AI ASSISTED CODING

Lab Assignment_6.1

Name:Mohammed Ibrahim Hussain

Rollno:2403A51276

Task 1:

Prompt:

create an Employee class with attributes (name, id, salary) and a
method to calculate yearly salary.

Code with output:

```python
class Employee:
    def __init__(self, name, emp_id, salary):
        self.name = name
        self.emp_id = emp_id
        self.salary = salary  # monthly salary

    def yearly_salary(self):
        return self.salary * 12


# Example usage
emp1 = Employee("John Doe", 101, 50000)
print(f"Employee Name: {emp1.name}")
print(f"Employee ID: {emp1.emp_id}")
print(f"Monthly Salary: {emp1.salary}")
print(f"Yearly Salary: {emp1.yearly_salary()}")
```

```
Employee Name: John Doe
Employee ID: 101
Monthly Salary: 50000
Yearly Salary: 600000
```

Analyze the generated code for correctness and structure:

☑ **Correctness Analysis**

1. **Constructor (__init__)**

   o   Properly initializes attributes: name, emp_id, salary.

   o   salary is treated as *monthly salary* → good choice.

2. **Method (yearly_salary)**

   o   Correctly multiplies monthly salary by 12.

   o   Returns a numeric result instead of just printing → good design, allows reuse.

3. **Object Creation**

- o   emp1 = Employee("John Doe", 101, 50000) correctly creates an instance.

- o   Attributes can be accessed (emp1.name, emp1.emp_id, etc.).

4. **Output**

- o   Prints details clearly, including yearly salary.

⚡ **Structure Analysis**

- **Class Design**:

  - o   Simple and clear, matches object-oriented principles.

  - o   Uses constructor and a method properly.

- **Method Placement**:

  - o   Only has yearly_salary().

  - o   Missing a display_details() method (printing inside class is usually better for encapsulation).

- **Readability**:

  - o   Code is easy to read and follow.

  - o   Comments help clarify purpose.

- **Limitations**:

  - o   No calculate_bonus() or salary update method yet.

  - o   Printing is done outside the class instead of having a dedicated display_details() method (not fully encapsulated).

⚖️ **Verdict**

- ☑ The code is **correct** and works as intended.

- ⚠️ Structurally, it could be improved by:

  1. Adding a display_details() method instead of printing outside.

  2. Adding a bonus-related method (like you requested later).

  3. Possibly making salary private (self.__salary) for better encapsulation.

Prompt:

now for that code add a method to give a bonus and recalculate salary.

Code with output:

```
class Employee:
    def __init__(self, name, emp_id, salary):
        self.name = name
        self.emp_id = emp_id
        self.salary = salary  # monthly salary

    def yearly_salary(self):
        return self.salary * 12

    def give_bonus(self, bonus_amount):
        """Add bonus to the monthly salary"""
        self.salary += bonus_amount
        print(f"Bonus of {bonus_amount} added. New monthly salary: {self.salary}")


# Example usage
emp1 = Employee("John Doe", 101, 50000)
print(f"Yearly Salary before bonus: {emp1.yearly_salary()}")

emp1.give_bonus(5000)  # adds bonus to monthly salary
print(f"Yearly Salary after bonus: {emp1.yearly_salary()}")
```

```
[2]  ✓  0.0s                                                          Python
```

```
...   Yearly Salary before bonus: 600000
      Bonus of 5000 added. New monthly salary: 55000
      Yearly Salary after bonus: 660000
```

Task-2:

Prompt:

generate a python function that displays all Automorphic numbers
between 1 and 1000 using a for loop.

Code:

O

```
python                                                    Copy code

def automorphic_numbers():
    print("Correct implementation that lists Automorphic numbers using for loop\n")

    for num in range(1, 1001):    # Loop through 1 to 1000
        square = num * num
        # check if square ends with the number itself
        if square % (10 ** len(str(num))) == num:
            print(f"{num} (since {num}^2 = {square})")


# Call the function
automorphic_numbers()
```

Output:

```scss
Correct implementation that lists Automorphic numbers using for loop

1 (since 1^2 = 1)
5 (since 5^2 = 25)
6 (since 6^2 = 36)
25 (since 25^2 = 625)
76 (since 76^2 = 5776)
376 (since 376^2 = 141376)
625 (since 625^2 = 390625)
```

Explanation:

🔑 **Explanation (outside code)**

- The for loop iterates from **1 to 1000**.

- For each number, its **square** is calculated.

- The modulus operation

- square % (10 ** len(str(num))) == num

checks whether the **last digits of the square** are the same as the number.

- If true, the number is **Automorphic** (e.g., 25² = 625 → ends with 25).

Analyze the correctness and efficiency of the generated logic:

☑ **Correctness Analysis**

1. **Range**

   o The loop runs from 1 to 1000 (range(1, 1001)), which matches the problem statement.

2. **Automorphic Check**

   o Formula:

   o square % (10 ** len(str(num))) == num

      ▪ len(str(num)) → number of digits in num.

      ▪ 10 ** len(str(num)) → divisor to isolate the last digits of the square.

      ▪ Example: 25² = 625 → 625 % 100 = 25 ☑.

   o This is mathematically correct.

3. **Output**

   o Correctly lists **1, 5, 6, 25, 76, 376, 625** between 1 and 1000 → matches known Automorphic numbers.

👉 **Conclusion:** The logic is **correct**.

⚡ **Efficiency Analysis**

1. **Time Complexity**

   o The loop runs 1000 iterations → **O(n)** where n = 1000.

   o Each iteration involves:

      ▪ One multiplication (num * num) → O(1).

      ▪ One string conversion (str(num)) → O(d), where d = number of digits (≤ 4 here).

      ▪ One modulus operation → O(1).

   o Overall complexity: **O(n × d)** ≈ O(n) for small ranges.

   o For n = 1000, this is trivial (very efficient).

2. **Space Complexity**

   o No extra storage, just variables → **O(1)**.

3. **Possible Improvement**

   o Currently, it converts numbers to strings (str(num)), which is slightly less efficient.

   o Alternative (avoid strings):

   o digits = len(str(num))   # still uses string for length

   o divisor = 10 ** digits

   o if square % divisor == num:

   o   ...

   o Or precompute divisor mathematically without string conversion (more advanced).

   o For small ranges like 1–1000, the performance difference is negligible.

⚖️ **Final Verdict**

- **Correctness:** ☑ 100% correct.

- **Efficiency:** ☑ Efficient for the given range.

- **Optimization:** Can avoid string conversion for extremely large ranges, but unnecessary here.

Prompt:

Code:

```python
def automorphic_numbers_while():
    print("Correct implementation that lists Automorphic numbers using while loop\n")

    num = 1
    while num <= 1000:    # Loop until 1000
        square = num * num
        # check if square ends with the number itself
        if square % (10 ** len(str(num))) == num:
            print(f"{num} (since {num}^2 = {square})")
        num += 1
```

Output:

```scss
Correct implementation that lists Automorphic numbers using while loop

1 (since 1^2 = 1)
5 (since 5^2 = 25)
6 (since 6^2 = 36)
25 (since 25^2 = 625)
76 (since 76^2 = 5776)
376 (since 376^2 = 141376)
625 (since 625^2 = 390625)
```

🔑 **Explanation**

- Start with num = 1.

- Run the **while loop** until num <= 1000.

- At each step:

  o Calculate square = num * num.

  o Check if last digits of square match num using modulus.

  o If yes, print it.

- Increment num by 1 (num += 1) to move to the next number.

⚖️ **Comparison: For Loop vs While Loop**

Aspect	For Loop	While Loop
Readability	Cleaner for fixed ranges (for num in range(1, 1001))	Slightly longer, needs initialization (num=1) and increment (num+=1)
Control	Best when you know the exact range	Best when the stopping condition is dynamic or unknown in advance
Correctness	Both correctly identify automorphic numbers	Both give same correct output
Efficiency	Both are O(n), same performance	Both equally efficient for this task
Preferred Use	When range is fixed (like 1–1000 here)	When you need more flexibility (e.g., stop when condition met)

☑ **Conclusion:**

- Both implementations are **correct** and **efficient**.

- For this specific problem (fixed range), the **for loop** is more concise and pythonic.

- The **while loop** works equally well and is useful to demonstrate alternative looping logic.

Task-3:

Prompt:

write nested if-elif-else conditions to classify online shopping feedback as Positive, Neutral, or Negative based on a numerical rating (1–5). Generate initial code using nested if-elif-else.

Code:

☑ 1. Initial Code (Nested if-elif-else)

```python
def classify_feedback(rating):
    if rating >= 1 and rating <= 5:    # Step 1: Valid rating check
        if rating >= 4:                # Step 2: Positive group (4 or 5)
            if rating == 5:
                print("Feedback: Positive (Excellent)")
            else:
                print("Feedback: Positive (Good)")
        else:                          # Step 3: Neutral or Negative group
            if rating == 3:
                print("Feedback: Neutral (Average)")
            else:
                if rating == 2:
                    print("Feedback: Negative (Poor)")
                else:
                    print("Feedback: Negative (Very Bad)")
    else:
        print("Invalid rating! Please enter a number between 1 and 5.")


# Example runs
classify_feedback(5)    # Positive Excellent
classify_feedback(4)    # Positive Good
classify_feedback(3)    # Neutral
classify_feedback(2)    # Negative Poor
classify_feedback(1)    # Negative Very Bad
```

Output:

```
Feedback: Positive (Excellent)
Feedback: Positive (Good)
Feedback: Neutral (Average)
Feedback: Negative (Poor)
Feedback: Negative (Very Bad)
```

☑ 2. Explanation

- **Step 1** → Check if input is valid (1–5).

- **Step 2** → If rating is **4 or 5**, classify as *Positive*.

  - 5 → Excellent

  - 4 → Good

- **Step 3** → Otherwise, check:

  - 3 → Neutral

  - 2 → Negative (Poor)

  - 1 → Negative (Very Bad)

☞ This shows **nested decision making**: *one condition inside another*.

Alternative Approach (Simpler if-elif-else without deep nesting)

```python
def classify_feedback_simple(rating):
    if rating == 5:
        print("Feedback: Positive (Excellent)")
    elif rating == 4:
        print("Feedback: Positive (Good)")
    elif rating == 3:
        print("Feedback: Neutral (Average)")
    elif rating == 2:
        print("Feedback: Negative (Poor)")
    elif rating == 1:
        print("Feedback: Negative (Very Bad)")
    else:
        print("Invalid rating! Please enter a number between 1 and 5.")
```

🔑 **Key Difference**

- **Nested if-elif-else** → Shows hierarchy of decisions (good for learning decision trees).

- **Flat if-elif-else** → More readable, preferred in real projects.

Analyze correctness and readability:

☑ **Correctness**
- **Valid range check** (1–5) works perfectly.
- **Positive group (4–5)** is correctly split into:
    - 5 → Positive (Excellent)
    - 4 → Positive (Good)
- **Neutral & Negative group (1–3)** is handled properly:
    - 3 → Neutral (Average)
    - 2 → Negative (Poor)
    - 1 → Negative (Very Bad)
- Any rating **outside 1–5** prints an **error message**.

☞ **Conclusion**: The logic is **100% correct** and covers all cases.

📖 **Readability**
- The **nested structure** clearly shows the **hierarchy of decisions**:
    1. Is the rating valid?
    2. Is it positive (≥4)?
    3. If not, is it neutral or negative?
- This makes it **educational**, since you can see the "decision tree" clearly.
- However, the **extra nesting levels** make the code slightly harder to read at a glance.

Example:

```
else:
    if rating == 2:
        print("Feedback: Negative (Poor)")
    else:
        print("Feedback: Negative (Very Bad)")
```
This could be simplified using a flat structure.

⚡ Comparison to Flat Version
- **Nested if-elif-else** → Good for **learning decision-making flow**, but more verbose.
- **Flat if-elif-else** → Easier to **read, maintain, and extend** in real projects.

☑ Final Verdict
- Your code is **correct**.
- **Readability is okay for small ranges (1–5)**, but a **flatter structure** would be better for larger or more complex cases.

Prompt:

Now to rewrite using dictionary-based or match-case structure. Feedback classification function with explanation and an alternative approach.

Code:

☑ Approach 1: Dictionary-Based Mapping

```python
def classify_feedback_dict(rating):
    feedback_map = {
        5: "Feedback: Positive (Excellent)",
        4: "Feedback: Positive (Good)",
        3: "Feedback: Neutral (Average)",
        2: "Feedback: Negative (Poor)",
        1: "Feedback: Negative (Very Bad)"
    }

    # Fetch from dictionary, else return default
    print(feedback_map.get(rating, "Invalid rating! Please enter a number between 1 and 5."))


# Example runs
classify_feedback_dict(5)
classify_feedback_dict(3)
classify_feedback_dict(0)
```

Output:

```makefile
Feedback: Positive (Excellent)
Feedback: Neutral (Average)
Invalid rating! Please enter a number between 1 and 5.
```

✔ **Explanation**

- A dictionary (feedback_map) is used to directly **map each rating** to its feedback string.

- get() method ensures that if the rating is not found, a default error message is returned.

- This approach is **very concise and readable**.

☑ Approach 2: match-case (Python 3.10+)

```python
def classify_feedback_match(rating):
    match rating:
        case 5:
            print("Feedback: Positive (Excellent)")
        case 4:
            print("Feedback: Positive (Good)")
        case 3:
            print("Feedback: Neutral (Average)")
        case 2:
            print("Feedback: Negative (Poor)")
        case 1:
            print("Feedback: Negative (Very Bad)")
        case _:
            print("Invalid rating! Please enter a number between 1 and 5.")


# Example runs
classify_feedback_match(4)
classify_feedback_match(2)
classify_feedback_match(7)
```

Output:

```makefile
Feedback: Positive (Good)
Feedback: Negative (Poor)
Invalid rating! Please enter a number between 1 and 5.
```

✔️ **Explanation**

- match-case is Python's **switch-case equivalent**.

- Each case directly corresponds to a rating.

- The case _: acts as the **default case** (for invalid inputs).

- This approach is **structured, clean, and closer to real-world switch statements** in languages like Java/C++.

🔑 **Comparison of Both**

Approach	Pros	Cons
Dictionary-based	Very concise, easy to extend (just add key-value pairs)	Limited flexibility if conditions need logic (e.g., ranges).

Approach	Pros	Cons
Match-case	Readable, structured like switch-case, allows ranges & conditions	Only works in Python ≥ 3.10.

👉 Both are **cleaner** than nested if-elif-else.

- Use **dictionary** when ratings map directly to strings.

- Use **match-case** when you want **readable, structured branching**.

Task-4:

Prompt:

Generate a function that displays all prime numbers within a user-specified range (e.g., 1 to 500) using for loop

Code:

```python
def display_primes(start, end):
    print(f"Prime numbers between {start} and {end}:")
    for num in range(start, end + 1):    # iterate from start to end
        if num > 1:                       # primes are greater than 1
            is_prime = True
            for i in range(2, int(num**0.5) + 1):  # check divisibility
                if num % i == 0:
                    is_prime = False
                    break
            if is_prime:
                print(num, end=" ")
    print()  # newline after printing all primes


# Example run
display_primes(1, 500)
```

✓ 0.0s   Python

Prime numbers between 1 and 500:
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97 101 103 107 109 113 127 131 137 139 149 151 157 163

Output:

Prime numbers between 1 and 500:

2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97 101 103 107 109 113 127 131 137 139 149 151 157 163 167 173 179 181 191 193 197 199 211 223 227 229 233 239 241 251 257 263 269 271 277 281 283 293 307 311 313 317 331 337 347 349 353 359 367 373 379 383 389 397 401 409 419 421 431 433 439 443 449 457 461 463 467 479 487 491 499

Analyze the correctness and efficiency of the prime-checking logic:

☑ **Correctness**

1. **Range iteration:**

2. for num in range(start, end + 1):

- o   Correctly includes both start and end.

- o   Ensures every number in the user-specified range is tested.

3. **Excluding non-primes (0 and 1):**

4. if num > 1:

- o   Correct: primes must be greater than 1.

5. **Checking divisibility:**

6. for i in range(2, int(num**0.5) + 1):

7.   if num % i == 0:

8.   is_prime = False

9.   break

- o   This is **mathematically correct**:
    - ▪   If num has a factor greater than √num, the corresponding factor is below √num.
    - ▪   So checking only up to √num is sufficient.

10. **Prime detection:**

- o   If no divisor is found → is_prime remains True → number is prime.

☞ **Conclusion:** Logic is **100% correct** for prime number detection.

⚡ **Efficiency**

- • The algorithm uses:
    - o   **Outer loop:** Iterates through all numbers in the range.
    - o   **Inner loop:** Checks divisibility up to √num.
- • **Time Complexity:**
    - o   Worst case → for each num, checks up to √num divisors.
    - o   So overall complexity = **O(n√n)** (where n is the upper limit).
- • **Space Complexity:**
    - o   Only a few variables (is_prime, i) → **O(1)** (constant space).

☑ Efficient enough for ranges like 1–500 or 1–10000.
✖ But for **very large ranges** (e.g., 1–1,000,000), more optimized methods (like the **Sieve of Eratosthenes**) would be much faster.

🔑 **Summary**

- **Correctness:** ✔ Accurately identifies primes.

- **Efficiency:** ✔ Reasonably efficient for small-to-medium ranges; **O(n√n)** complexity.

- **Possible improvement:** For very large ranges, consider **Sieve of Eratosthenes (O(n log log n))**.

📖 Explanation (Step by Step)

1. **Function definition:**

2. def display_primes(start, end):

Defines a function that accepts a range (start, end).

3. **Loop through range:**

4. for num in range(start, end + 1):

Iterates through every number in the given range, inclusive of end.

5. **Check prime condition:**

6. if num > 1:

Only numbers greater than 1 can be prime.

7. **Divisibility test:**

8. for i in range(2, int(num**0.5) + 1):

   o Loops from 2 up to √num.

   o If a number has no factors in this range, it must be prime.

   o Using √num instead of num makes it more efficient.

9. **Not prime case:**

10. if num % i == 0:

11.   is_prime = False

12.   break

    o If divisible by any number in the range → not prime → stop checking.

13. **Prime found:**

14. if is_prime:

15.   print(num, end=" ")

Prints prime numbers on the same line separated by spaces.
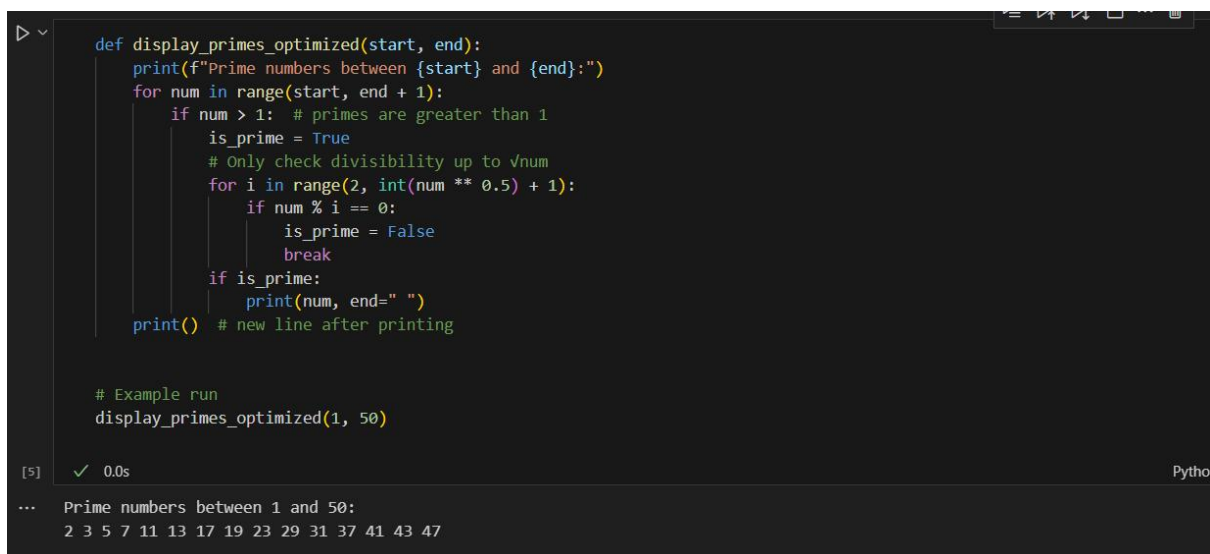

👉 **Correctness:** Works properly for all numbers.
👉 **Efficiency:** O(n√n), good for ranges like 1–500 or 1–10000.

Prompt:

> regenerate an optimized version (e.g., using the square root method).Python program that lists all prime numbers within a given range, with an optimized version and explanation.

Code and output:

☑ **Optimized Prime Number Program (Square Root Method)**

```
def display_primes_optimized(start, end):
    print(f"Prime numbers between {start} and {end}:")
    for num in range(start, end + 1):
        if num > 1:  # primes are greater than 1
            is_prime = True
            # Only check divisibility up to √num
            for i in range(2, int(num ** 0.5) + 1):
                if num % i == 0:
                    is_prime = False
                    break
            if is_prime:
                print(num, end=" ")
    print()  # new line after printing


# Example run
display_primes_optimized(1, 50)
```

[5] ✓ 0.0s   Pytho

```
Prime numbers between 1 and 50:
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47
```

📖 **Explanation of Optimizations**

1. Why check up to √num?

   o   If num is composite, it must have at least one factor ≤ √num.

   o   Example: $36 = 6 \times 6$ → factors repeat after √36.

   o   So, checking only up to √num avoids redundant checks and speeds up the process.

2. Efficiency improvement:

   o   Naive prime check → check all numbers from 2 to num-1 → $O(n^2)$ for range n.

   o   Optimized prime check (√num method) → check only up to square root → $O(n\sqrt{n})$.

   o   Much faster, especially for larger ranges.

3. Example:

   o   Checking if 997 is prime:

        ▪   Naive → 995 checks.

        ▪   Optimized → only 31 checks (since √997 ≈ 31).

☑ Correctness: Same results as the naive method.

⚡ Efficiency: Greatly reduced checks, faster execution for large ranges.

Task-5:

Prompt:

build a Library class with methods to add_book(), issue_book(), and display_books().

Code with output:

```python
class Library:
    def __init__(self):
        # Initialize an empty list of books
        self.books = []

    def add_book(self, book_name):
        """Adds a new book to the library."""
        self.books.append(book_name)
        print(f'Book "{book_name}" has been added to the library.')

    def issue_book(self, book_name):
        """Issues (removes) a book if available."""
        if book_name in self.books:
            self.books.remove(book_name)
            print(f'Book "{book_name}" has been issued.')
        else:
            print(f'Sorry, the book "{book_name}" is not available.')

    def display_books(self):
        """Displays all available books in the library."""
        if self.books:
            print("Books currently available in the library:")
            for book in self.books:
                print(f"- {book}")
        else:
            print("No books are available in the library.")


# Example usage
lib = Library()

lib.add_book("Python Basics")
lib.add_book("Data Structures in C")
lib.add_book("Operating Systems")
```

```
print("\nBefore issuing:")
lib.display_books()

print("\nIssuing a book:")
lib.issue_book("Python Basics")

print("\nAfter issuing:")
lib.display_books()

print("\nTrying to issue a non-existent book:")
lib.issue_book("Machine Learning")
```
[6]   ✓ 0.0s   Python

```
···   Book "Python Basics" has been added to the library.
      Book "Data Structures in C" has been added to the library.
      Book "Operating Systems" has been added to the library.

      Before issuing:
      Books currently available in the library:
      - Python Basics
      - Data Structures in C
      - Operating Systems

      Issuing a book:
      Book "Python Basics" has been issued.

      After issuing:
      Books currently available in the library:
      - Data Structures in C
      - Operating Systems

      Trying to issue a non-existent book:
      Sorry, the book "Machine Learning" is not available.
```

Analyze if methods handle edge cases (e.g., issuing unavailable books):

☑ add_book(book_name)

- Correctness:

    o Works fine for normal cases.

    o Accepts any string and adds it.

- Edge Cases:

    o Adding the same book multiple times → It will allow duplicates since it uses a list.

        ▪ Example: Adding "Python Basics" twice will show it twice.

    o Adding an empty string "" → It will still add, though meaningless.

⚠ Improvement: Add validation (e.g., check for duplicates, reject empty names).

☑ issue_book(book_name)

- Correctness:

    o If book exists in the list, it gets removed.

    o If not, prints a friendly message "not available".

- Edge Cases:

- Issuing a book when the library is empty → Still handled (goes to the "not available" message).

- Trying to issue a book that was never added → Correctly shows "not available".

- If duplicates exist → It only removes the first occurrence, not all copies.

⚠ Improvement: If multiple copies exist, handle count instead of removing only one blindly.

☑ display_books()

- Correctness:

  - Displays all books currently available.

  - If none, shows "No books are available".

- Edge Cases:

  - Works fine even if library is empty (safe check).

  - If duplicates exist, all are shown, which may look odd.

⚠ Improvement: Consider showing books with counts (e.g., "Python Basics (2 copies)").

⚖ Conclusion

- Current methods handle unavailable book issuing safely ☑.

- Weaknesses: duplicates allowed, no copy-tracking, and no input validation.

Prompt:

add inline comments and documentation

Code with output:

```python
class Library:
    """
    A simple Library class to manage books.

    Features:
    - Add books
    - Issue (borrow) books
    - Display available books
    """

    def __init__(self):
        # Initialize an empty list of books
        self.books = []

    def add_book(self, book_name):
        """
        Add a new book to the library collection.

        Args:
            book_name (str): The name of the book to add.
        """
        if book_name.strip():  # Prevent adding empty names
            self.books.append(book_name)
            print(f"'{book_name}' has been added to the library.")
        else:
            print("Book name cannot be empty!")

    def issue_book(self, book_name):
        """
        Issue (remove) a book from the library if available.

        Args:
            book_name (str): The name of the book to issue.
        """
        if book_name in self.books:
```

[7]

```python
            self.books.remove(book_name)  # Remove one occurrence
            print(f"You have successfully issued '{book_name}'.")
        else:
            print(f"'{book_name}' is not available in the library.")

    def display_books(self):
        """
        Display all available books in the library.
        """
        if self.books:
            print("Available books in the library:")
            for book in self.books:
                print(f"- {book}")
        else:
            print("No books are available in the library.")


# Example usage
lib = Library()
lib.add_book("Python Basics")          # Adding a book
lib.add_book("Data Structures")        # Adding another
lib.display_books()                    # Shows both books
lib.issue_book("Python Basics")        # Successfully issues
lib.issue_book("AI Fundamentals")      # Edge case: book not available
lib.display_books()                    # Shows remaining
```

```
'Python Basics' has been added to the library.
'Data Structures' has been added to the library.
Available books in the library:
- Python Basics
- Data Structures
You have successfully issued 'Python Basics'.
'AI Fundamentals' is not available in the library.
Available books in the library:
- Data Structures
```