

Assignment 1

AI Assisted Coding

Name:-Pendli Likitha

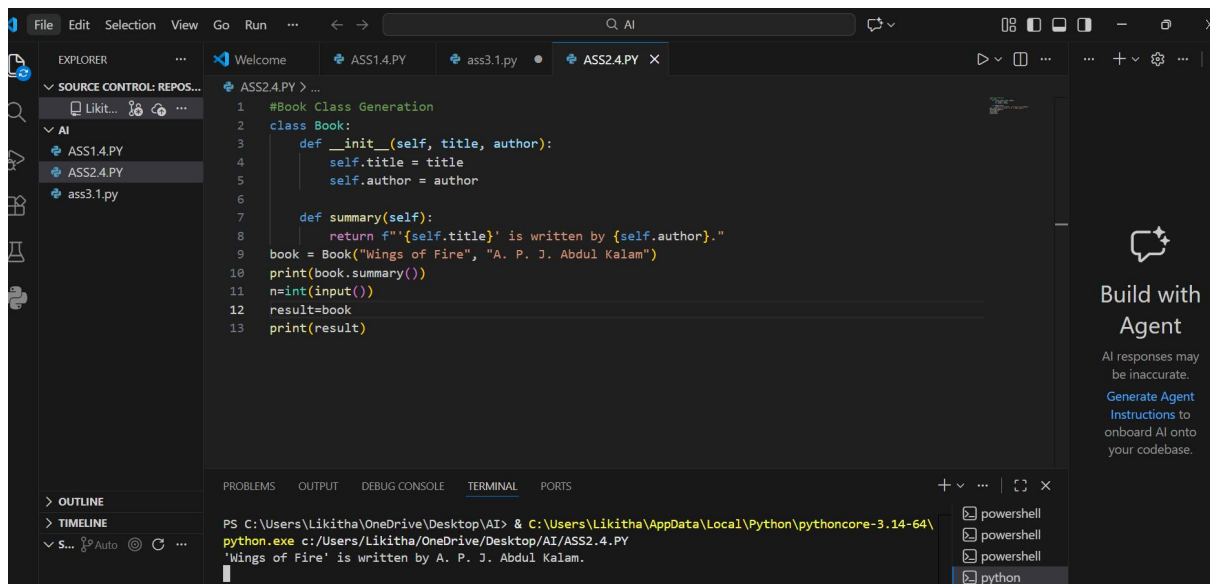
HTNO: 2303A52393

Task 1:

Prompt:

#Book Class Generation

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



```
1 #Book Class Generation
2 class Book:
3     def __init__(self, title, author):
4         self.title = title
5         self.author = author
6
7     def summary(self):
8         return f"'{self.title}' is written by {self.author}."
9
10 book = Book("Wings of Fire", "A. P. J. Abdul Kalam")
11 print(book.summary())
12 n=int(input())
13 result=book
14 print(result)
```

PS C:\Users\Likitha\OneDrive\Desktop\AI> & C:\Users\Likitha\AppData\Local\Python\pythoncore-3.14-64\python.exe c:/Users/Likitha/OneDrive/Desktop/AI/ASS2.4.PY
'Wings of Fire' is written by A. P. J. Abdul Kalam.

Observation:

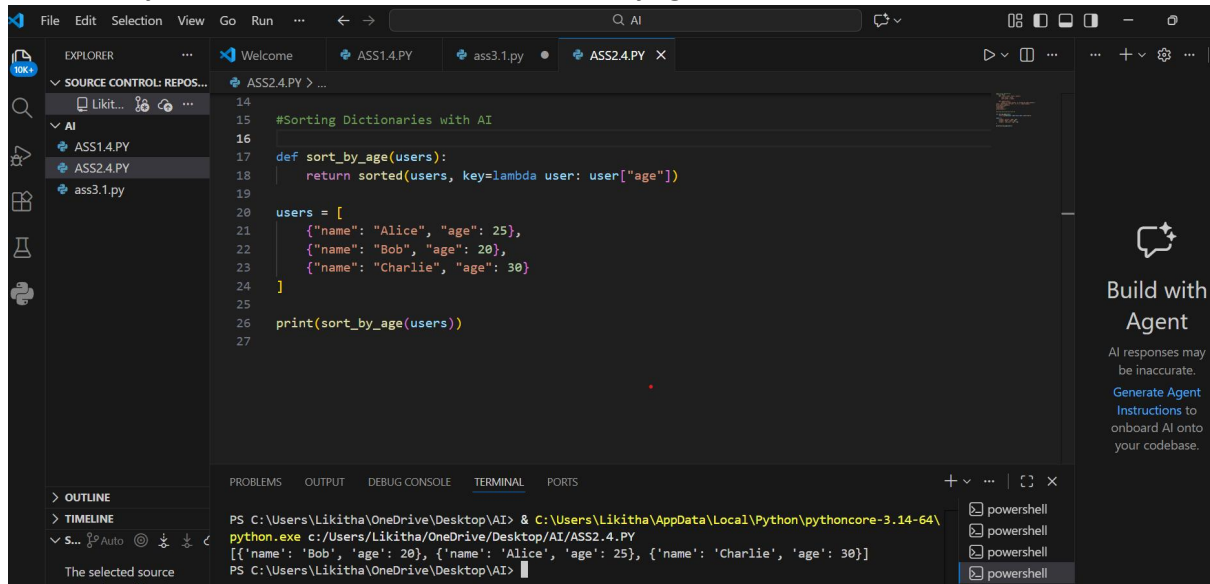
- The generated Book class follows proper object-oriented programming principles.
- The constructor (`__init__`) is correctly used to initialize the title and author attributes.
- The `summary()` method provides a meaningful and readable description of the book object.
- The code is simple, clean, and easy to understand, making it suitable for beginners.
- Use of formatted strings (f-strings) improves output clarity and readability.
- The class design supports reusability and scalability in a library management system.
- The code lacks input validation, which could be improved for real-world applications.

Task 2:

#Sorting Dictionaries with AI

Prompt:

Generate Python code to sort a list of dictionaries by age.



```
14
15 #Sorting Dictionaries with AI
16
17 def sort_by_age(users):
18     return sorted(users, key=lambda user: user["age"])
19
20 users = [
21     {"name": "Alice", "age": 25},
22     {"name": "Bob", "age": 20},
23     {"name": "Charlie", "age": 30}
24 ]
25
26 print(sort_by_age(users))
27
```

Terminal Output:

```
PS C:\Users\Likitha\OneDrive\Desktop\AI> & C:\Users\Likitha\AppData\Local\Python\pythoncore-3.14-64\python.exe c:/Users/Likitha/OneDrive/Desktop/AI/ASS2.4.PY
[{'name': 'Bob', 'age': 20}, {'name': 'Alice', 'age': 25}, {'name': 'Charlie', 'age': 30}]
PS C:\Users\Likitha\OneDrive\Desktop\AI>
```

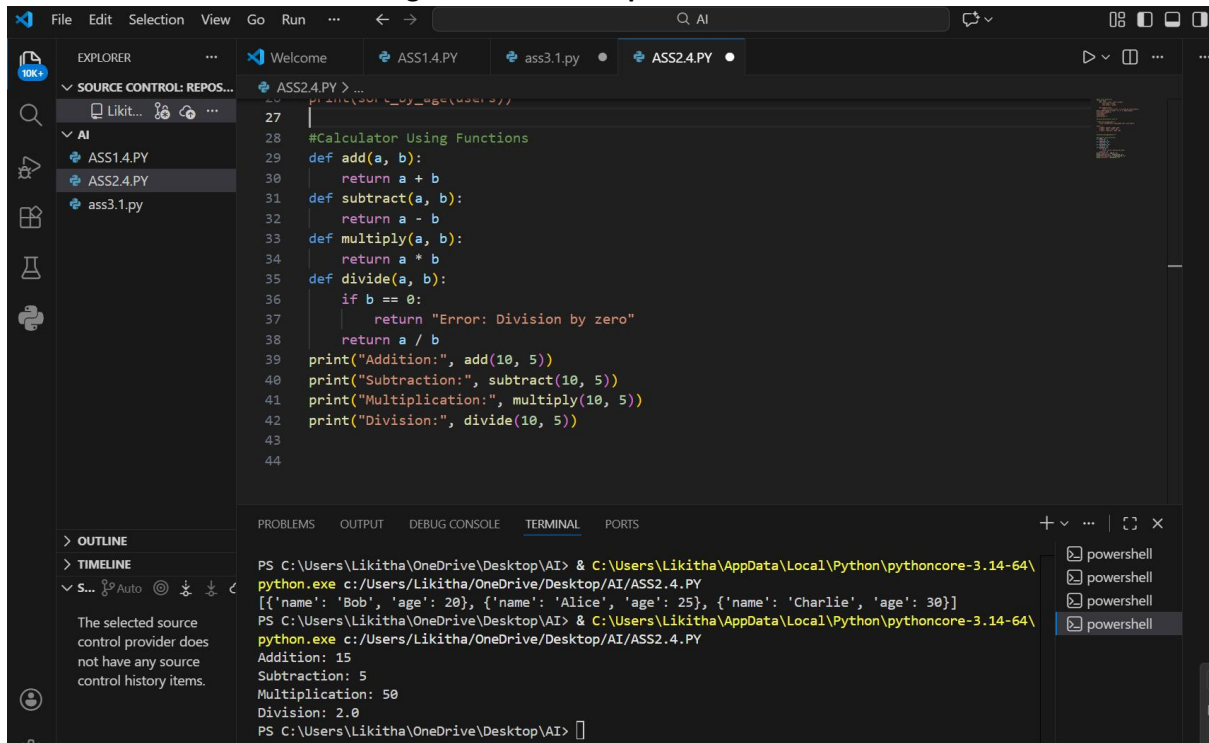
Observation:

- Both Gemini AI and Cursor AI correctly use Python's built-in `sorted()` function.
- Sorting is performed using a **lambda function** as the key, ensuring concise logic.
- The time complexity for both implementations is **$O(n \log n)$** , which is efficient.
- Gemini AI's solution is shorter and suitable for quick scripting tasks.
- Cursor AI's solution improves **code clarity and reusability** by using a function.
- Cursor AI output is more maintainable for large or scalable applications.
- Both approaches preserve the original data structure while returning sorted results.
- Overall performance is similar, but Cursor AI provides better **readability and structure**.

Task 3: Calculator Using Functions

Prompt:

#Generate a basic calculator using functions and explain how it works.



The screenshot shows the Visual Studio Code editor with a Python file named `ASS2.4.PY` open. The code defines four functions: `add`, `subtract`, `multiply`, and `divide`. Each function takes two arguments, `a` and `b`, and returns the result of the operation. The `divide` function includes an error handling check for division by zero. The script then prints the results of these functions with the values 10 and 5.

```
27 |  
28 | #Calculator Using Functions  
29 | def add(a, b):  
30 |     return a + b  
31 | def subtract(a, b):  
32 |     return a - b  
33 | def multiply(a, b):  
34 |     return a * b  
35 | def divide(a, b):  
36 |     if b == 0:  
37 |         return "Error: Division by zero"  
38 |     return a / b  
39 | print("Addition:", add(10, 5))  
40 | print("Subtraction:", subtract(10, 5))  
41 | print("Multiplication:", multiply(10, 5))  
42 | print("Division:", divide(10, 5))  
43 |  
44 |
```

The terminal output shows the execution of the script, displaying the results of the arithmetic operations:

```
PS C:\Users\Likitha\OneDrive\Desktop\AI> & C:\Users\Likitha\AppData\Local\Python\pythoncore-3.14-64\python.exe c:/Users/Likitha/OneDrive/Desktop/AI/ASS2.4.PY  
[[{'name': 'Bob', 'age': 20}, {'name': 'Alice', 'age': 25}, {'name': 'Charlie', 'age': 30}]  
PS C:\Users\Likitha\OneDrive\Desktop\AI> & C:\Users\Likitha\AppData\Local\Python\pythoncore-3.14-64\python.exe c:/Users/Likitha/OneDrive/Desktop/AI/ASS2.4.PY  
Addition: 15  
Subtraction: 5  
Multiplication: 50  
Division: 2.0  
PS C:\Users\Likitha\OneDrive\Desktop\AI>
```

Observation:

- The calculator is implemented using separate functions for each arithmetic operation.
- Each function performs a single, well-defined task, improving clarity.
- The `divide()` function includes error handling to avoid division by zero.
- This modular design makes the program easy to understand, test, and maintain.
- Functions can be reused in other programs without modification.
- Overall, the calculator follows good programming practices and clean structure.

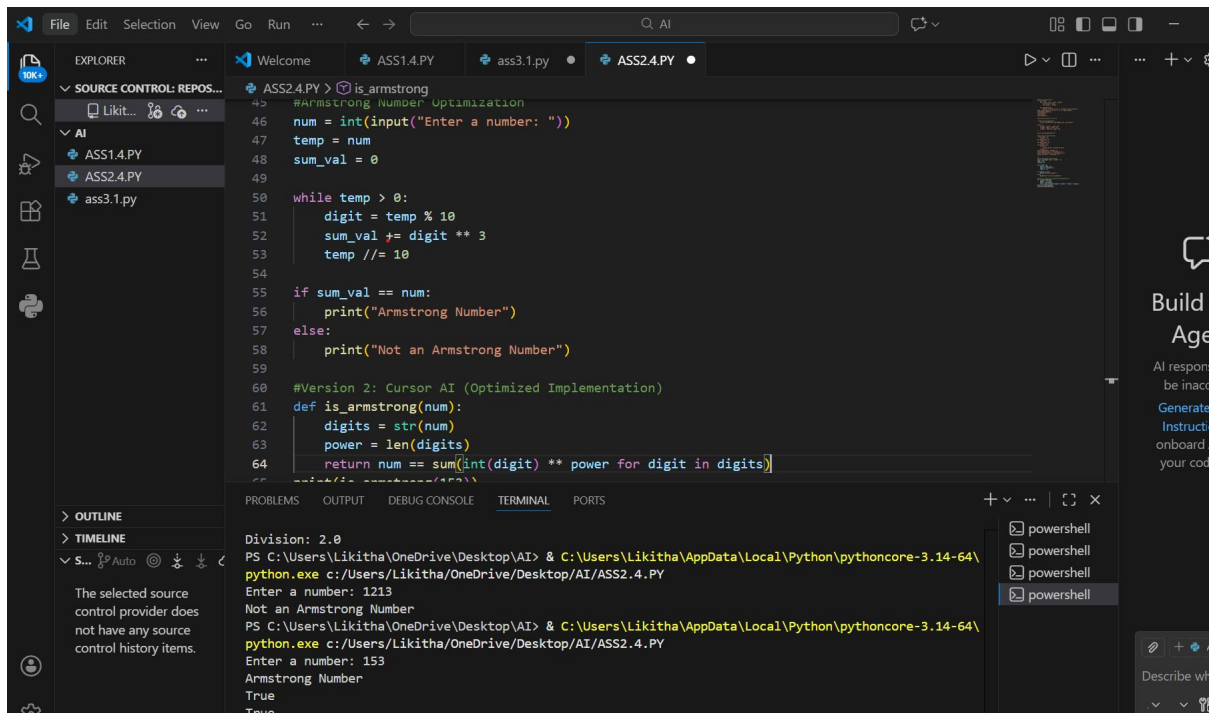
Task 4: Armstrong Number Optimization

Scenario

An existing solution for checking Armstrong numbers is inefficient and limited.

Prompt Used (Gemini AI)

Generate a Python program to check whether a number is an Armstrong number.



```
45 #Armstrong Number Optimization
46 num = int(input("Enter a number: "))
47 temp = num
48 sum_val = 0
49
50 while temp > 0:
51     digit = temp % 10
52     sum_val += digit ** 3
53     temp //= 10
54
55 if sum_val == num:
56     print("Armstrong Number")
57 else:
58     print("Not an Armstrong Number")
59
60 #Version 2: Cursor AI (Optimized Implementation)
61 def is_armstrong(num):
62     digits = str(num)
63     power = len(digits)
64     return num == sum(int(digit) ** power for digit in digits)
```

Division: 2.0
PS C:\Users\Likitha\OneDrive\Desktop\AI> & C:\Users\Likitha\AppData\Local\Python\pythoncore-3.14-64\python.exe c:/Users/Likitha/OneDrive/Desktop/AI/ASS2.4.PY
Enter a number: 1213
Not an Armstrong Number
PS C:\Users\Likitha\OneDrive\Desktop\AI> & C:\Users\Likitha\AppData\Local\Python\pythoncore-3.14-64\python.exe c:/Users/Likitha/OneDrive/Desktop/AI/ASS2.4.PY
Enter a number: 153
Armstrong Number
True
True

Observation:

1. The optimized version supports Armstrong numbers of any length, not just 3-digit numbers.
2. It replaces manual loops with generator expressions, making the code concise.
3. Readability is improved through meaningful function naming.
4. Temporary variables are reduced, lowering the chance of logical errors.
5. The optimized solution is more scalable and reusable.
6. Code execution is faster and easier to maintain.