

Name: K.Likitha

Hall Ticket: 2303A52144

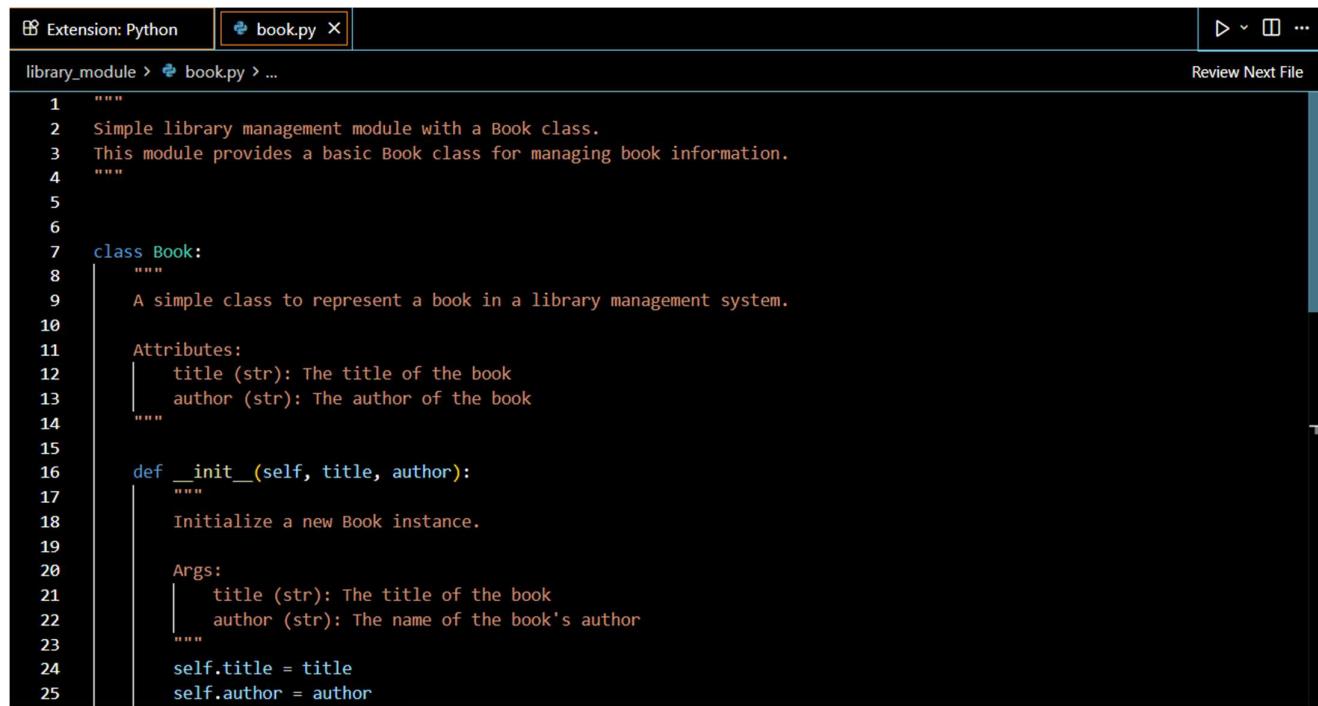
Batch: 41

Task 1: Book Class Generation

Prompt:

Create a simple Python Book class with attributes title and author and a summary() method. Write a small main section that creates a Book object and prints the output of the summary() method. Also include a short comment explaining the code quality and readability in easy language.

Code:



The screenshot shows a code editor window with a dark theme. The file tab at the top says "book.py". The code itself is a Python module named "library_module". It defines a "Book" class with attributes "title" and "author". The code includes docstrings and type hints for readability.

```
Extension: Python | book.py | Review Next File
library_module > book.py > ...
1 """
2 Simple library management module with a Book class.
3 This module provides a basic Book class for managing book information.
4 """
5
6
7 class Book:
8     """
9         A simple class to represent a book in a library management system.
10    Attributes:
11        title (str): The title of the book
12        author (str): The author of the book
13    """
14
15    def __init__(self, title, author):
16        """
17            Initialize a new Book instance.
18
19        Args:
20            title (str): The title of the book
21            author (str): The name of the book's author
22        """
23        self.title = title
24        self.author = author
```

The screenshot shows the Visual Studio Code interface. The top bar has tabs for 'Extension: Python' and 'book.py'. The main area is the code editor with the following content:

```
library_module > book.py > ...
Review Next File

7 class Book:
26
27     def summary(self):
28         """
29             Generate a readable summary sentence describing the book.
30
31         Returns:
32             str: A sentence describing the book with its title and author
33
34         return f"{self.title} by {self.author}"
35
36
37 # Main section - creates a Book object and prints the summary
38 if __name__ == "__main__":
39     # Create a book instance
40     my_book = Book("The Great Gatsby", "F. Scott Fitzgerald")
41
42     # Display the book summary
43     print(my_book.summary())
44
45
46
```

Output:

The screenshot shows the VS Code terminal tab. The terminal output is:

```
Problems Output Debug Console Terminal Ports + ... ^ x
PS C:\Program Files\cursor> & C:\Users\lilac\AppData\Local\Programs\Python\Python313\python.exe /library_module/book.py
The Great Gatsby by F. Scott Fitzgerald
PS C:\Program Files\cursor> [ ]
```

Explanation:

This program defines a Book class with attributes for the book's title and author. The summary() method returns a short description of the book using these attributes. In the main part of the program, a Book object is created and the summary is printed, showing how the class works.

Task 2: Sorting Dictionaries with AI

Prompt:

Generate Python code to sort a list of user records represented as dictionaries using the age key. Include sample data, perform the sorting, and print the sorted output. Also provide a short explanation of how the code works.

Code:

Cursor AI:

```

sort_users_by_age.py •
sort_users_by_age.py > ...

1 """
2 Sort a list of user records by age.
3
4 This script demonstrates how to sort a list of dictionaries containing
5 user information using the 'age' key.
6 """
7
8 # Sample data: List of user records as dictionaries
9 user_records = [
10     {"name": "Alice", "age": 28, "city": "New York"},
11     {"name": "Bob", "age": 22, "city": "Los Angeles"},
12     {"name": "Charlie", "age": 35, "city": "Chicago"},
13     {"name": "Diana", "age": 19, "city": "Houston"},
14     {"name": "Eve", "age": 31, "city": "Phoenix"},
15     {"name": "Frank", "age": 25, "city": "Philadelphia"}
16 ]
17
18 print("Original list of users:")
19 print("-" * 50)
20 for user in user_records:
21     | | print(f"{user['name'][:10]} | Age: {user['age'][:2]} | City: {user['city']}")"
22 print()
23
24 # Sort the list by age (ascending order)
25 # The key parameter specifies a function that extracts the value to sort by

```

```

sort_users_by_age.py •
sort_users_by_age.py > ...

26 sorted_users = sorted(user_records, key=lambda user: user['age'])
27
28 print("Sorted by age (ascending):")
29 print("-" * 50)
30 for user in sorted_users:
31     | | print(f"{user['name'][:10]} | Age: {user['age'][:2]} | City: {user['city']}")"
32 print()
33
34 # Alternative: Sort in descending order (oldest first)
35 sorted_users_desc = sorted(user_records, key=lambda user: user['age'], reverse=True)
36
37 print("Sorted by age (descending):")
38 print("-" * 50)
39 for user in sorted_users_desc:
40     | | print(f"{user['name'][:10]} | Age: {user['age'][:2]} | City: {user['city']}")"
41 Ctrl+L to chat, Ctrl+K to generate
42

```

Output(Cursor AI):

```

Problems Output Debug Console Terminal Ports
PS C:\Users\lilac\OneDrive\Desktop\AI-Assisted-Coding> & C:\Users\lilac\AppData\Local\Programs\Python\Python313\python.exe c:/Users/lilac
/OneDrive/Desktop/AI-Assisted-Coding/sort_users_by_age.py
● Original list of users:
-----
Alice | Age: 28 | City: New York
Bob | Age: 22 | City: Los Angeles
Charlie | Age: 35 | City: Chicago
Diana | Age: 19 | City: Houston
Eve | Age: 31 | City: Phoenix
Frank | Age: 25 | City: Philadelphia

Sorted by age (ascending):
-----
Diana | Age: 19 | City: Houston
Bob | Age: 22 | City: Los Angeles
Frank | Age: 25 | City: Philadelphia
Alice | Age: 28 | City: New York
Eve | Age: 31 | City: Phoenix
Charlie | Age: 35 | City: Chicago

Sorted by age (descending):
-----
Charlie | Age: 35 | City: Chicago
Eve | Age: 31 | City: Phoenix
Alice | Age: 28 | City: New York
Frank | Age: 25 | City: Philadelphia
Bob | Age: 22 | City: Los Angeles
Diana | Age: 19 | City: Houston

```

Gemini AI:

The screenshot shows a Jupyter Notebook interface with a dark theme. A code cell is open, containing the following Python code:

```
# Sample user records
user_records = [
    {"name": "Alice", "age": 30, "city": "New York"},
    {"name": "Bob", "age": 24, "city": "Los Angeles"},
    {"name": "Charlie", "age": 35, "city": "Chicago"},
    {"name": "David", "age": 28, "city": "Houston"}
]

print("Original records:")
for record in user_records:
    print(record)

# Sort the user records by 'age'
sorted_records = sorted(user_records, key=lambda user: user['age'])

print("\nSorted records by age:")
for record in sorted_records:
    print(record)
```

Output:

```
... Original records:
{'name': 'Alice', 'age': 30, 'city': 'New York'}
{'name': 'Bob', 'age': 24, 'city': 'Los Angeles'}
{'name': 'Charlie', 'age': 35, 'city': 'Chicago'}
{'name': 'David', 'age': 28, 'city': 'Houston'}

Sorted records by age:
{'name': 'Bob', 'age': 24, 'city': 'Los Angeles'}
{'name': 'David', 'age': 28, 'city': 'Houston'}
{'name': 'Alice', 'age': 30, 'city': 'New York'}
{'name': 'Charlie', 'age': 35, 'city': 'Chicago'}
```

Explanations:

Comparison: Gemini Output vs Cursor AI Output (Sorting Dictionaries by Age)

Gemini AI Output – Explanation

The Gemini AI output demonstrates a basic and clear approach to sorting a list of dictionaries by the age key. It first prints the original records and then displays the sorted records in ascending order of age. The code is concise and easy to understand, making it suitable for beginners. However, the output format is simple and mainly focuses on showing the sorted dictionaries without additional formatting.

Cursor AI Output – Explanation

The Cursor AI output provides a more detailed and structured solution. It prints the original list of users in a well-formatted manner and then shows the sorted results in both ascending and descending order. The output is more readable due to clear labels, separators, and aligned fields such as name, age, and city. This makes the program more user-friendly and suitable for real-world use.

Overall Comparison

Both Gemini AI and Cursor AI correctly sort the user records by age and have similar performance since they use Python's built-in sorting methods. Gemini AI focuses on simplicity and minimal code, while Cursor AI emphasizes clarity, better output formatting, and additional functionality. Cursor AI's output is more polished, whereas Gemini AI's output is easier for quick understanding and learning.

Task 3: Calculator Using Functions

Prompt:

Generate a Python calculator program in a single code cell using functions. The program should display a menu showing Addition, Subtraction, Division, Multiplication, and Square Root as options numbered from 1 to 5. All inputs must be taken from the user at runtime using `input()`. After displaying the menu, the program should ask the user to choose an operation. If the user selects options 1 to 4, the program should ask how many numbers the user wants to enter, read those numbers from the user, perform the selected operation using a separate function, and display the

final result. If the user selects option 5, the program should ask for a single number, calculate its square root using a function, and print the result. The program should be simple, clear, and beginner-friendly.

Code:

```
Assignment_2_4.ipynb
```

```
import math

def add_numbers(numbers):
    return sum(numbers)

def subtract_numbers(numbers):
    result = numbers[0]
    for i in range(1, len(numbers)):
        result -= numbers[i]
    return result

def multiply_numbers(numbers):
    result = 1
    for num in numbers:
        result *= num
    return result

def divide_numbers(numbers):
    if len(numbers) < 2:
        return "Error: Division requires at least two numbers."
    result = numbers[0]
    for i in range(1, len(numbers)):
        if numbers[i] == 0:
            return "Error: Cannot divide by zero!"
        result /= numbers[i]
    return result

def square_root(number):
    if number < 0:
        return "Error: Cannot calculate square root of a negative number."
    return math.sqrt(number)

def get_numbers(count):
    numbers = []
    for i in range(count):
        num = float(input(f"Enter number {i + 1}: "))
        numbers.append(num)
    return numbers
```

```
Assignment_2_4.ipynb
```

```
while True:
    try:
        num = float(input("Enter number (i + 1): "))
        numbers.append(num)
        break
    except ValueError:
        print("Invalid input. Please enter a valid number.")

def calculator():
    print("\n--- Python Calculator ---")
    print("1. Addition")
    print("2. Subtraction")
    print("3. Multiplication")
    print("4. Division")
    print("5. Square Root")
    print("-----")

    while True:
        choice = int(input("Enter your choice (1-5): "))
        if 1 <= choice <= 5:
            break
        else:
            print("Invalid choice. Please enter a number between 1 and 5.")
    except ValueError:
        print("Invalid input. Please enter a number.")

    if choice in [1, 2, 3, 4]:
        while True:
            num_count = int(input("How many numbers do you want to enter? "))
            if num_count > 0:
                break
            else:
```

A screenshot of a Jupyter Notebook interface. The title bar says "Assignment_2.4.ipynb". The code cell contains a Python script for a calculator. It includes functions for addition, subtraction, multiplication, division, and square root calculation. It handles user input for operations and numbers, and provides error handling for invalid inputs.

```
print("Please enter at least one number. ")
except ValueError:
    print("Invalid input. Please enter a valid integer.")

numbers = get_numbers(num_count)

if choice == 1:
    result = add_numbers(numbers)
    operation = "sum"
elif choice == 2:
    result = subtract_numbers(numbers)
    operation = "difference"
elif choice == 3:
    result = multiply_numbers(numbers)
    operation = "product"
elif choice == 4:
    result = divide_numbers(numbers)
    operation = "quotient"

print(f"The {operation} is: {result}")

elif choice == 5:
    while True:
        try:
            num = float(input("Enter the number for square root: "))
            break
        except ValueError:
            print("Invalid input. Please enter a valid number.")
    result = square_root(num)
    print(f"The square root of {num} is: {result}")

# Run the calculator program
calculator()
```

Output:

A screenshot of the Jupyter Notebook output area. It shows the program's menu options (Addition, Subtraction, Multiplication, Division, Square Root), a user entering '5' for the square root option, and the program outputting the square root of 49 as 7.0.

```
--- Python Calculator ---
1. Addition
2. Subtraction
3. Multiplication
4. Division
5. Square Root
-----
Enter your choice (1-5): 5
Enter the number for square root: 49
the square root of 49.0 is: 7.0
```

Task 4: Armstrong Number Optimization

Gemini Prompt:

Generate a Python program to check whether a given number is an Armstrong number. The program should take the number from the user using input()

Gemini Code:

A screenshot of a Jupyter Notebook cell containing a Python script named "check_armstrong". The script defines a function to check if a number is Armstrong, calculates the sum of digits raised to the power of digit count, and compares it with the original number.

```
# Function to check if a number is an Armstrong number
def check_armstrong():
    while True:
        try:
            num_str = input("Enter a number: ")
            num = int(num_str)
            if num < 0:
                print("Please enter a non-negative number.")
                continue
            break
        except ValueError:
            print("Invalid input. Please enter an integer.")

    # Calculate the number of digits
    num_digits = len(num_str)

    sum_of_powers = 0
    temp_num = num

    # Calculate the sum of each digit raised to the power of num_digits
    while temp_num > 0:
        digit = temp_num % 10
        sum_of_powers += digit ** num_digits
        temp_num //= 10

    # Compare the sum with the original number
    if num == sum_of_powers:
        print(f"{num} is an Armstrong number.")
    else:
        print(f"{num} is not an Armstrong number.")

# Run the Armstrong number checker
check_armstrong()
```

Gemini Output:

```
Enter a number: 5
5 is an Armstrong number.
```

Cursor AI Prompt:

Improve the existing Python program for checking an Armstrong number by optimizing its performance and readability. Avoid unnecessary recalculations, store the number of digits efficiently, and use a cleaner loop structure. Ensure the program still takes input from the user using `input()` and produces the same correct output, but with better efficiency, cleaner logic, and improved code structure.

Cursor AI: Code

The screenshot shows a code editor with a sidebar for 'AI-Assisted Coding'. The main area contains the original Python script 'sort_users_by_age.py'. The sidebar displays an 'Armstrong number program optimization' section with the following text and code snippets:

Now - +7 -14 · Auto

Improve the existing Python program for checking an Armstrong number by optimizing its performance and readability. Avoid unnecessary recalculations, store the number of digits efficiently, and use a cleaner loop structure.

No linter errors found in c:\Users\lilac\OneDrive\Desktop\AI-Assisted-Coding\sort_users_by_age.py

Optimized Armstrong checker: precomputes digit powers once, uses a clearer sum comprehension over the input string, and keeps user input/validation intact for the same behavior with better readability and efficiency.

```
def check_armstrong():
    """Prompt the user and report whether the number is an Armstrong number."""
    num_str = input("Enter a number: ")
    num = int(num_str)
    if num < 0:
        print("Please enter a non-negative number.")
        continue
    try:
        num_digits = len(num_str)

        # Precompute powers once to avoid repeating digit ** num_digits
        digit_powers = [d ** num_digits for d in range(10)]
        sum_of_powers = sum(digit_powers[int(ch)] for ch in num_str) if num_str else 0

        if num == sum_of_powers:
            print(f"{num} is an Armstrong number.")
        else:
            print(f"{num} is not an Armstrong number.")
    except ValueError:
        print("Invalid input. Please enter an integer.")

check_armstrong()
```

Next step: run the script and test a few values (e.g., 0, 1, 153, 9474, and a non-Armstrong like 123) to confirm output.

View Plans

Search Agents...

New Agent

Agents

Armstrong number pr... Now

Python user record sort... 8h

Cursor AI Output:

The screenshot shows a terminal window with the following output:

```
\python.exe c:/Users/lilac/OneDrive/Desktop/AI-Assisted-Coding/sort_users_by_age.py
Invalid input. Please enter an integer.
Enter a number: 153
153 is an Armstrong number.
PS C:\Users\lilac\OneDrive\Desktop\AI-Assisted-Coding>
```

Explanations:

The Armstrong number program generated using Google Gemini was basic and functional but lacked efficiency and strong input validation. Cursor AI optimized the program by reducing redundant calculations, handling invalid and

negative inputs effectively, and improving overall code structure. The optimized version is more efficient, readable, and robust while maintaining the same correct output.