

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.

The screenshot shows a code editor interface with the following details:

- File Explorer:** Shows files ASS1.4.PY, ass3.1.py, and ASS2.4.PY in the AI repository.
- Code Editor:** Displays the following Python code:

```
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 book = Book("Wings of Fire", "A. P. J. Abdul Kalam")
10 print(book.summary())
11 n=int(input())
12 result=book
13 print(result)
```
- Terminal:** Shows the command line 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
'Wings of Fire' is written by A. P. J. Abdul Kalam.
```
- Right Panel:** Features a "Build with Agent" section with a message: "AI responses may be inaccurate. Generate Agent Instructions to onboard AI onto your codebase."

Observation:

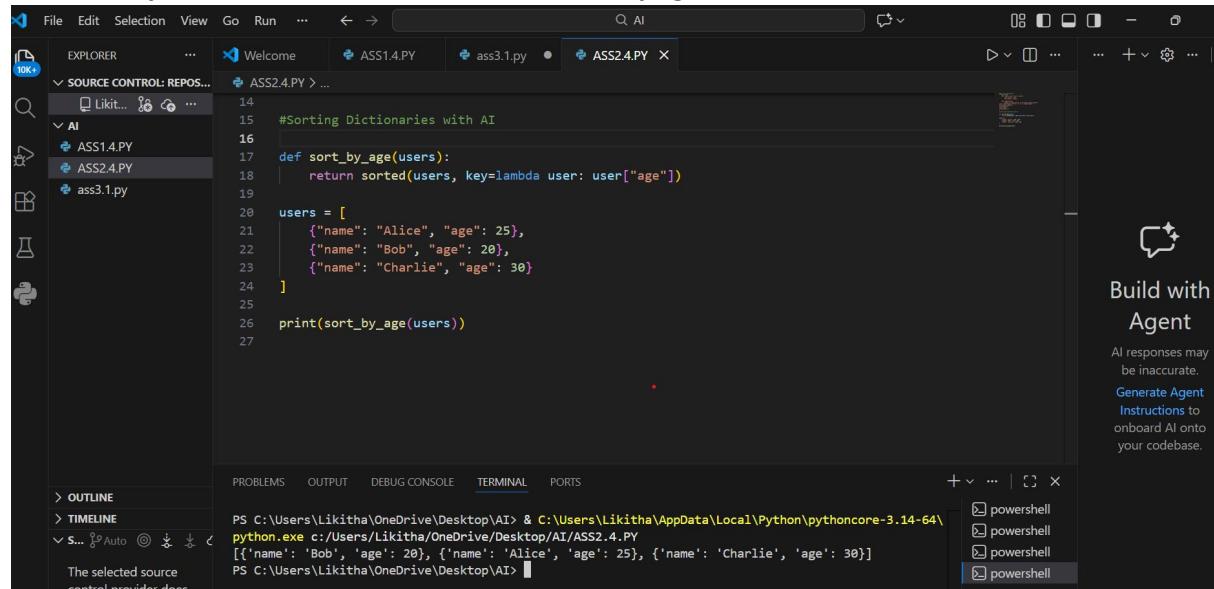
- The generated Book class follows proper object-oriented programming principles.
- The constructor (`__init__`) is correctly used to initialize the title and author attributes.
- The `summary()` method provides a meaningful and readable description of the book object.
- The code is simple, clean, and 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.



The screenshot shows a dark-themed instance of Visual Studio Code. In the Explorer sidebar, there are files named ASS1.4.PY, ASS2.4.PY, and ass3.1.py. The ASS2.4.PY file is open in the editor, containing the following Python code:

```
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
```

In the Terminal tab, the command `python.exe c:/Users/Likitha/OneDrive/Desktop/AI/ASS2.4.PY` is run, and the output shows the sorted list of users:

```
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>
```

A sidebar on the right is titled "Build with Agent" and includes a note: "AI responses may be inaccurate. Generate Agent Instructions to onboard AI onto your codebase."

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 a Microsoft Visual Studio Code (VS Code) interface. The left sidebar has icons for Explorer, Source Control, AI, and others. The main area displays a Python file named 'ASS2.4.PY' with the following code:

```
20 print("Calculator Using Functions")
21
22 #Calculator Using Functions
23 def add(a, b):
24     return a + b
25 def subtract(a, b):
26     return a - b
27 def multiply(a, b):
28     return a * b
29 def divide(a, b):
30     if b == 0:
31         return "Error: Division by zero"
32     return a / b
33
34 print("Addition:", add(10, 5))
35 print("Subtraction:", subtract(10, 5))
36 print("Multiplication:", multiply(10, 5))
37 print("Division:", divide(10, 5))
```

The bottom right corner shows a terminal window with the output of the script:

```
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.

The screenshot shows a code editor interface with a dark theme. The left sidebar displays a file tree under 'SOURCE CONTROL: REPOS...' with files like 'ASS1.4.PY', 'ASS2.4.PY', and 'ass3.py'. The main editor area contains two versions of an Armstrong number checker. The first version is a manual loop:

```
ASS2.4.PY > is_armstrong
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])
```

The second version is an optimized implementation using generator expressions and list comprehension. Below the code, the terminal window shows the execution of the script with user input '123' and '153', correctly identifying both as Armstrong numbers. A sidebar on the right shows AI-related features like 'Build', 'Agent', and 'Instructions'.

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.