ASSIGNMENT – 2.4

2303A51850

B-13

Task 1: Book Class Generation

❖ Scenario:

You are building a simple library management module.

❖ Task:

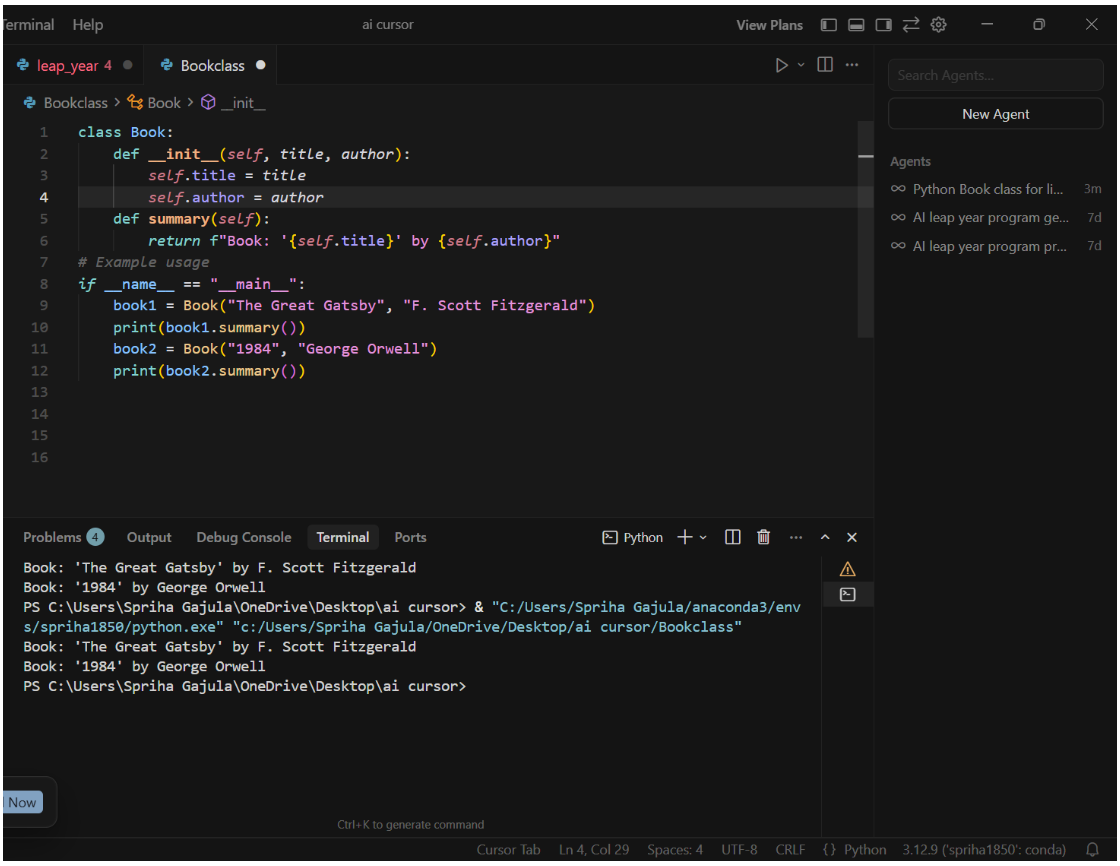
Use Cursor AI to generate a Python class Book with attributes title, author,and a summary() method.

❖ Expected Output:

➢ Generated class

➢ Student commentary on code quality

CODE:



```python
class Book:
    def __init__(self, title, author):
        self.title = title
        self.author = author
    def summary(self):
        return f"Book: '{self.title}' by {self.author}"
# Example usage
if __name__ == "__main__":
    book1 = Book("The Great Gatsby", "F. Scott Fitzgerald")
    print(book1.summary())
    book2 = Book("1984", "George Orwell")
    print(book2.summary())
```

```
Book: 'The Great Gatsby' by F. Scott Fitzgerald
Book: '1984' by George Orwell
PS C:\Users\Spriha Gajula\OneDrive\Desktop\ai cursor> & "C:/Users/Spriha Gajula/anaconda3/env
s/spriha1850/python.exe" "c:/Users/Spriha Gajula/OneDrive/Desktop/ai cursor/Bookclass"
Book: 'The Great Gatsby' by F. Scott Fitzgerald
Book: '1984' by George Orwell
PS C:\Users\Spriha Gajula\OneDrive\Desktop\ai cursor>
```

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:

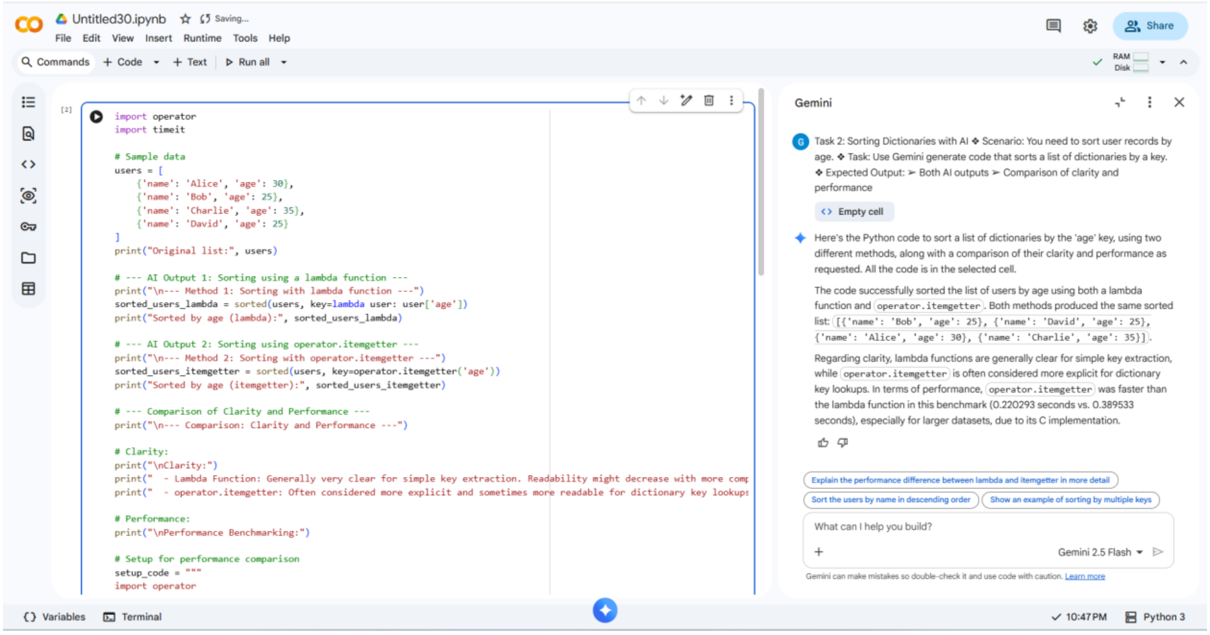
Use Gemini and Cursor AI to generate code that sorts a list of dictionaries by a key.

❖ Expected Output:

➢ Both AI outputs

➢ Comparison of clarity and performance

CODE FROM GEMINI:



Untitled30.ipynb
Saving...
File
Edit
View
Insert
Runtime
Tools
Help
Commands
+ Code
+ Text
Run all
Share
RAM
Disk
[2]
Gemini
Empty cell
Explain the performance difference between lambda and itemgetter in more detail
Sort the users by name in descending order
Show an example of sorting by multiple keys
What can I help you build?
Gemini 2.5 Flash
Variables
Terminal
10:47 PM
Python 3

```python
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 comp
print("  - 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
```

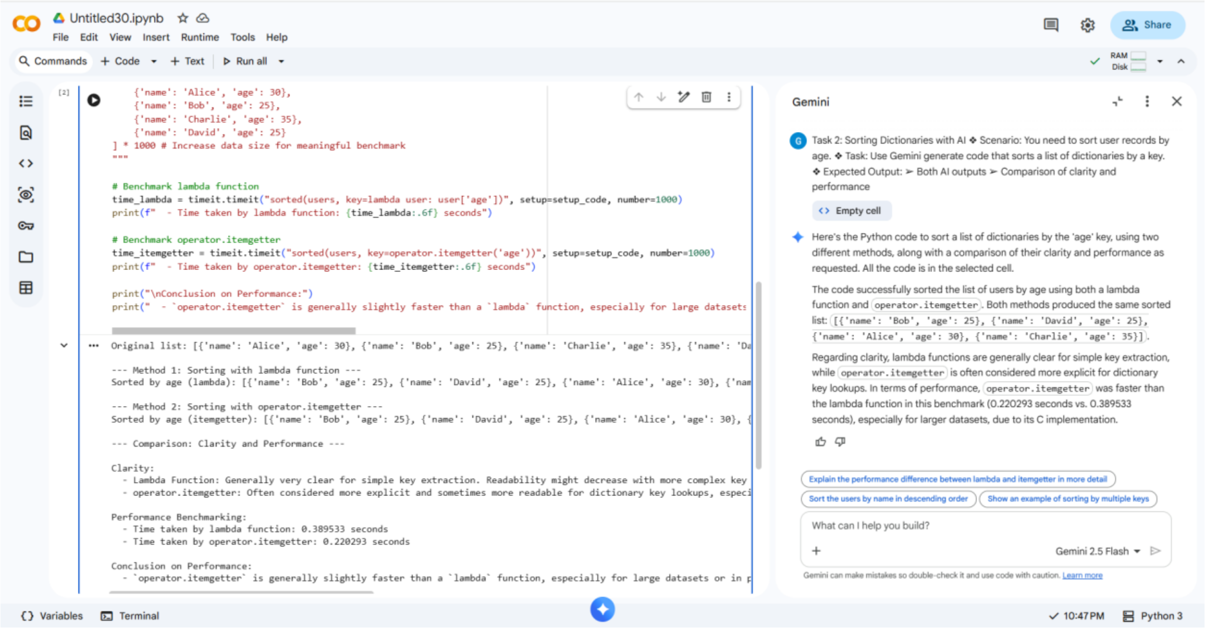
Task 2: Sorting Dictionaries with AI ❖ Scenario: You need to sort user records by age. ❖ Task: Use Gemini generate code that sorts a list of dictionaries by a key. ❖ Expected Output: ➢ Both AI outputs ➢ Comparison of clarity and performance

Here's the Python code to sort a list of dictionaries by the 'age' key, using two different methods, along with a comparison of their clarity and performance as requested. All the code is in the selected cell.

The code successfully sorted the list of users by age using both a lambda function and `operator.itemgetter`. Both methods produced the same sorted list: `[{'name': 'Bob', 'age': 25}, {'name': 'David', 'age': 25}, {'name': 'Alice', 'age': 30}, {'name': 'Charlie', 'age': 35}]`.

Regarding clarity, lambda functions are generally clear for simple key extraction, while `operator.itemgetter` is often considered more explicit for dictionary key lookups. In terms of performance, `operator.itemgetter` was faster than the lambda function in this benchmark (0.220293 seconds vs. 0.389533 seconds), especially for larger datasets, due to its C implementation.

Gemini can make mistakes so double-check it and use code with caution. Learn more


Untitled30.ipynb
File Edit View Insert Runtime Tools Help
Commands + Code + Text Run all
Share
RAM Disk
Gemini
Empty cell
Explain the performance difference between lambda and itemgetter in more detail
Sort the users by name in descending order
Show an example of sorting by multiple keys
What can I help you build?
Gemini 2.5 Flash
Gemini can make mistakes so double-check it and use code with caution. Learn more
Variables Terminal
10:47 PM
Python 3

```
    {'name': 'Alice', 'age': 30},
    {'name': 'Bob', 'age': 25},
    {'name': 'Charlie', 'age': 35},
    {'name': 'David', 'age': 25}
] * 1000 # Increase data size for meaningful benchmark
"""

# Benchmark lambda function
time_lambda = timeit.timeit("sorted(users, key=lambda user: user['age'])", setup=setup_code, number=1000)
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=1000)
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': 'Da

--- Method 1: Sorting with lambda function ---
Sorted by age (lambda): [{'name': 'Bob', 'age': 25}, {'name': 'David', 'age': 25}, {'name': 'Alice', 'age': 30}, {'nam

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

--- Comparison: Clarity and Performance ---

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

Performance Benchmarking:
  - 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 p
```

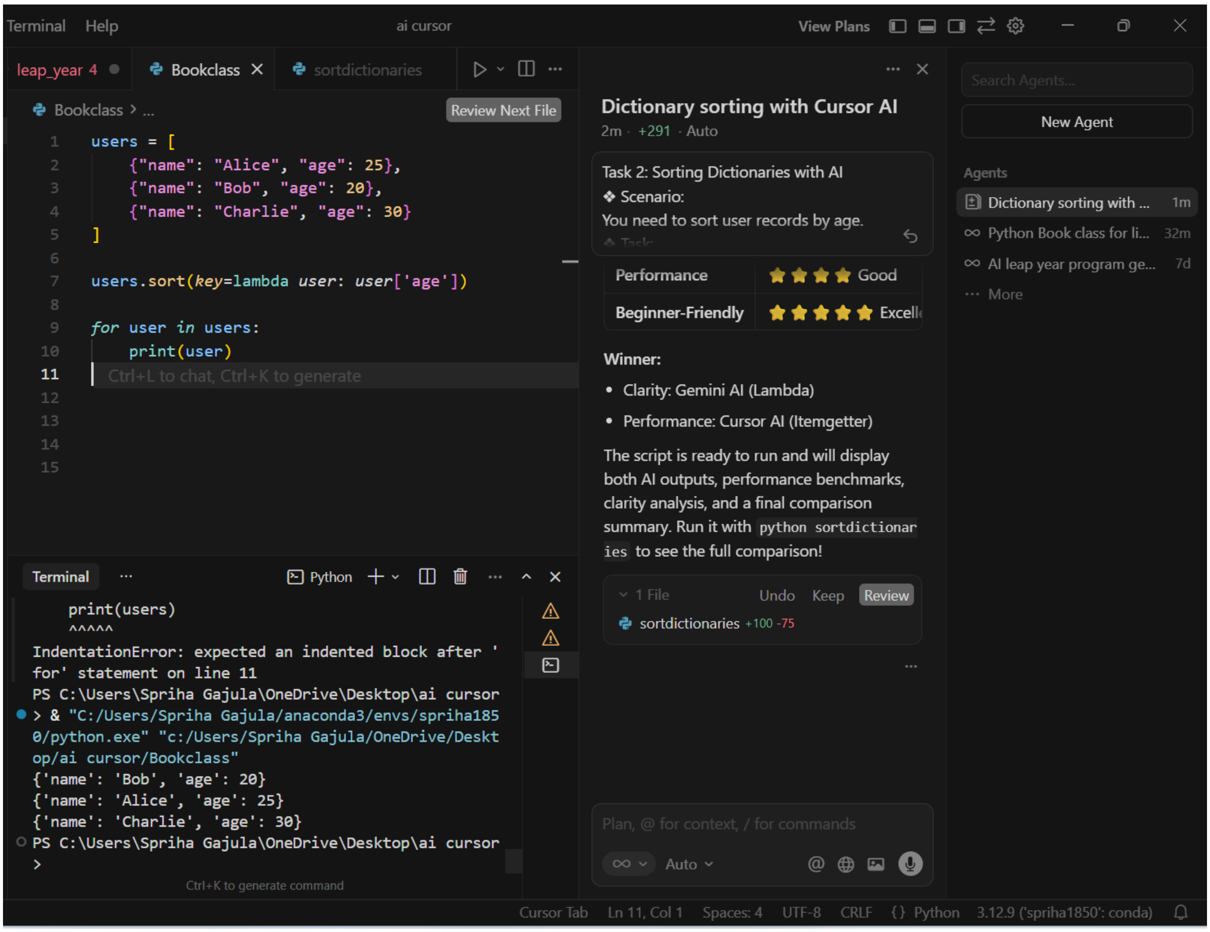
Task 2: Sorting Dictionaries with AI ❖ Scenario: You need to sort user records by age. ❖ Task: Use Gemini generate code that sorts a list of dictionaries by a key. ❖ Expected Output: ➢ Both AI outputs ➢ Comparison of clarity and performance

Here's the Python code to sort a list of dictionaries by the 'age' key, using two different methods, along with a comparison of their clarity and performance as requested. All the code is in the selected cell.

The code successfully sorted the list of users by age using both a lambda function and `operator.itemgetter`. Both methods produced the same sorted list: `[{'name': 'Bob', 'age': 25}, {'name': 'David', 'age': 25}, {'name': 'Alice', 'age': 30}, {'name': 'Charlie', 'age': 35}]`.

Regarding clarity, lambda functions are generally clear for simple key extraction, while `operator.itemgetter` is often considered more explicit for dictionary key lookups. In terms of performance, `operator.itemgetter` was faster than the lambda function in this benchmark (0.220293 seconds vs. 0.389533 seconds), especially for larger datasets, due to its C implementation.

CODE FROM CURSOR AI:


Terminal Help
ai cursor
View Plans
leap_year 4
Bookclass
sortdictionaries
Bookclass > ...
Review Next File
Ctrl+L to chat, Ctrl+K to generate
Terminal
Python
Ctrl+K to generate command
Search Agents...
New Agent
Agents
Dictionary sorting with ... 1m
Python Book class for li... 32m
AI leap year program ge... 7d
More
1 File
Undo Keep Review
sortdictionaries +100 -75
Plan, @ for context, / for commands
Auto
Cursor Tab Ln 11, Col 1 Spaces: 4 UTF-8 CRLF {} Python 3.12.9 ('spriha1850': conda)

```
users = [
    {"name": "Alice", "age": 25},
    {"name": "Bob", "age": 20},
    {"name": "Charlie", "age": 30}
]

users.sort(key=lambda user: user['age'])

for user in users:
    print(user)
```

```
    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
0/python.exe" "c:/Users/Spriha Gajula/OneDrive/Deskt
op/ai cursor/Bookclass"
{'name': 'Bob', 'age': 20}
{'name': 'Alice', 'age': 25}
{'name': 'Charlie', 'age': 30}
PS C:\Users\Spriha Gajula\OneDrive\Desktop\ai cursor
>
```

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	★★★★★ Excell

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!

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.

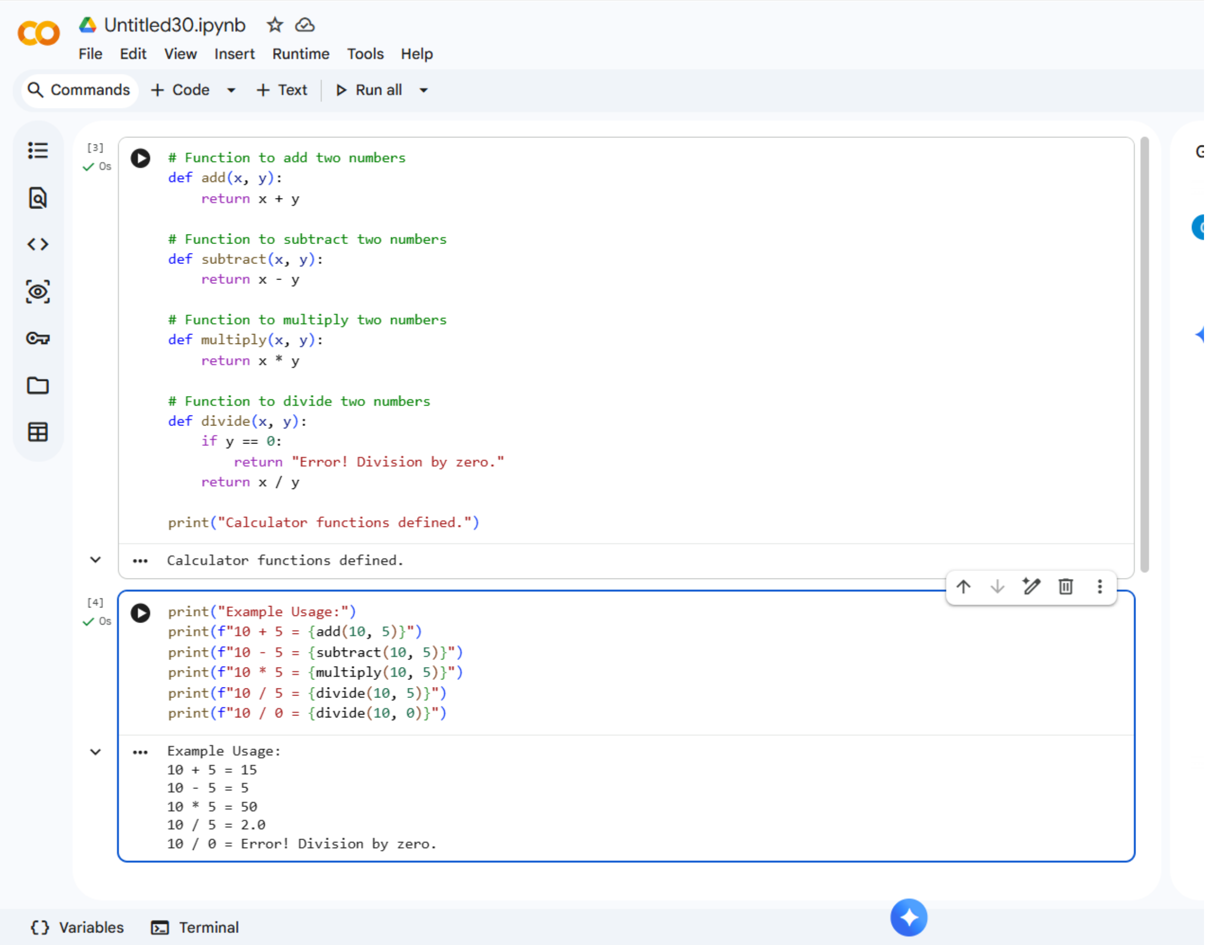
❖ Expected Output:

➢ Calculator code

➢ AI explanation

➢ Screenshot

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.
```

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.

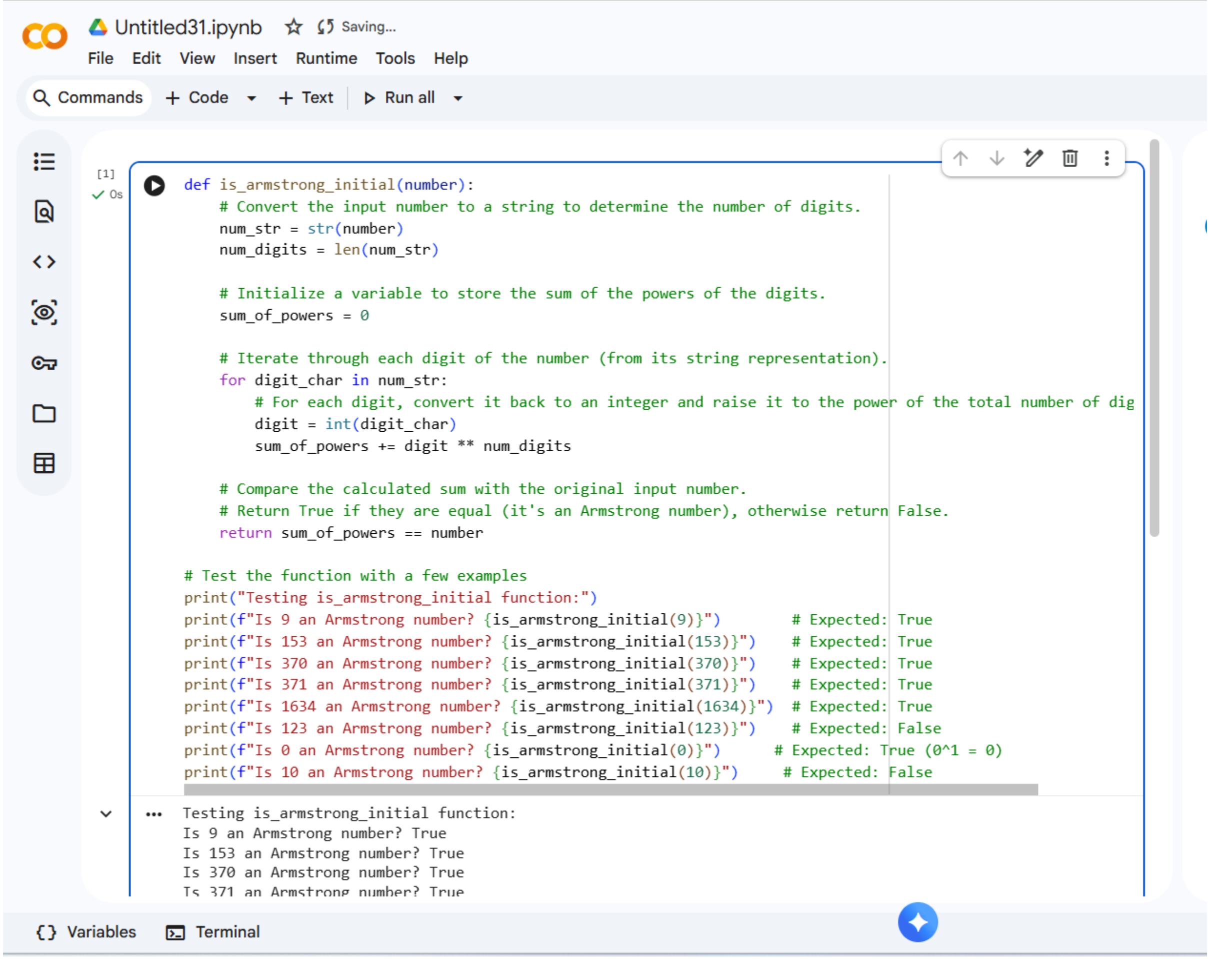
❖ Expected Output:

➢ 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

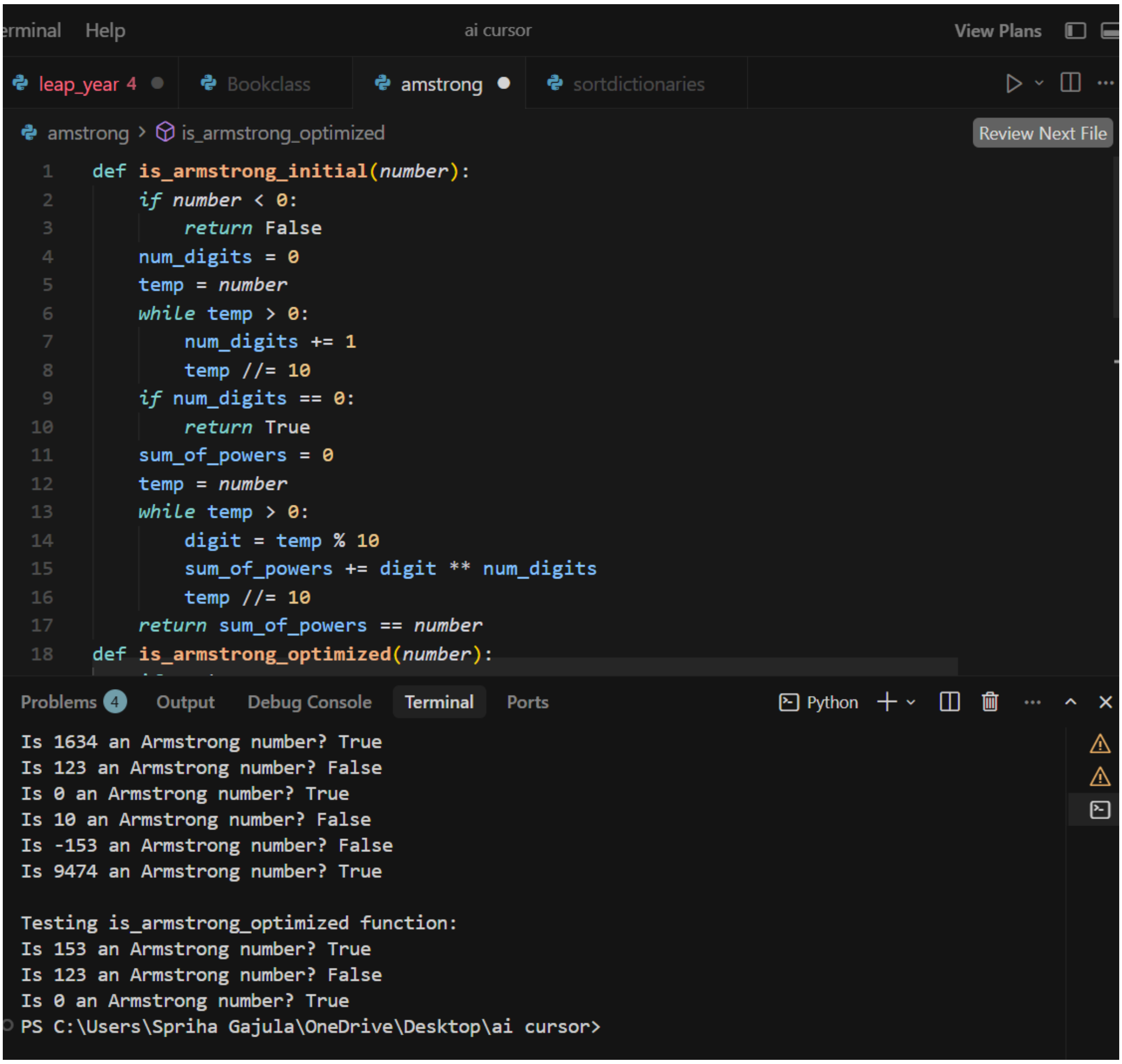
CODE USING GEMINI:



```python
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 dig
        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
```

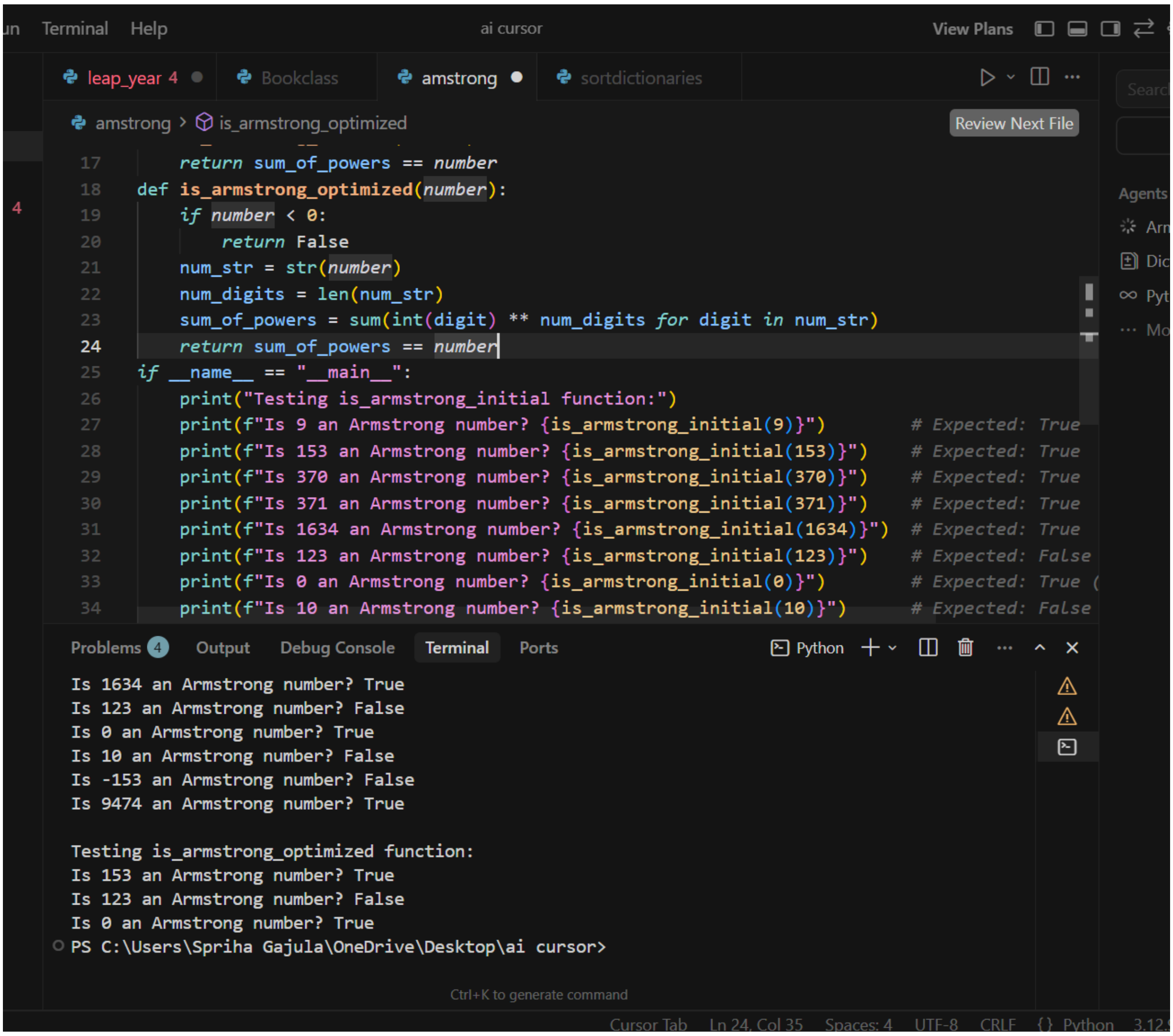
```
Testing is_armstrong_initial function:
Is 9 an Armstrong number? True
Is 153 an Armstrong number? True
Is 370 an Armstrong number? True
Is 371 an Armstrong number? True
```

IMPROVEMENT CODE OF CURSOR AI:



```python
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
def is_armstrong_optimized(number):
```

```
Is 1634 an Armstrong number? True
Is 123 an Armstrong number? False
Is 0 an Armstrong number? True
Is 10 an Armstrong number? False
Is -153 an Armstrong number? False
Is 9474 an Armstrong number? True

Testing is_armstrong_optimized function:
Is 153 an Armstrong number? True
Is 123 an Armstrong number? False
Is 0 an Armstrong number? True
PS C:\Users\Spriha Gajula\OneDrive\Desktop\ai cursor>
```



```
    return sum_of_powers == number
def is_armstrong_optimized(number):
    if number < 0:
        return False
    num_str = str(number)
    num_digits = len(num_str)
    sum_of_powers = sum(int(digit) ** num_digits for digit in num_str)
    return sum_of_powers == number
if __name__ == "__main__":
    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 (
    print(f"Is 10 an Armstrong number? {is_armstrong_initial(10)}")      # Expected: False
```

```
Is 1634 an Armstrong number? True
Is 123 an Armstrong number? False
Is 0 an Armstrong number? True
Is 10 an Armstrong number? False
Is -153 an Armstrong number? False
Is 9474 an Armstrong number? True

Testing is_armstrong_optimized function:
Is 153 an Armstrong number? True
Is 123 an Armstrong number? False
Is 0 an Armstrong number? True
PS C:\Users\Spriha Gajula\OneDrive\Desktop\ai cursor>
```