

ASSIGNMENT – 2.4

Name: R.Shashideepika

H.no:2303A51871

B-13

Task 1: Book Class Generation

❖ Scenario:

You are building a simple library management module.

❖ Task:

Use Cursor AI to generate a Python class Book with attributes title, author, and a summary() method.

❖ Expected Output:

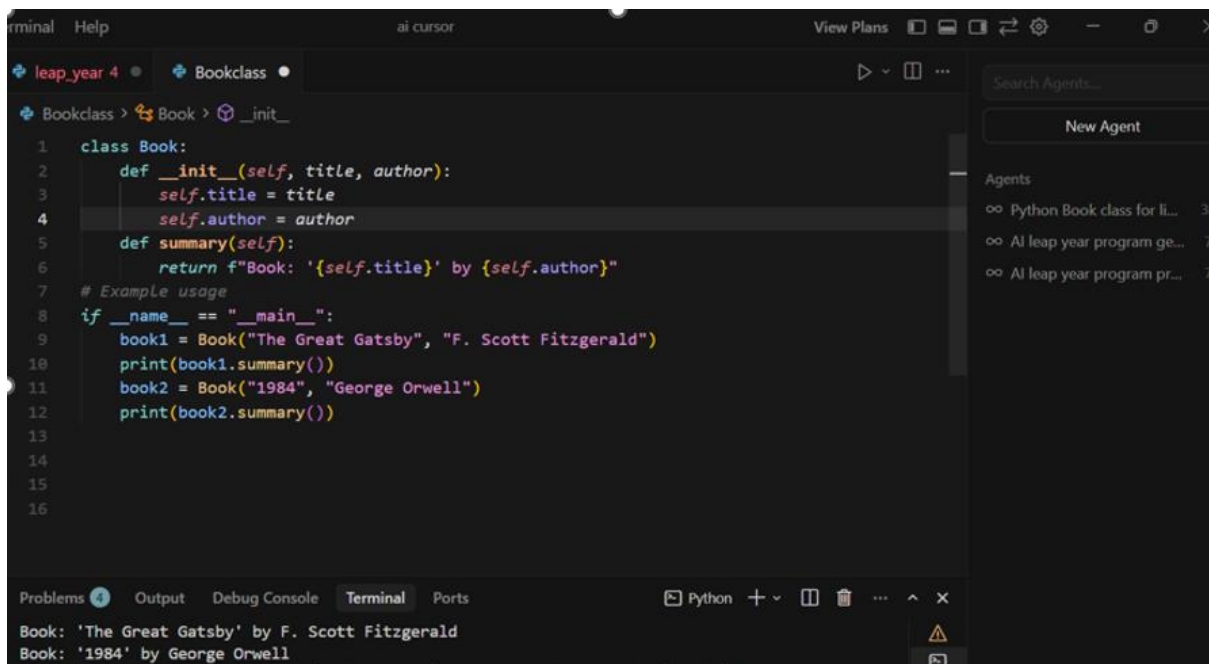
➤ Generated class

➤ Student commentary on code quality

PROMPT:

Generate a Python class Book with attributes title, author, and a summary() method.

CODE:



```
1 class Book:
2     def __init__(self, title, author):
3         self.title = title
4         self.author = author
5     def summary(self):
6         return f"Book: '{self.title}' by {self.author}"
7 # Example usage
8 if __name__ == "__main__":
9     book1 = Book("The Great Gatsby", "F. Scott Fitzgerald")
10    print(book1.summary())
11    book2 = Book("1984", "George Orwell")
12    print(book2.summary())
13
14
15
16
```

Book: 'The Great Gatsby' by F. Scott Fitzgerald
Book: '1984' by George Orwell

JUSTIFICATION:

This program is designed to store book details such as the title and author as a single unit using a class. The class allows the creation of multiple book objects without rewriting the same code, making the program reusable and organized. The summary function displays the book information in a clear and consistent format. The main block executes the program and demonstrates the output by creating and displaying details of different books.

Task 2: Sorting Dictionaries with AI

❖ Scenario:

You need to sort user records by age.

❖ Task:

Use Gemini and Cursor AI to generate code that sorts a list of dictionaries by a key.

❖ Expected Output:

- Both AI outputs
- Comparison of clarity and performance

PROMPT:

Generate code that sorts a list of dictionaries by a key.

CODE FROM GEMINI:

The screenshot shows a Jupyter Notebook titled 'Untitled30.ipynb' with a menu bar (File, Edit, View, Insert, Runtime, Tools, Help) and a toolbar (Commands, Code, Text, Run all). The code in the notebook is as follows:

```
[2] In [ ]: import operator
import timeit

# Sample data
users = [
    {'name': 'Alice', 'age': 30},
    {'name': 'Bob', 'age': 25},
    {'name': 'Charlie', 'age': 35},
    {'name': 'David', 'age': 25}
]
print("Original list:", users)

# --- AI Output 1: Sorting using a lambda function ---
print("\n--- Method 1: Sorting with lambda function ---")
sorted_users_lambda = sorted(users, key=lambda user: user['age'])
print("Sorted by age (lambda):", sorted_users_lambda)

# --- AI Output 2: Sorting using operator.itemgetter ---
print("\n--- Method 2: Sorting with operator.itemgetter ---")
sorted_users_itemgetter = sorted(users, key=operator.itemgetter('age'))
print("Sorted by age (itemgetter):", sorted_users_itemgetter)

# --- Comparison of Clarity and Performance ---
print("\n--- Comparison: Clarity and Performance ---")

# Clarity:
print("\nClarity:")
print("- Lambda Function: Generally very clear for simple key extraction. Readability might decrease with more complex key lookups.")
print("- operator.itemgetter: Often considered more explicit and sometimes more readable for dictionary key lookups.")

# Performance:
print("\nPerformance Benchmarking:")

# Setup for performance comparison
setup_code = """
import operator

```

The Gemini sidebar on the right displays the following information:

- Task 2: Sorting Dictionaries with AI** Scenario: You need to sort user records by age. Task: Use Gemini generate code that sorts a list of dictionaries by a key. Expected Output: > Both AI outputs > Comparison of clarity and performance
- Empty cell**
- Here's the Python code to sort a list of dictionaries by the 'age' key, using two different methods, along with a comparison of their clarity and performance as requested. All the code is in the selected cell.
- The code successfully sorted the list of users by age using both a lambda function and operator.itemgetter. Both methods produced the same sorted list: [{'name': 'Bob', 'age': 25}, {'name': 'David', 'age': 25}, {'name': 'Alice', 'age': 30}, {'name': 'Charlie', 'age': 35}].
- Regarding clarity, lambda functions are generally clear for simple key extraction, while operator.itemgetter is often considered more explicit for dictionary key lookups. In terms of performance, operator.itemgetter was faster than the lambda function in this benchmark (0.220293 seconds vs. 0.389533 seconds), especially for larger datasets, due to its C implementation.
- Buttons: Explain the performance difference between lambda and itemgetter in more detail, Sort the users by name in descending order, Show an example of sorting by multiple keys.
- What can I help you build? Gemini 2.5 Flash
- Gemini can make mistakes so double-check it and use code with caution. Learn more

The screenshot shows the continuation of the Jupyter Notebook. The code in the notebook is as follows:

```
... ] * 1000 # Increase data size for meaningful benchmark
"""

# Benchmark lambda function
time_lambda = timeit.timeit("sorted(users, key=lambda user: user['age'])", setup=setup_code, number=1000)
print(f"Time taken by lambda function: {time_lambda:.6f} seconds")

# Benchmark operator.itemgetter
time_itemgetter = timeit.timeit("sorted(users, key=operator.itemgetter('age'))", setup=setup_code, number=1000)
print(f"Time taken by operator.itemgetter: {time_itemgetter:.6f} seconds")

print("\nConclusion on Performance:")
print("- operator.itemgetter is generally slightly faster than a 'lambda' function, especially for large datasets or in performance-critical sections of code.")

# --- Comparison: Clarity and Performance ---

# Clarity:
print("\nClarity:")
print("- Lambda Function: Generally very clear for simple key extraction. Readability might decrease with more complex key lookups.")
print("- operator.itemgetter: Often considered more explicit and sometimes more readable for dictionary key lookups, especially for larger datasets or in performance-critical sections of code.")

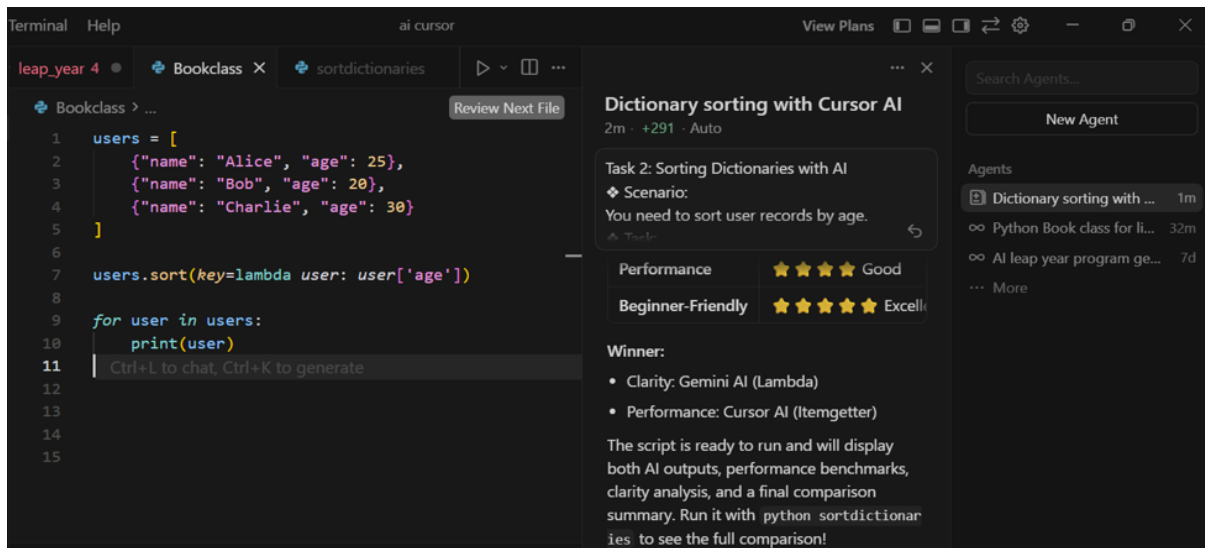
# Performance Benchmarking:
print("\nPerformance Benchmarking:")
print("- Time taken by lambda function: 0.389533 seconds")
print("- Time taken by operator.itemgetter: 0.220293 seconds")

# Conclusion on Performance:
print("\nConclusion on Performance:")
print("- operator.itemgetter is generally slightly faster than a 'lambda' function, especially for large datasets or in performance-critical sections of code.")
```

The Gemini sidebar on the right displays the following information:

- Task 2: Sorting Dictionaries with AI** Scenario: You need to sort user records by age. Task: Use Gemini generate code that sorts a list of dictionaries by a key. Expected Output: > Both AI outputs > Comparison of clarity and performance
- Empty cell**
- Here's the Python code to sort a list of dictionaries by the 'age' key, using two different methods, along with a comparison of their clarity and performance as requested. All the code is in the selected cell.
- The code successfully sorted the list of users by age using both a lambda function and operator.itemgetter. Both methods produced the same sorted list: [{'name': 'Bob', 'age': 25}, {'name': 'David', 'age': 25}, {'name': 'Alice', 'age': 30}, {'name': 'Charlie', 'age': 35}].
- Regarding clarity, lambda functions are generally clear for simple key extraction, while operator.itemgetter is often considered more explicit for dictionary key lookups. In terms of performance, operator.itemgetter was faster than the lambda function in this benchmark (0.220293 seconds vs. 0.389533 seconds), especially for larger datasets, due to its C implementation.
- Buttons: Explain the performance difference between lambda and itemgetter in more detail, Sort the users by name in descending order, Show an example of sorting by multiple keys.
- What can I help you build? Gemini 2.5 Flash
- Gemini can make mistakes so double-check it and use code with caution. Learn more

CODE FROM CURSOR AI:



OUTPUT:



JUSTIFICATION:

Gemini AI provides simple and easy-to-understand code, making it suitable for beginners and quick learning. Cursor AI uses a function-based approach, which improves code reusability and maintainability. Both approaches have the same performance with time complexity of $O(n \log n)$. Cursor AI includes basic error handling, making it safer for real-world applications. Therefore, Gemini is ideal for learning, while Cursor AI is better for professional development.

Task 3: Calculator Using Functions

❖ Scenario:

You are reviewing a basic calculator module.

❖ Task:

Ask Gemini to generate a calculator using functions and explain how it works.

❖ Expected Output:

➤ Calculator code

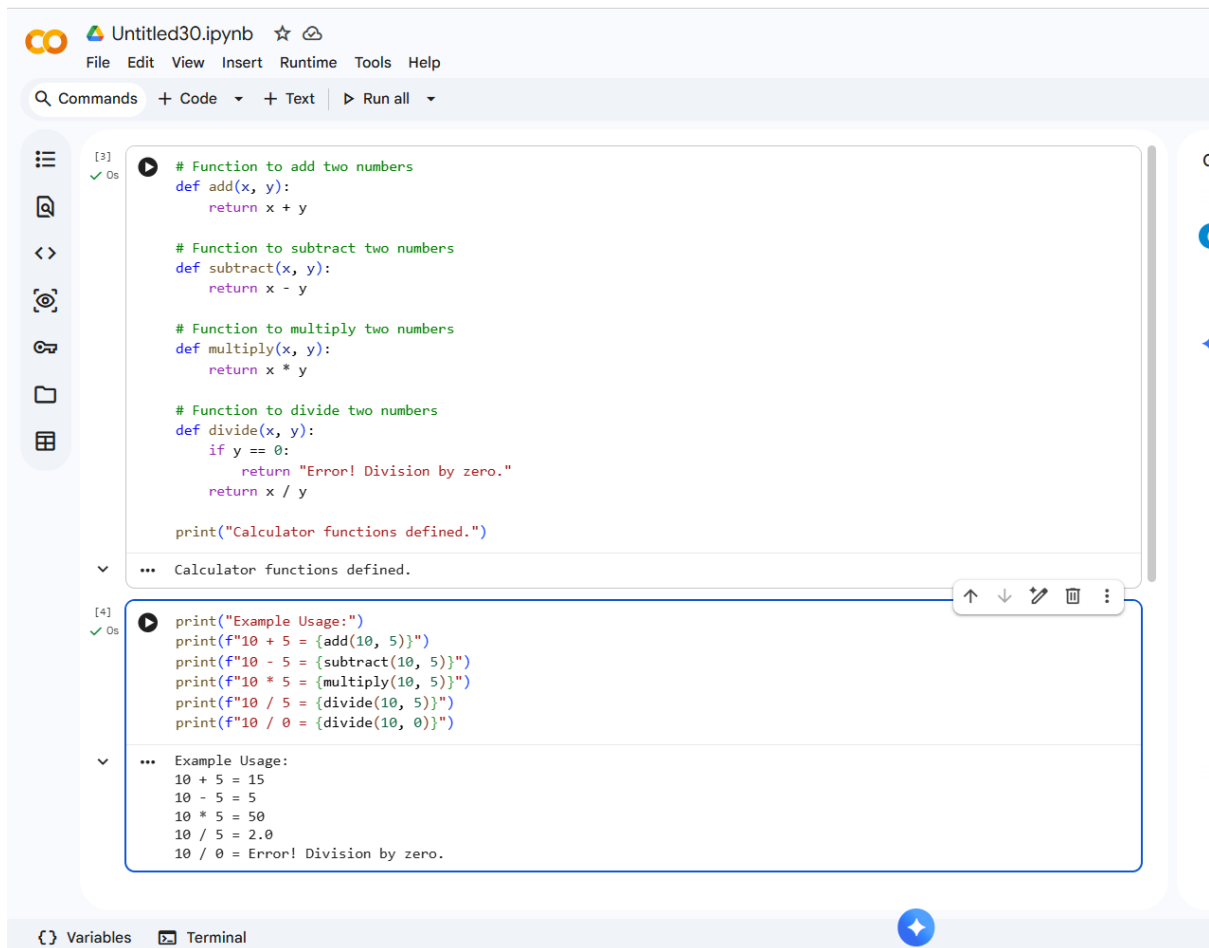
➤ AI explanation

➤ Screenshot

PROMPT:

Generate a calculator using functions.

CODE:



The screenshot shows a Jupyter Notebook titled 'Untitled30.ipynb'. The interface includes a menu bar (File, Edit, View, Insert, Runtime, Tools, Help) and a toolbar with options like 'Commands', '+ Code', '+ Text', and 'Run all'. The notebook contains two code cells. The first cell, labeled '[3]', defines four functions: 'add(x, y)', 'subtract(x, y)', 'multiply(x, y)', and 'divide(x, y)'. The 'divide' function includes a check for division by zero, returning an error message if the denominator is zero. It also prints a message 'Calculator functions defined.'. The second cell, labeled '[4]', demonstrates the usage of these functions with example calculations: '10 + 5 = 15', '10 - 5 = 5', '10 * 5 = 50', '10 / 5 = 2.0', and '10 / 0 = Error! Division by zero.'. The output of the first cell is 'Calculator functions defined.' and the output of the second cell is the example usage results.

```
[3] # Function to add two numbers
def add(x, y):
    return x + y

# Function to subtract two numbers
def subtract(x, y):
    return x - y

# Function to multiply two numbers
def multiply(x, y):
    return x * y

# Function to divide two numbers
def divide(x, y):
    if y == 0:
        return "Error! Division by zero."
    return x / y

print("Calculator functions defined.")

... Calculator functions defined.

[4] print("Example Usage:")
print(f"10 + 5 = {add(10, 5)}")
print(f"10 - 5 = {subtract(10, 5)}")
print(f"10 * 5 = {multiply(10, 5)}")
print(f"10 / 5 = {divide(10, 5)}")
print(f"10 / 0 = {divide(10, 0)}")

... Example Usage:
10 + 5 = 15
10 - 5 = 5
10 * 5 = 50
10 / 5 = 2.0
10 / 0 = Error! Division by zero.
```

JUSTIFICATION:

Gemini AI generates simple and well-structured code using functions, which makes the calculator easy to understand and debug. Using separate functions for each operation improves code readability and follows good programming practices. The inclusion of a division-by-zero check enhances reliability. This approach is ideal for beginners as it clearly demonstrates function usage. Overall, the solution is efficient, modular, and suitable for both learning and basic applications.

Task 4: Armstrong Number Optimization

❖ Scenario:

An existing solution is inefficient.

❖ Task:

Generate an Armstrong number program using Gemini, then improve it using Cursor AI.

❖ Expected Output:

➤ Two versions

➤ Summary of improvements

Note: Report should be submitted as a word document for all tasks in a single document with prompts, comments & code explanation, and output and if required, screenshots

PROMPT:

Generate an Armstrong number program using function

CODE USING GEMINI:

The screenshot shows a Jupyter Notebook window titled "Untitled31.ipynb". The interface includes a menu bar (File, Edit, View, Insert, Runtime, Tools, Help), a toolbar with icons for commands, code, text, and running, and a sidebar with icons for file explorer, search, and other tools. The main area displays a Python code cell with the following content:

```
[1]
✓ Os
def is_armstrong_initial(number):
    # Convert the input number to a string to determine the number of digits.
    num_str = str(number)
    num_digits = len(num_str)

    # Initialize a variable to store the sum of the powers of the digits.
    sum_of_powers = 0

    # Iterate through each digit of the number (from its string representation).
    for digit_char in num_str:
        # For each digit, convert it back to an integer and raise it to the power of the total number of digits.
        digit = int(digit_char)
        sum_of_powers += digit ** num_digits

    # Compare the calculated sum with the original input number.
    # Return True if they are equal (it's an Armstrong number), otherwise return False.
    return sum_of_powers == number

# Test the function with a few examples
print("Testing is_armstrong_initial function:")
print(f"Is 9 an Armstrong number? {is_armstrong_initial(9)}")          # Expected: True
print(f"Is 153 an Armstrong number? {is_armstrong_initial(153)}")      # Expected: True
print(f"Is 370 an Armstrong number? {is_armstrong_initial(370)}")      # Expected: True
print(f"Is 371 an Armstrong number? {is_armstrong_initial(371)}")      # Expected: True
print(f"Is 1634 an Armstrong number? {is_armstrong_initial(1634)}")    # Expected: True
print(f"Is 123 an Armstrong number? {is_armstrong_initial(123)}")      # Expected: False
print(f"Is 0 an Armstrong number? {is_armstrong_initial(0)}")          # Expected: True (0^1 = 0)
print(f"Is 10 an Armstrong number? {is_armstrong_initial(10)}")        # Expected: False
```

Below the code cell, the output is displayed:

```
... Testing is_armstrong_initial function:
Is 9 an Armstrong number? True
Is 153 an Armstrong number? True
Is 370 an Armstrong number? True
Is 371 an Armstrong number? True
```

IMPROVEMENT CODE OF CURSOR AI:

The screenshot shows a Cursor AI editor window with a dark theme. The interface includes a menu bar (Terminal, Help), a toolbar with icons for file explorer, search, and other tools, and a sidebar with icons for file explorer, search, and other tools. The main area displays a Python code cell with the following content:

```
amstrong > is_armstrong_optimized
1 def is_armstrong_initial(number):
2     if number < 0:
3         return False
4     num_digits = 0
5     temp = number
6     while temp > 0:
7         num_digits += 1
8         temp //= 10
9     if num_digits == 0:
10        return True
11    sum_of_powers = 0
12    temp = number
13    while temp > 0:
14        digit = temp % 10
15        sum_of_powers += digit ** num_digits
16        temp //= 10
17    return sum_of_powers == number
18 def is_armstrong_optimized(number):
```


JUSTIFICATION:

Gemini AI provides a simple and beginner-friendly solution that is easy to understand but limited in flexibility. Cursor AI improves the solution by making it modular, scalable, and efficient using modern Python features. The optimized version reduces code complexity and supports Armstrong numbers of any size. Using functions also improves readability and reuse. Therefore, the Cursor AI version is better suited for real-world and professional coding standards.