

# **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:**

## **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 the Gemini AI interface in a Jupyter Notebook environment. The code cell contains Python code demonstrating two methods for sorting a list of user records by age. The first method uses a lambda function, and the second uses operator.itemgetter. The code also includes performance benchmarks and a comparison of clarity between the two methods.

```
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." +
      "- 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
"""

# Increase data size for meaningful benchmark
n = 10000 # Increase data size for meaningful benchmark

# Benchmark lambda function
time_lambda = timeit.timeit("sorted(users, key=lambda user: user['age'])", setup=setup_code, number=n)
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=n)
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")

... Original list: [{"name": "Alice", "age": 30}, {"name": "Bob", "age": 25}, {"name": "Charlie", "age": 35}, {"name": "David", "age": 25}]

... Method 1: Sorting with lambda function ---
Sorted by age (lambda): [{"name": "Bob", "age": 25}, {"name": "David", "age": 25}, {"name": "Alice", "age": 30}, {"name": "Charlie", "age": 35}]

... Method 2: Sorting with operator.itemgetter ---
Sorted by age (itemgetter): [{"name": "Bob", "age": 25}, {"name": "David", "age": 25}, {"name": "Alice", "age": 30}, {"name": "Charlie", "age": 35}]

... Comparison: Clarity and Performance ---

Clarity:
- Lambda Function: Generally very clear for simple key extraction. Readability might decrease with more complex key lookups.
- operator.itemgetter: Often considered more explicit and sometimes more readable for dictionary key lookups, especially for large datasets.

Performance:
- Time taken by lambda function: 0.389533 seconds
- Time taken by operator.itemgetter: 0.220293 seconds

Conclusion on Performance:
- 'operator.itemgetter' is generally slightly faster than a 'lambda' function, especially for large datasets or in performance-critical applications.
```

This screenshot shows the same Gemini AI interface with the same code as the previous one, but it includes a detailed performance analysis and comparison section. The analysis highlights that 'operator.itemgetter' is generally faster than a 'lambda' function, especially for large datasets.

```
... Conclusion on Performance:
- 'operator.itemgetter' is generally slightly faster than a 'lambda' function, especially for large datasets or in performance-critical applications.

Gemini 2.5 Flash > ▶
```

## CODE FROM CURSOR AI:

The screenshot shows the Cursor AI interface. On the left, a code editor displays Python code for sorting a list of users by age. On the right, a summary card for "Dictionary sorting with Cursor AI" provides details about the task, scenario, performance, and beginner-friendliness.

```
leap_year 4 ● Bookclass X sortdictionaries Review Next File
1 users = [
2     {"name": "Alice", "age": 25},
3     {"name": "Bob", "age": 20},
4     {"name": "Charlie", "age": 30}
5 ]
6
7 users.sort(key=lambda user: user['age'])
8
9 for user in users:
10     print(user)
11
12
13
14
15
```

**Dictionary sorting with Cursor AI**  
2m · +291 · Auto

**Task 2: Sorting Dictionaries with AI**  
❖ Scenario:  
You need to sort user records by age.

**Performance** ★★★★ Good  
**Beginner-Friendly** ★★★★★ Excellent

**Winner:**

- Clarity: Gemini AI (Lambda)
- Performance: Cursor AI (Itemgetter)

The script is ready to run and will display both AI outputs, performance benchmarks, clarity analysis, and a final comparison summary. Run it with `python sortdictionaries` to see the full comparison!

## OUTPUT:

The terminal output shows the sorted list of users:

```
op/ai cursor/Bookclass"
{'name': 'Bob', 'age': 20}
{'name': 'Alice', 'age': 25}
{'name': 'Charlie', 'age': 30}
```

Plan, @ for context, / for commands

## 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 interface with the following code:

```
# 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.

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 interface with the following details:

- Title Bar:** Untitled31.ipynb
- Toolbar:** File, Edit, View, Insert, Runtime, Tools, Help
- Menu Bar:** Q Commands, + Code, + Text, ▶ Run all
- Code Cell:**

```
[1] 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
```
- Output Cell:**

```
... 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
```
- Bottom Navigation:** Variables, Terminal

## IMPROVEMENT CODE OF CURSOR AI:

The screenshot shows a terminal window with the following details:

- Terminal Tab:** leap\_year 4
- Current Directory:** amstrong
- Code:**

```
def is_armstrong_initial(number):
    if number < 0:
        return False
    num_digits = 0
    temp = number
    while temp > 0:
        num_digits += 1
        temp //= 10
    if num_digits == 0:
        return True
    sum_of_powers = 0
    temp = number
    while temp > 0:
        digit = temp % 10
        sum_of_powers += digit ** num_digits
        temp //= 10
    return sum_of_powers == number
```
- Bottom Right:** Review Next File

n Terminal Help ai cursor View Plans

leap\_year 4 Bookclass amstrong sortdictionaries

amstrong > is\_armstrong\_optimized

Review Next File

```
17     return sum_of_powers == number
18 def is_armstrong_optimized(number):
19     if number < 0:
20         return False
21     num_str = str(number)
22     num_digits = len(num_str)
23     sum_of_powers = sum(int(digit) ** num_digits for digit in num_str)
24     return sum_of_powers == number
25 if __name__ == "__main__":
26     print("Testing is_armstrong_initial function:")
27     print(f"Is 9 an Armstrong number? {is_armstrong_initial(9)}") # Expected: True
28     print(f"Is 153 an Armstrong number? {is_armstrong_initial(153)}") # Expected: True
29     print(f"Is 370 an Armstrong number? {is_armstrong_initial(370)}") # Expected: True
30     print(f"Is 371 an Armstrong number? {is_armstrong_initial(371)}") # Expected: True
31     print(f"Is 1634 an Armstrong number? {is_armstrong_initial(1634)}") # Expected: True
32     print(f"Is 123 an Armstrong number? {is_armstrong_initial(123)}") # Expected: False
33     print(f"Is 0 an Armstrong number? {is_armstrong_initial(0)}") # Expected: True
34     print(f"Is 10 an Armstrong number? {is_armstrong_initial(10)}") # Expected: False
```

## **OUTPUT:**

## **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.