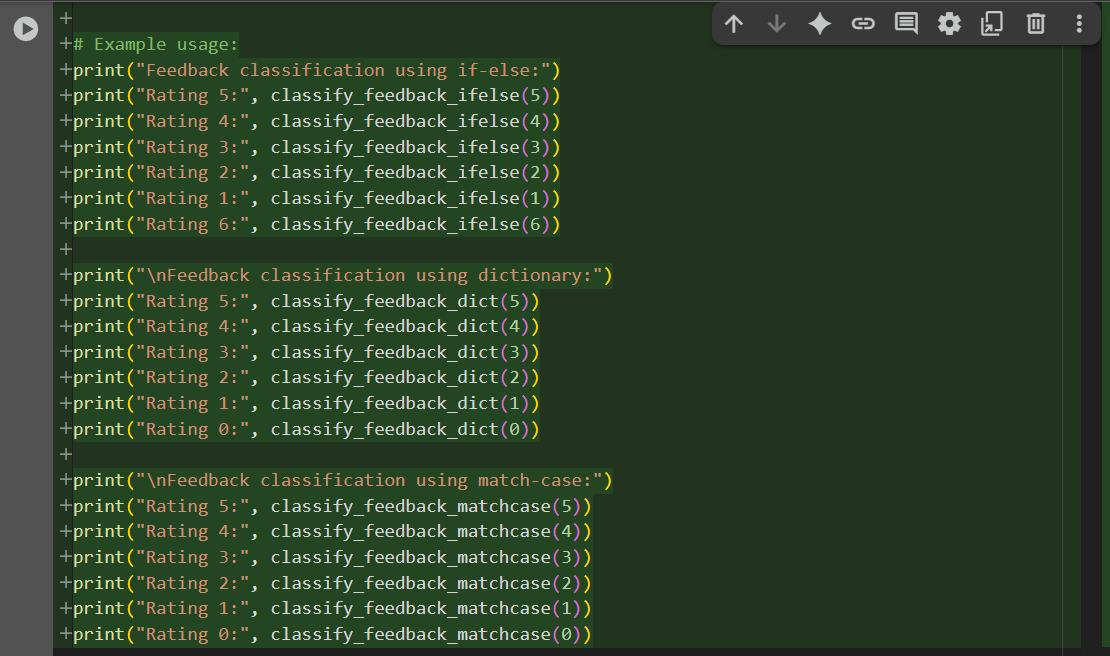
* For very large ranges, performance differences may become noticeable:
* for loop: marginally faster due to optimized iteration.
* while loop: slightly slower due to manual increment and condition checking.

**Task Description #3 (Conditional Statements – Online Shopping Feedback Classification)**

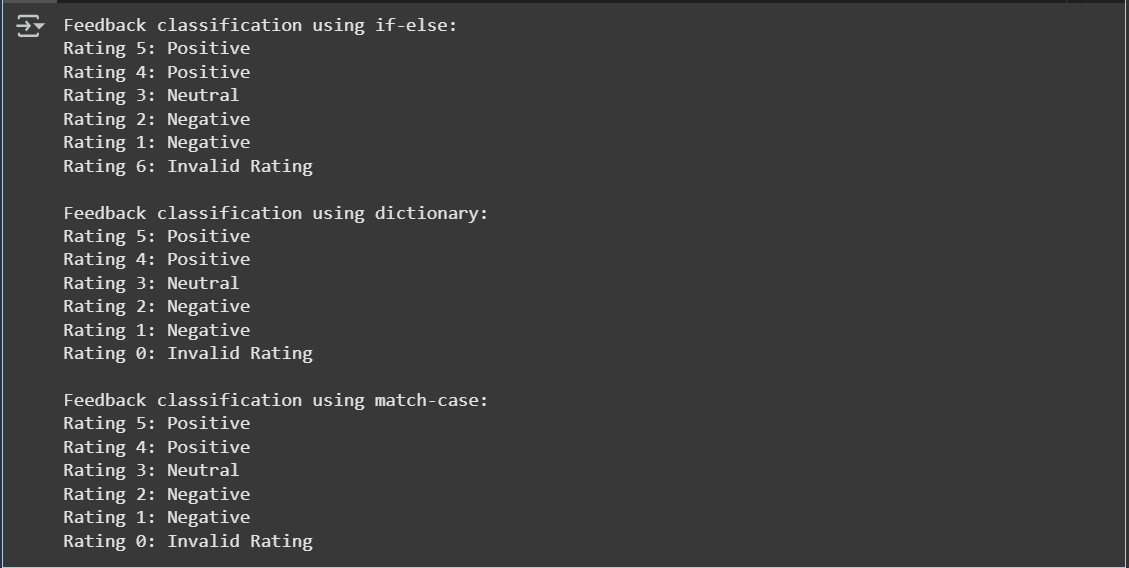
* **Task:** Ask AI to write nested if-elif-else conditions to classify online shopping feedback as Positive, Neutral, or Negative based on a numerical rating (1–5).
* **Instructions:**
  + Generate initial code using nested if-elif-else.
  + Analyze correctness and readability.
  + Ask AI to rewrite using dictionary-based or match-case structure.

**Prompt:**

**#**Write a Python function using nested if-elif-else to classify online shopping feedback (rating 1–5) as Positive, Neutral, or Negative. Then rewrite it using dictionary or match-case.

**Code:******

**Output:**

****

**Analysis:**

All three versions are logically correct and handle valid/invalid ratings properly.

If-else: beginner-friendly, step-by-step logic, but a bit verbose.

Dictionary: concise, elegant, and easy to extend by editing mappings.

.get() neatly handles invalid ratings in the dictionary version.

Match-case: modern, very readable grouping (5 | 4, 2 | 1).

Match-case requires Python 3.10+, so less portable.

If-else is best for teaching and clarity.

Dictionary is best for maintainability with static mappings.

Match-case is best for clean readability in modern Python.

**Overall:** all are correct; the choice depends on the audience and Python version.

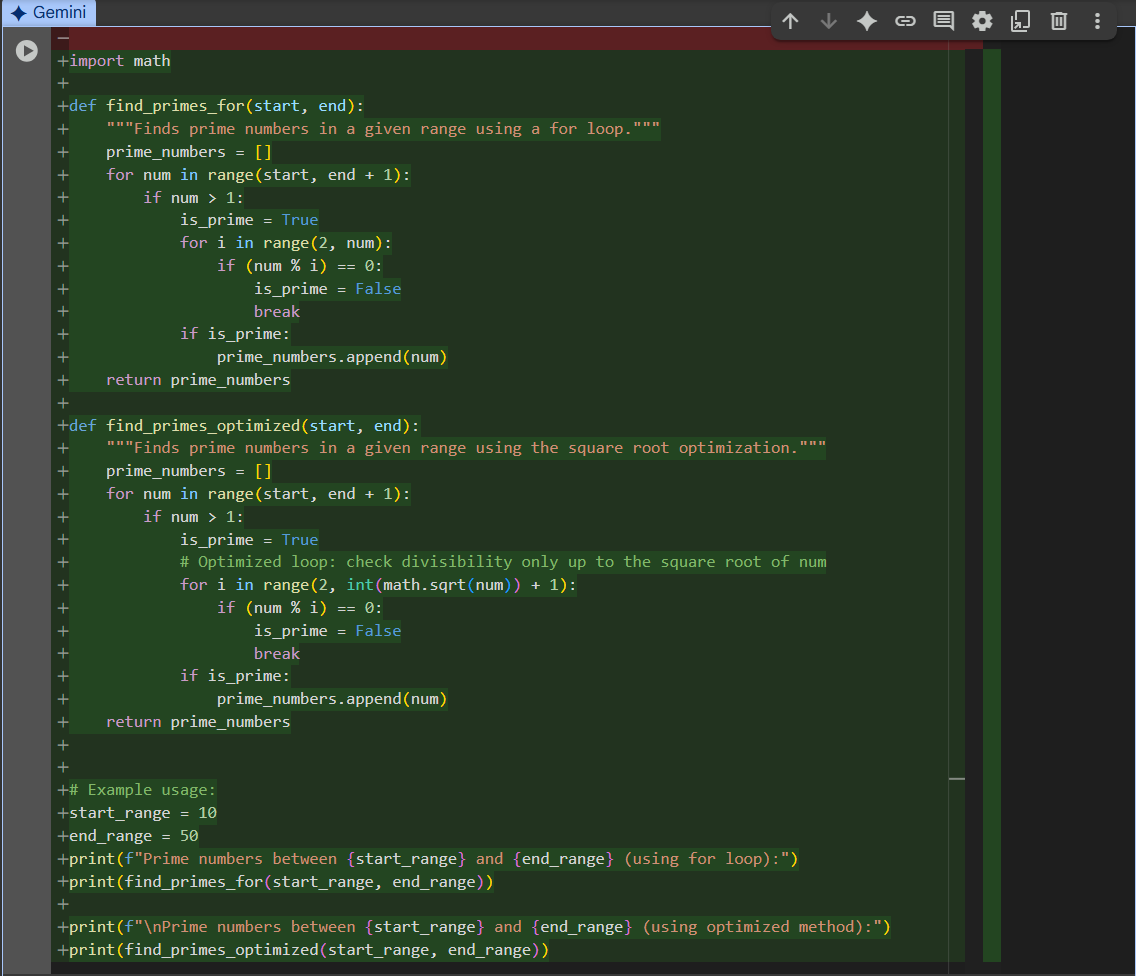
**Task Description #4 (Loops – Prime Numbers in a Range)**

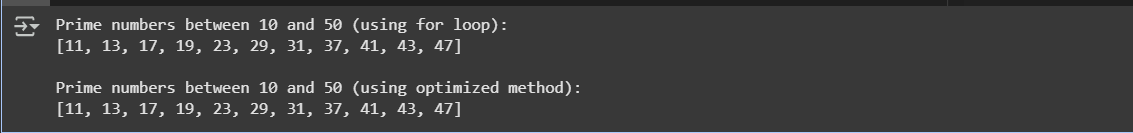
* **Task**: Generate a function using AI that displays all prime numbers within a user-specified range (e.g., 1 to 500).
* **Instructions:**
  + Get AI-generated code to list all primes using a for loop.
  + Analyze the correctness and efficiency of the prime-checking logic.
  + Ask AI to regenerate an optimized version (e.g., using the square root method).

Prompt:

#Write a Python function using a for loop to list all prime numbers in a user-defined range. Then optimize it using the square root method and explain the efficiency improvements.

**Code:**



**Output:  
 **

**Analysis:**

The naive prime-checking method tests all numbers up to num-1, making it slow with time complexity O(n²). The optimized version only checks divisors up to num\sqrt{num}num​, since factors repeat beyond that. This reduces checks drastically, e.g., 9971 vs. 100 for 9973, making it much faster and scalable.

**Task Description #5 (Classes – Library System)**

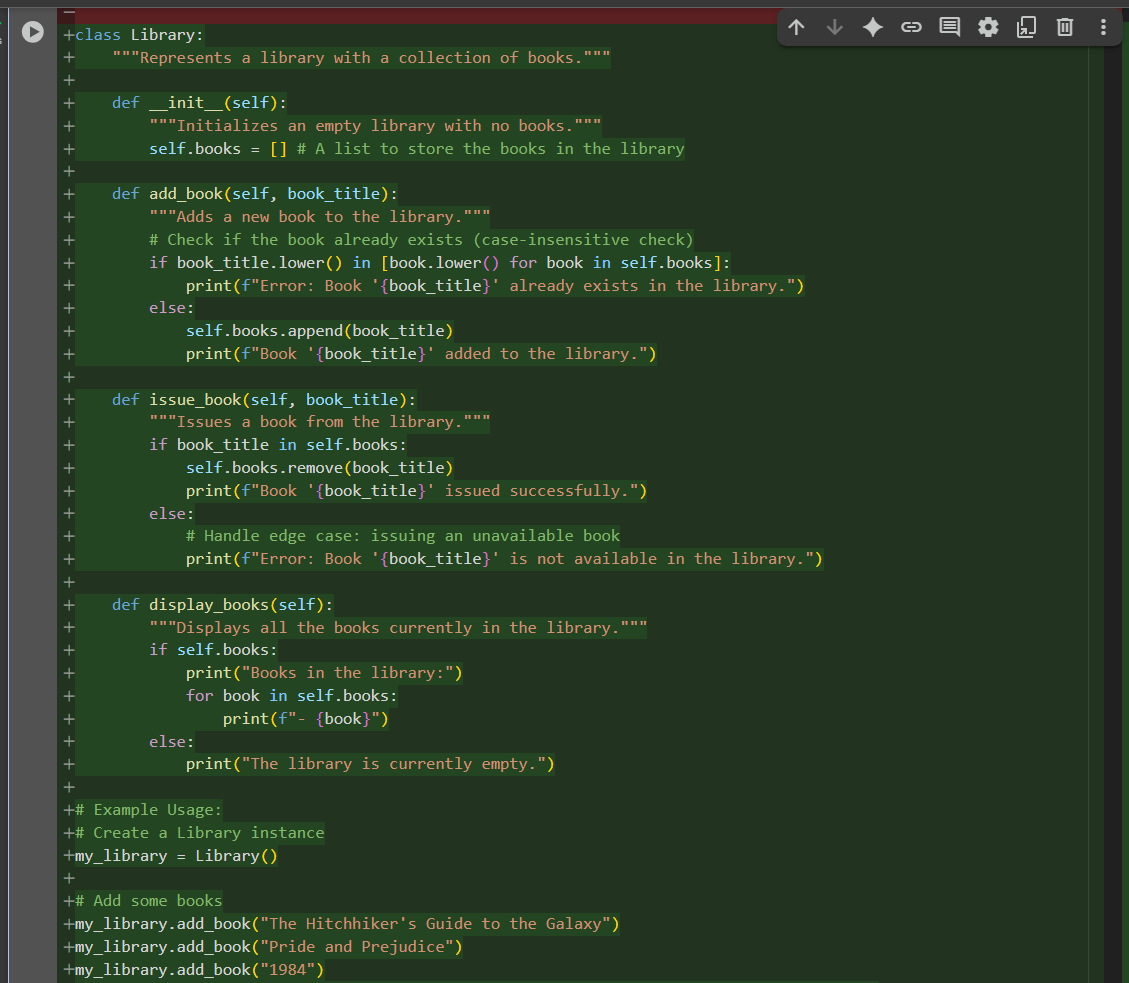
* Task: Use AI to build a Library class with methods to add\_book(), issue\_book(), and display\_books().
* Instructions:
  + Generate Library class code using AI.
  + Analyze if methods handle edge cases (e.g., issuing unavailable books).

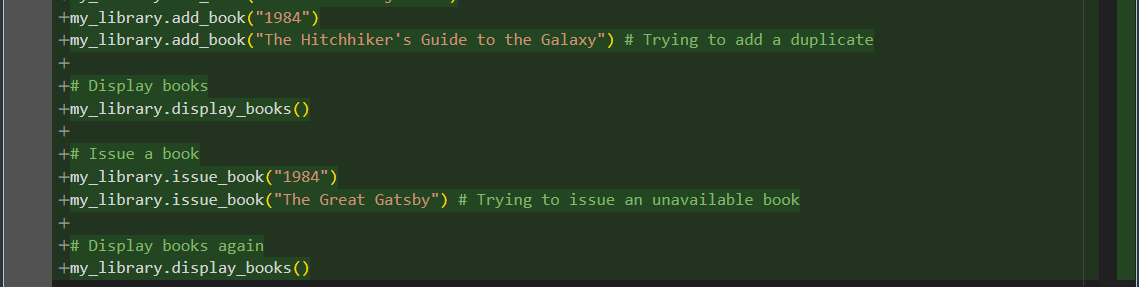
Ask AI to add comments and documentation

**Prompt:**

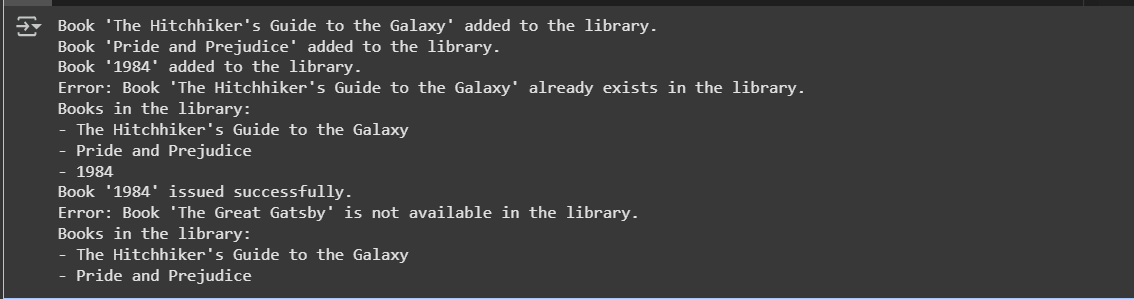
#Create a Python Library class with methods add\_book(), issue\_book(), and display\_books(). Ensure edge cases are handled, like issuing unavailable books. Add comments and docstrings for clarity

**Code:**





**Output:**

****

**Analysis:**

The methods handle most edge cases well: duplicates are prevented using case-insensitive checks in add\_book, and issue\_book gracefully handles unavailable titles with error messages. display\_books covers the empty library scenario. However, issued books ignore case sensitivity, unlike add\_book, which could cause inconsistencies. Overall, edge case handling is strong but improvable.