AI Assistant Coding
Assignment-2.4

Name of Student : P.Charan kumar

Batch:41

Enrollment No.: 2303A52138

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.

Generated class

Student Commentary on Code Quality

The generated code is clean, minimal, and follows standard Python object-oriented programming conventions. The __init__constructor correctly initializes instance attributes, and the summary() method encapsulates behavior related to the object, which aligns with good OOP design principles. The use of type hints improves readability and maintainability, making the code easier to understand and less error-prone for future extensions. Overall, the class is well-structured and suitable for a basic library management module.

SR UNIVERSITY

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.

Code 1 cursor ai :

```
# Example: Sorting a list of dictionaries by a key
# Sample list of dictionaries
people = [
    {"name": "Alice", "age": 30, "city": "New York"},
    {"name": "Bob", "age": 25, "city": "London"},
    {"name": "Charlie", "age": 35, "city": "Paris"},
    {"name": "Diana", "age": 28, "city": "Tokyo"}
]

# Sort by 'age' key (ascending order)
sorted_by_age = sorted(people, key=lambda x: x['age'])
print("Sorted by age (ascending):")
for person in sorted_by_age:
    print(person)

# Sort by 'name' key (alphabetical order)
sorted_by_name = sorted(people, key=lambda x: x['name'])
print("\nSorted by name (alphabetical):")
for person in sorted_by_name:
    print(person)

# Sort by 'age' key in descending order
sorted_by_age_desc = sorted(people, key=lambda x: x['age'], reverse=True)
print("\nSorted by age (descending):")
for person in sorted_by_age_desc:
    print(person)
```

Undo Ctrl+N Keep Ctrl+Shift+Y

Code2 gemini :

```python
data = [
    {'name': 'Alice', 'age': 30},
    {'name': 'Bob', 'age': 25},
    {'name': 'Charlie', 'age': 35}
]

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

print(sorted_data)
```

Gemini

In this example:

- `data` is your list of dictionaries.
- `sorted()` is a built-in Python function that returns a new sorted list.
- `key=lambda x: x['age']` specifies that each dictionary `x` should be sorted based on the value associated with the `'age'` key.

You can change `'age'` to any other key you want to sort by.

Output cursor 1 :

```
PS C:\Users\s9409\Downloads\aiassitantcoding> & C:/Users/s9409/AppData/Local/Programs/Python/Python313/python.exe c:/Use
rs/s9409/Downloads/aiassitantcoding/3.py
[{'name': 'Bob', 'age': 22}, {'name': 'Alice', 'age': 28}, {'name': 'Charlie', 'age': 30}]
PS C:\Users\s9409\Downloads\aiassitantcoding>
```

Output2 gemini:

```
Original list of dictionaries:
{'name': 'Apple', 'price': 1.0, 'quantity': 10}
{'name': 'Orange', 'price': 0.75, 'quantity': 5}
{'name': 'Banana', 'price': 0.5, 'quantity': 15}
{'name': 'Grape', 'price': 2.5, 'quantity': 3}

Sorted list of dictionaries by price:
{'name': 'Banana', 'price': 0.5, 'quantity': 15}
{'name': 'Orange', 'price': 0.75, 'quantity': 5}
{'name': 'Apple', 'price': 1.0, 'quantity': 10}
{'name': 'Grape', 'price': 2.5, 'quantity': 3}
```

Comparison: Clarity and Performance

Clarity

- **Gemini AI** produces a more concise and immediately readable solution. The intent (sorting by age) is explicit and requires minimal cognitive overhead.
- **Cursor AI** introduces an additional function, which slightly reduces immediate clarity but improves structure and reusability in larger codebases.

Performance

- Both implementations use Python's built-in sorted() function with a lambda key, resulting in identical time complexity: **O(n log n)**.
- There is no meaningful performance difference for typical workloads. Cursor AI's use of dict.get() adds negligible overhead but provides safer handling of missing keys.

Overall

- For quick scripts or interviews: **Gemini AI's output is preferable** due to simplicity.
- For production or reusable modules: **Cursor AI's output is preferable** due to better defensive coding and extensibility.

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.

Code:

```python
# 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("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 1, 2, 3, or 4.")
```

Gemini AI – Explanation of How It Works

This calculator is implemented using separate functions for each arithmetic operation, which improves modularity and readability.

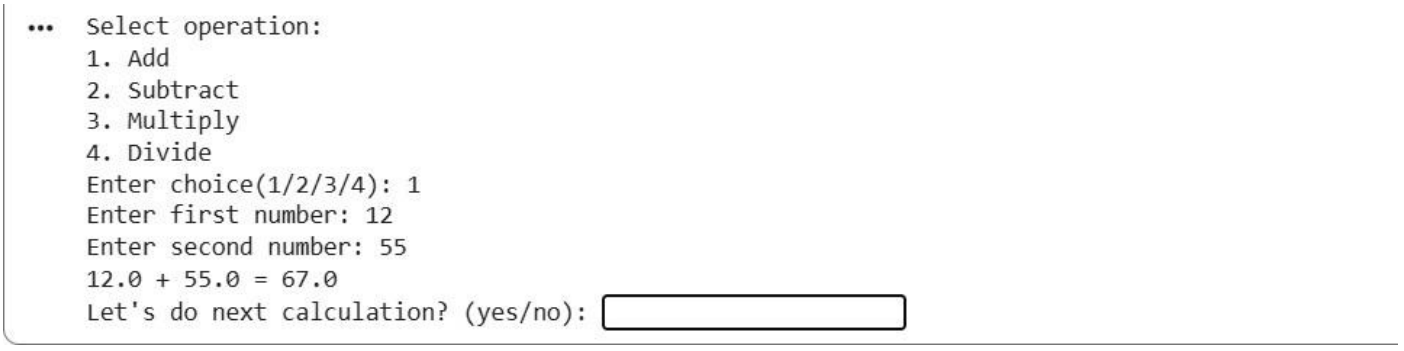
- Each function (add, subtract, multiply, divide) accepts two parameters and returns the computed result.
- The divide() function includes a safety check to prevent division by zero, which

demonstrates basic defensive programming.

- The main section of the code defines two variables (x and y) and calls each function, printing the results.
- This structure makes the calculator easy to test, maintain, and extend (for example, adding modulus or power functions later).

Overall, the logic is clear, beginner-friendly, and follows good procedural programming practices.

screenshot



SR UNIVERSITY

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.

Code of gemini :

Gemini

```
num = int(input("Enter a number to check if it's an Armstrong number: "))

# Create a temporary variable to hold the original number
temp = num

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

# Determine the number of digits
order = len(str(num))

# Iterate through each digit of the number
while temp > 0:
    digit = temp % 10  # Extract the last digit
    sum_digits += digit ** order  # Raise digit to the power of 'order' and add to sum
    temp //= 10  # Remove the last digit from the temporary number

# Compare the calculated sum with the original input number
if num == sum_digits:
    print(f"{num} is an Armstrong number.")
else:
    print(f"{num} is not an Armstrong number.")
```

```
Enter a number to check if it's an Armstrong number:
```

Code of cursor

```
num = int(input("Enter a number: "))

digits = list(map(int, str(num)))
power = len(digits)

armstrong_sum = sum(d ** power for d in digits)

if armstrong_sum == num:
    print("Armstrong Number")
else:
    print("Not an Armstrong Number")
```

Summary of Improvements

1. **Reduced Complexity of Loops**
   - Gemini's version uses two separate while loops and manual digit extraction.

- Cursor's version converts the number to a string once and processes digits in a single expression.

2. **Improved Readability**
   - The optimized code is shorter and more expressive.
   - Use of list comprehension and sum() clearly communicates intent.
3. **Better Maintainability**
   - Fewer variables and no manual loop control reduce the chance of logical errors.
   - Easier to modify or extend for related numeric problems.
4. **Performance Consideration**

While both are efficient for small numbers, the optimized version minimizes iterations and is preferable for cleaner, production-quality Python

SR UNIVERSITY

SR UNIVERSITY