

ASSIGNMENT – 2.4

Name:N.Soumya

H NO:2303A51869

Batch-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

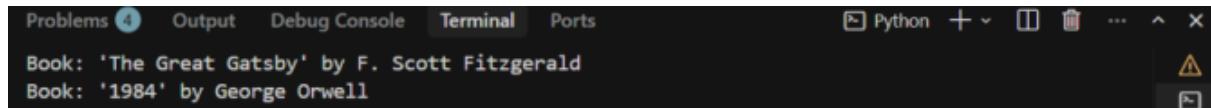
PROMPT:

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

CODE:

```
Bookclass > Book > __init__  
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
```

OUTPUT:



A screenshot of a Python terminal window. The title bar says "Python". The menu bar includes "File", "Edit", "View", "Insert", "Cell", "Kernel", "Help", and "File". The main area displays two lines of text: "Book: 'The Great Gatsby' by F. Scott Fitzgerald" and "Book: '1984' by George Orwell".

JUSTIFICATION:

This program is used to store book details like title and author in a single unit.

The class helps create multiple book objects without rewriting code.

The summary function prints book information in a clear and fixed format.

The main block runs the program and shows the output for 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:

```

▶ # A list of dictionaries
data = [
    {"name": "Alice", "age": 30, "city": "New York"},
    {"name": "Bob", "age": 24, "city": "Los Angeles"},
    {"name": "Charlie", "age": 35, "city": "Chicago"},
    {"name": "David", "age": 24, "city": "Houston"}
]

# Sort the list of dictionaries by the 'age' key
sorted_data = sorted(data, key=lambda x: x['age'])

print("Original data:")
for item in data:
    print(item)

print("\nSorted data by 'age':")
for item in sorted_data:
    print(item)

# You can also sort by another key, for example 'name'
sorted_data_by_name = sorted(data, key=lambda x: x['name'])

print("\nSorted data by 'name':")
for item in sorted_data_by_name:
    print(item)

# For descending order, add reverse=True
sorted_data_desc = sorted(data, key=lambda x: x['age'], reverse=True)

print("\nSorted data by 'age' (descending):")
for item in sorted_data_desc:
    print(item)

```

OUTPUT:

```

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

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

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

Sorted data by 'age' (descending):
{'name': 'Charlie', 'age': 35, 'city': 'Chicago'}
{'name': 'Alice', 'age': 30, 'city': 'New York'}
{'name': 'Bob', 'age': 24, 'city': 'Los Angeles'}
{'name': 'David', 'age': 24, 'city': 'Houston'}

```

CODE FROM CURSOR AI:

The screenshot shows a code editor interface with two tabs: 'Bookclass' and 'sortdictionaries'. The 'sortdictionaries' tab contains the following Python code:

```

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

```

Below the code, there's a note: 'Ctrl+L to chat, Ctrl+K to generate'.

To the right of the code editor is a comparison panel titled 'Dictionary sorting with Cursor AI' which includes a summary, scenario details, performance metrics, and a 'Winner' section comparing Gemini AI and Cursor AI.

OUTPUT:

The terminal window shows the following command and output:

```

print(users)
^^^^^
IndentationError: expected an indented block after 'for' statement on line 11
PS C:\Users\Spriha Gajula\OneDrive\Desktop\ai cursor> & "C:/Users/Spriha Gajula/anaconda3/envs/spriha185@/python.exe" "c:/Users/Spriha Gajula/OneDrive/Desktop/op/ai cursor/Bookclass"
{'name': 'Bob', 'age': 20}
{'name': 'Alice', 'age': 25}
{'name': 'Charlie', 'age': 30}

```

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:

```
❶ def add(x, y):
    return x + y

def subtract(x, y):
    return x - y

def multiply(x, y):
    return x * y

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

def calculator():
    print("Select operation:")
    print("1. Add")
    print("2. Subtract")
    print("3. Multiply")
    print("4. Divide")

    while True:
        choice = input("Enter choice(1/2/3/4): ")

        if choice in ('1', '2', '3', '4'):
            try:
                num1 = float(input("Enter first number: "))
                num2 = float(input("Enter second number: "))
            except ValueError:
                print("Invalid input. Please enter numbers only.")
                continue

            if choice == '1':
                print(f"{num1} + {num2} = {add(num1, num2)}")

            elif choice == '2':
                print(f"{num1} - {num2} = {subtract(num1, num2)}")

            elif choice == '3':
                print(f"{num1} * {num2} = {multiply(num1, num2)}")

            elif choice == '4':
                result = divide(num1, num2)
                print(f"{num1} / {num2} = {result}")

            next_calculation = input("Let's do next calculation? (yes/no): ")
            if next_calculation.lower() == "no":
                break
            else:
                print("Invalid input. Please enter a valid choice.")

calculator()
```

OUTPUT:

```
... Select operation:
1. Add
2. Subtract
3. Multiply
4. Divide
Enter choice(1/2/3/4): 1
Enter first number: 23
Enter second number: 76
23.0 + 76.0 = 99.0
Let's do next calculation? (yes/no): yes
Enter choice(1/2/3/4): 4
Enter first number: 67
Enter second number: 17
67.0 / 17.0 = 3.9411764705882355
Let's do next calculation? (yes/no): no
```

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

PROMPT:

Generate an Armstrong number program using function

CODE USING GEMINI:

```
❶ def is_armstrong_number(num):  
    # Convert number to string to find its length (number of digits)  
    num_str = str(num)  
    n = len(num_str)  
  
    sum_of_powers = 0  
    temp_num = num  
  
    while temp_num > 0:  
        digit = temp_num % 10  
        sum_of_powers += digit ** n  
        temp_num //= 10  
  
    return num == sum_of_powers  
  
def find_armstrong_numbers_in_range(start, end):  
    armstrong_numbers = []  
    for i in range(start, end + 1):  
        if is_armstrong_number(i):  
            armstrong_numbers.append(i)  
    return armstrong_numbers  
  
# Get input from the user  
try:  
    number_to_check = int(input("Enter a number to check if it's an Armstrong number: "))  
    if is_armstrong_number(number_to_check):  
        print(f"{number_to_check} is an Armstrong number.")  
    else:
```

OUTPUT:

```
... Enter a number to check if it's an Armstrong number: 234
234 is not an Armstrong number.

--- Finding Armstrong numbers in a range (e.g., 1 to 1000) ---
Armstrong numbers between 1 and 1000 are: [1, 2, 3, 4, 5, 6, 7, 8, 9, 153, 370, 371, 407]
```

IMPROVEMENT CODE OF CURSOR AI:

```
leap_year 4 BOOKCLASS armstrong sortdictionaries
amstrong > is_armstrong_optimized
Review Next File

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):
```

```
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.