AI ASSISTED CODING

ASSIGNMENT 2.2

NAME : S RUCHITHA

HALLTICKET.NO : 2403A52316

BATCH.NO : 01 YR

AND SEM : 2ND AND 3RD

TASK DESCRIPTION 1:

QUESTION:

Use Gemini in Colab to write a function that filters out all negative numbers from a given list

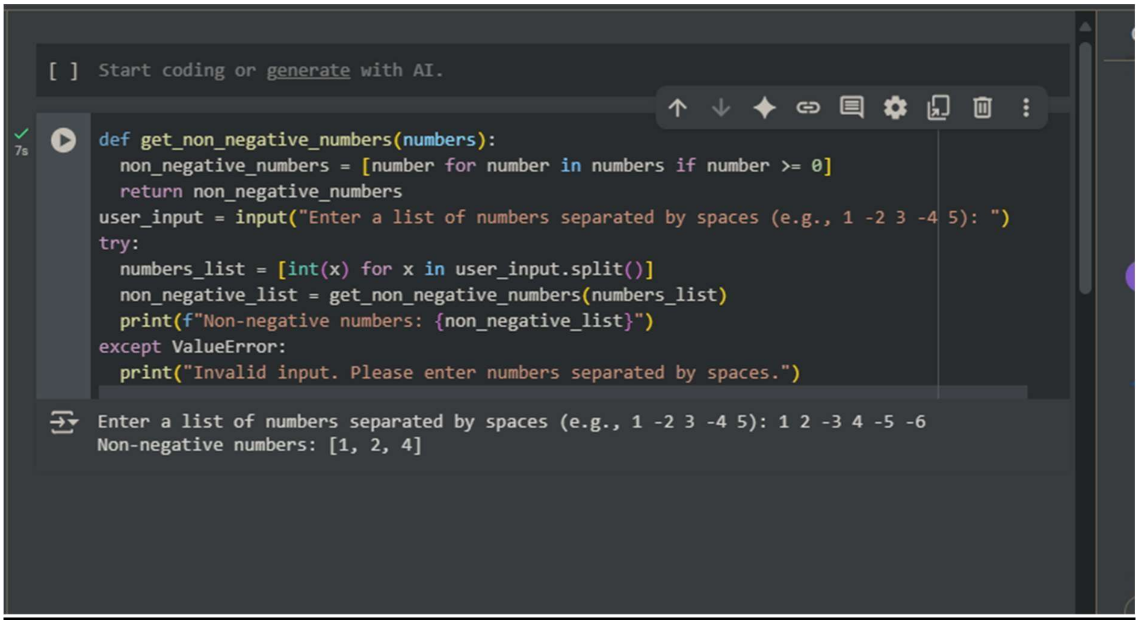
PROMPT:

write a code in python in which input of list is taken from user and create a function which separates the negative numbers from list and return filtered list

EXPECTED OUTPUT:

Functional code with before/after input and output shown in Colab, plus a screenshot

CODE WITH OUTPUT:



```
def get_non_negative_numbers(numbers):
  non_negative_numbers = [number for number in numbers if number >= 0]
  return non_negative_numbers
user_input = input("Enter a list of numbers separated by spaces (e.g., 1 -2 3 -4 5): ")
try:
  numbers_list = [int(x) for x in user_input.split()]
  non_negative_list = get_non_negative_numbers(numbers_list)
  print(f"Non-negative numbers: {non_negative_list}")
except ValueError:
  print("Invalid input. Please enter numbers separated by spaces.")
```

```
Enter a list of numbers separated by spaces (e.g., 1 -2 3 -4 5): 1 2 -3 4 -5 -6
Non-negative numbers: [1, 2, 4]
```

OBSERVATION: IT IS OBSERVED THAT CODE IS GENERATED WITH NO TIME AND WITH BETTER OUTPUT

EXPLAINATION:

This code takes a list of numbers entered by the user, separates them into negative and nonnegative numbers, and then prints both lists.

TASK DESCRIPTION 2:

QUESTION:

Ask Gemini to generate code that reads a text file and counts the frequency of each word. Then ask it to explain the code

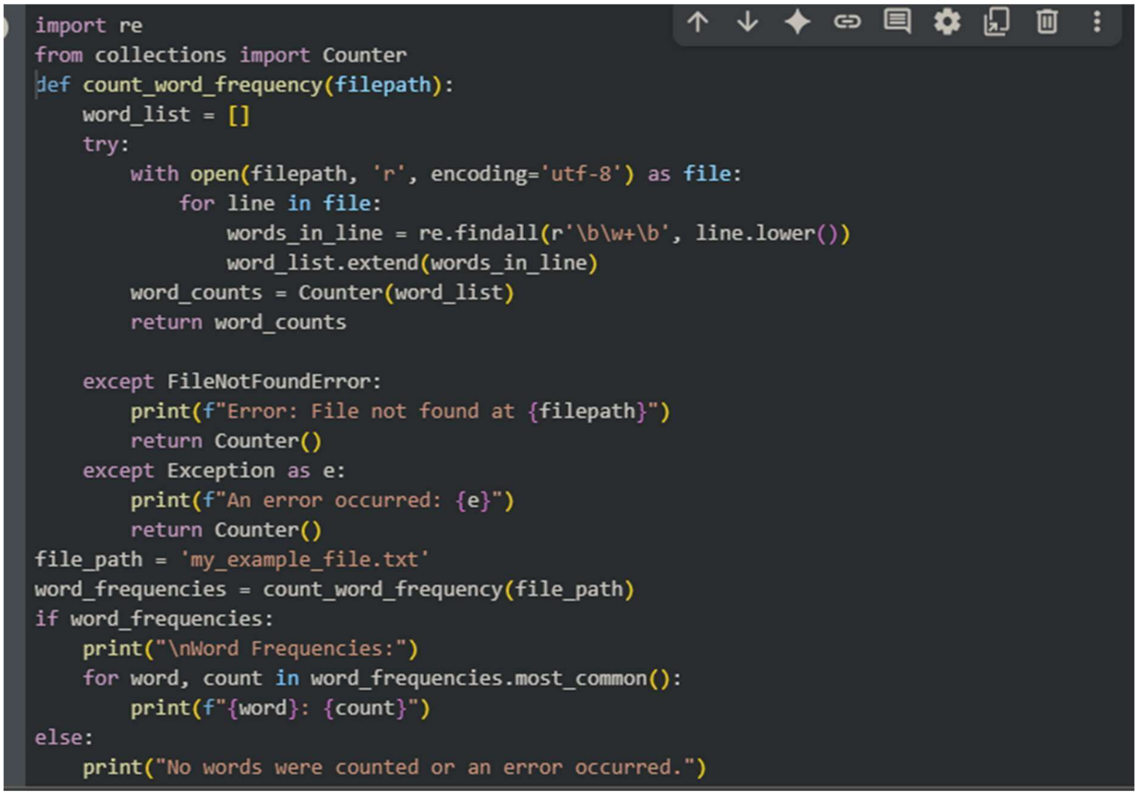
PROMPT:

generate code in python that reads a text file and counts the frequency of each word

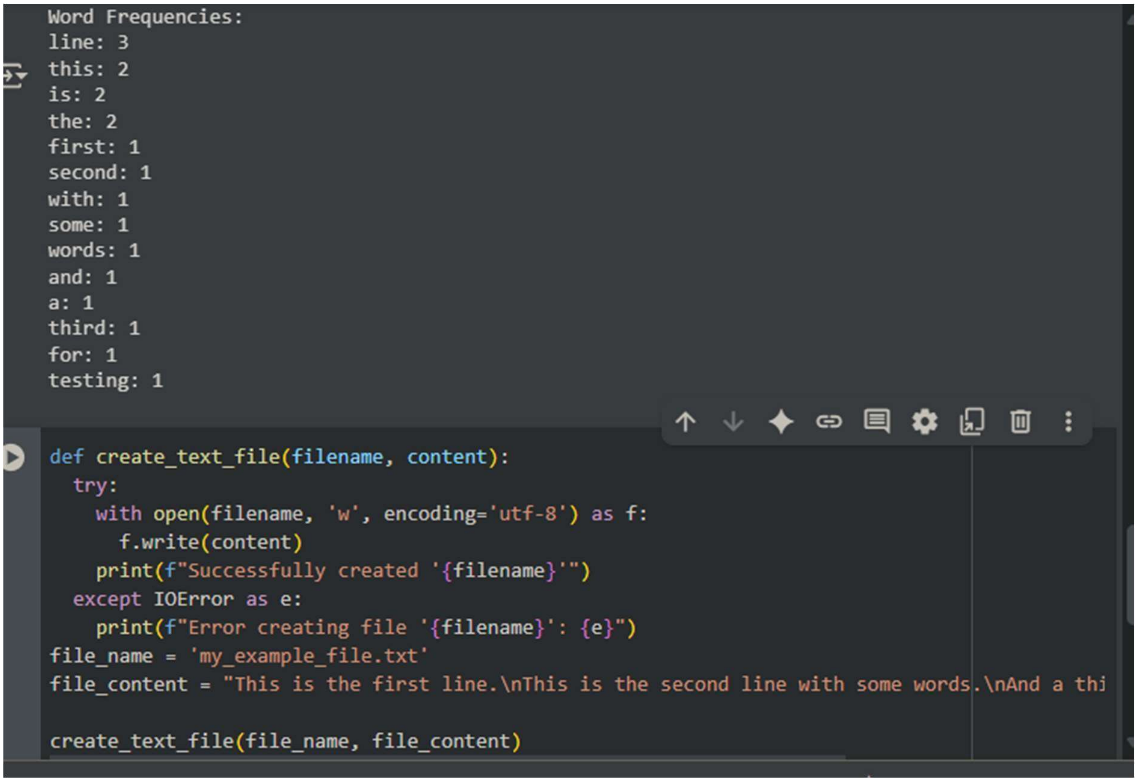
EXPECTED OUTPUT:

Working code and Gemini's explanation side-by-side

CODE WITH OUTPUT:



```
import re
from collections import Counter
def count_word_frequency(filepath):
    word_list = []
    try:
        with open(filepath, 'r', encoding='utf-8') as file:
            for line in file:
                words_in_line = re.findall(r'\b\w+\b', line.lower())
                word_list.extend(words_in_line)
        word_counts = Counter(word_list)
        return word_counts

    except FileNotFoundError:
        print(f"Error: File not found at {filepath}")
        return Counter()
    except Exception as e:
        print(f"An error occurred: {e}")
        return Counter()
file_path = 'my_example_file.txt'
word_frequencies = count_word_frequency(file_path)
if word_frequencies:
    print("\nWord Frequencies:")
    for word, count in word_frequencies.most_common():
        print(f"{word}: {count}")
else:
    print("No words were counted or an error occurred.")
```



```
Word Frequencies:
line: 3
this: 2
is: 2
the: 2
first: 1
second: 1
with: 1
some: 1
words: 1
and: 1
a: 1
third: 1
for: 1
testing: 1
```

```
def create_text_file(filename, content):
  try:
    with open(filename, 'w', encoding='utf-8') as f:
      f.write(content)
    print(f"Successfully created '{filename}'")
  except IOError as e:
    print(f"Error creating file '{filename}': {e}")
file_name = 'my_example_file.txt'
file_content = "This is the first line.\nThis is the second line with some words.\nAnd a thi

create_text_file(file_name, file_content)
```

OBSERVATION:

The provided code is a good balance of simplicity and efficiency for creating a text file and counting word frequencies. It uses standard, well-optimized Python libraries and techniques. While more advanced methods exist for handling truly massive datasets, this code is efficient and perfectly suitable for many common text processing tasks.

EXPLAINATION:

In summary, this code provides a robust way to read a text file, extract words, and count how

many times each word appears, handling potential file-related errors.

TASK DESCRIPTION 3:

QUESTION:

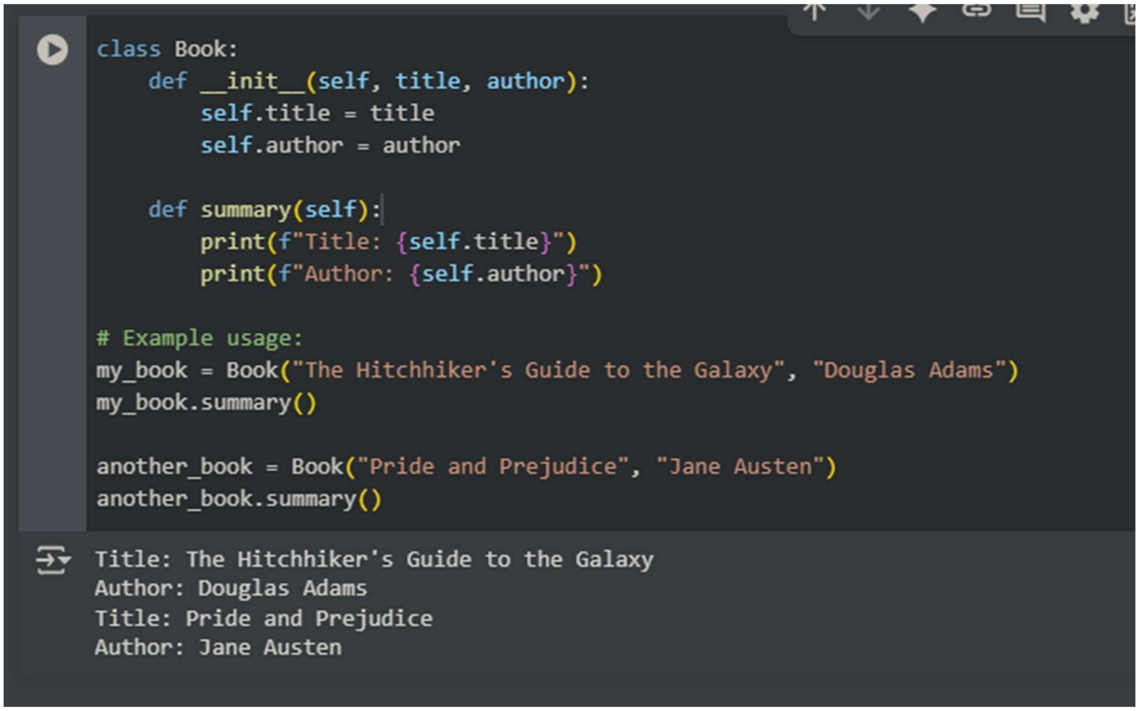
Use Cursor AI to generate a Python class called Book with attributes title, author, and a method summary() that prints the details.

PROMPT IN GEMINI:

generate a code in Python class called Book with attributes title, author, and a method summary() that prints the details.

EXPECTED OUTPUT:

Screenshot comparisons and student commentary on code clarity and performance

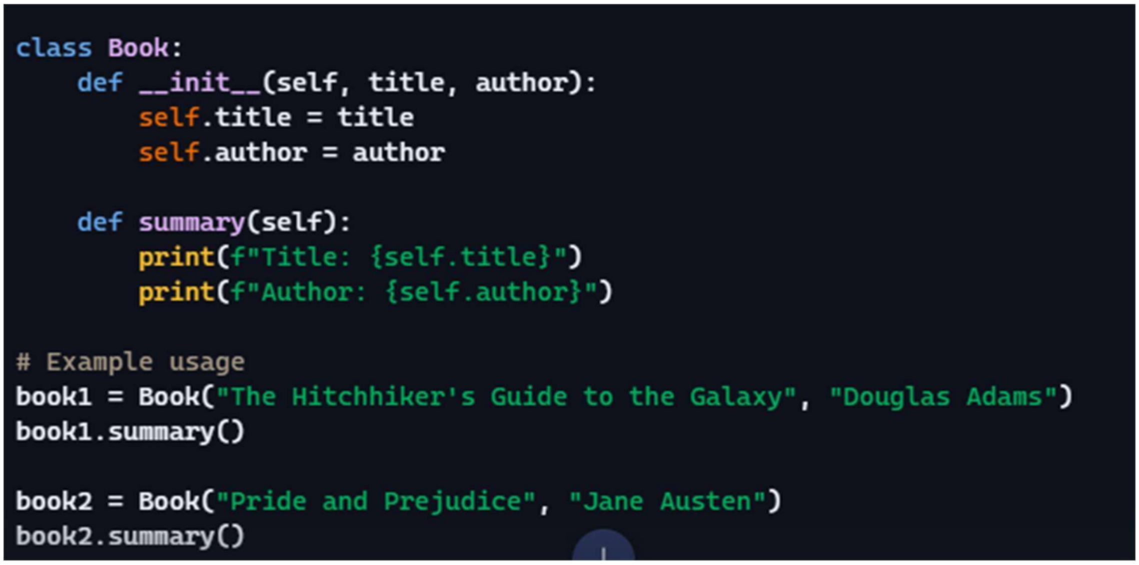


```python
class Book:
    def __init__(self, title, author):
        self.title = title
        self.author = author

    def summary(self):
        print(f"Title: {self.title}")
        print(f"Author: {self.author}")

# Example usage:
my_book = Book("The Hitchhiker's Guide to the Galaxy", "Douglas Adams")
my_book.summary()

another_book = Book("Pride and Prejudice", "Jane Austen")
another_book.summary()
```

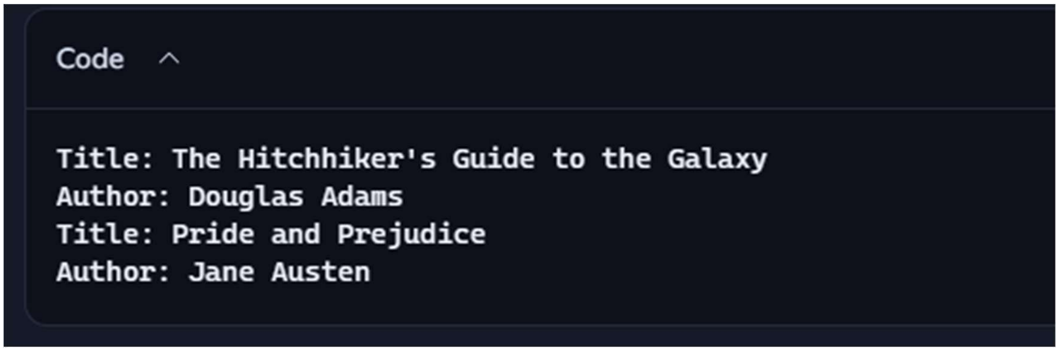
```
Title: The Hitchhiker's Guide to the Galaxy
Author: Douglas Adams
Title: Pride and Prejudice
Author: Jane Austen
```

PROMPT IN COPILOT:

generate a code in Python class called Book with attributes title, author, and a method summary() that prints the details.



```
class Book:
    def __init__(self, title, author):
        self.title = title
        self.author = author

    def summary(self):
        print(f"Title: {self.title}")
        print(f"Author: {self.author}")

# Example usage
book1 = Book("The Hitchhiker's Guide to the Galaxy", "Douglas Adams")
book1.summary()

book2 = Book("Pride and Prejudice", "Jane Austen")
book2.summary()
```



```
Title: The Hitchhiker's Guide to the Galaxy
Author: Douglas Adams
Title: Pride and Prejudice
Author: Jane Austen
```

COMMENTRY ON CODE CLARITY AND PERFORMANCE:

- Clarity: COPILOT’s code is like a simple, well-written instruction manual. I can quickly understand how to make a "book" object and get its basic information. There are no confusing steps or jargon.
- Performance: It's like a lightning-fast little machine. When I ask it to create a book or tell you it’s summary, it does it almost instantly without breaking a sweat. It's not doing any unnecessary work.

EXPLAINATION:

- This code defines a template for creating "Book" objects, each holding its own title and author, and provides a way to display that information.

TASK DESCRIPTION 4:

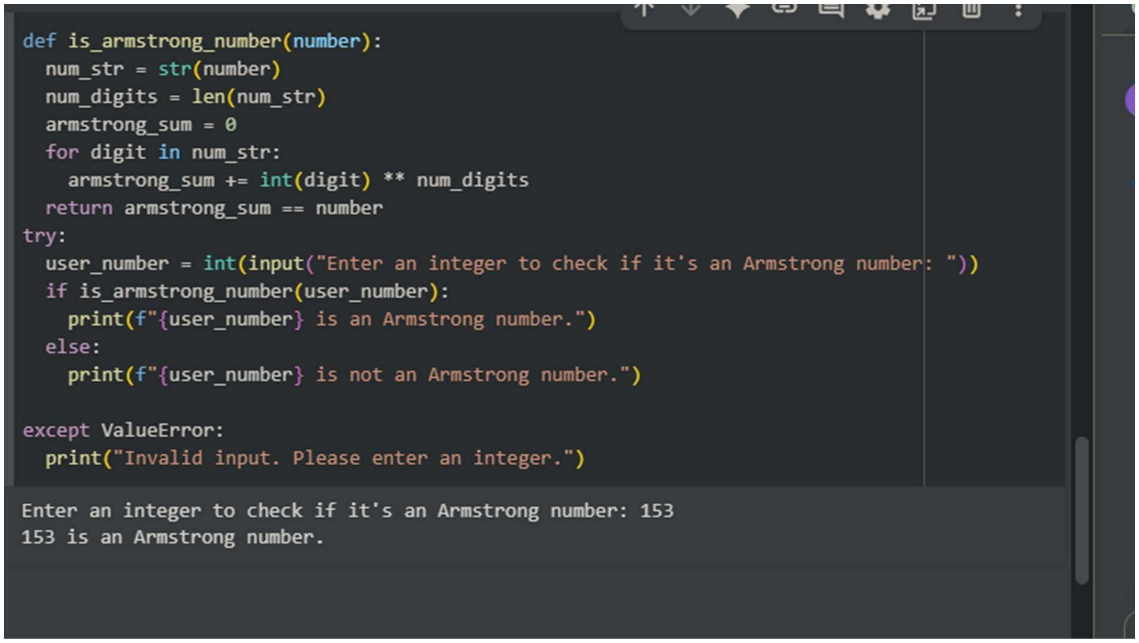
QUESTION:

Ask Gemini to write a program that checks whether a number is an Armstrong number, and then modify it using Cursor AI to improve performance or structure

EXPECTED OUTPUT:

Two versions of the code with screenshots, and a summary of what changes were made by Cursor. PROMPT IN GEMINI:

generate a code when a number is given it should check whether a number is an Armstrong number

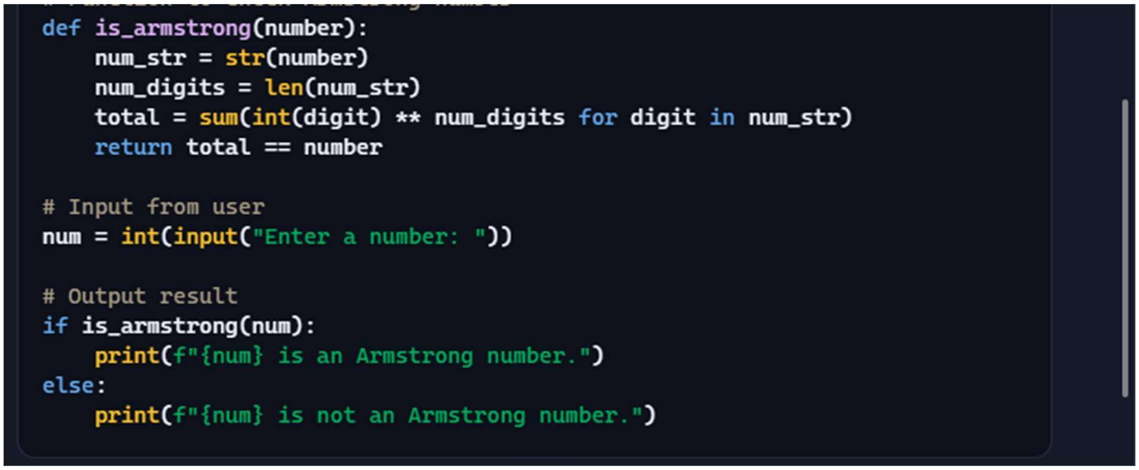


```python
def is_armstrong_number(number):
  num_str = str(number)
  num_digits = len(num_str)
  armstrong_sum = 0
  for digit in num_str:
    armstrong_sum += int(digit) ** num_digits
  return armstrong_sum == number
try:
  user_number = int(input("Enter an integer to check if it's an Armstrong number: "))
  if is_armstrong_number(user_number):
    print(f"{user_number} is an Armstrong number.")
  else:
    print(f"{user_number} is not an Armstrong number.")

except ValueError:
  print("Invalid input. Please enter an integer.")
```

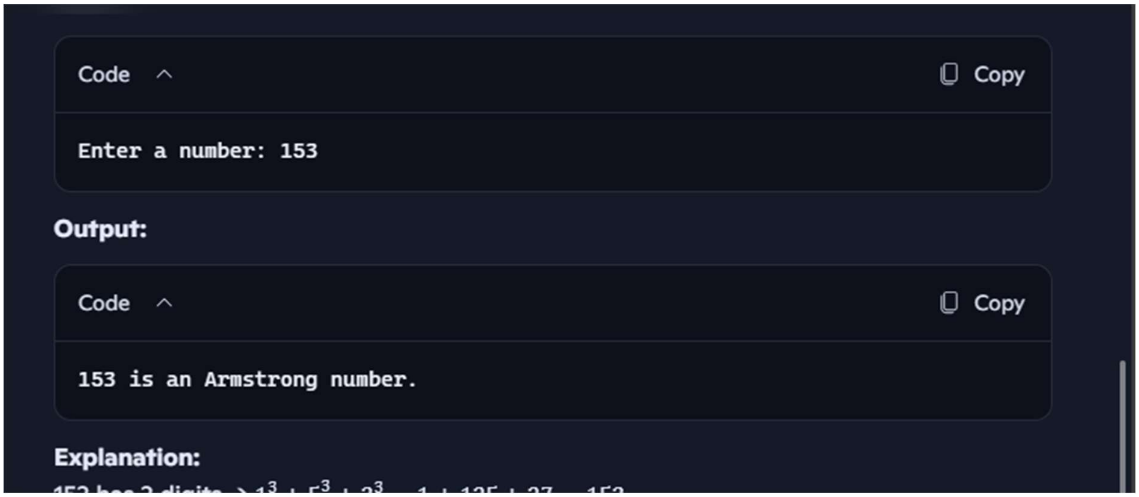
```
Enter an integer to check if it's an Armstrong number: 153
153 is an Armstrong number.
```

PROMPT IN COPILOT:

generate a code when a number is given it should check whether a number is an Armstrong number



```
def is_armstrong(number):
    num_str = str(number)
    num_digits = len(num_str)
    total = sum(int(digit) ** num_digits for digit in num_str)
    return total == number

# Input from user
num = int(input("Enter a number: "))

# Output result
if is_armstrong(num):
    print(f"{num} is an Armstrong number.")
else:
    print(f"{num} is not an Armstrong number.")
```



```
Enter a number: 153
```

Output:

```
153 is an Armstrong number.
```

Explanation:

SUMMARY:

- Clarity: COPILOT’s code is like a clear recipe for checking Armstrong numbers. Most of the steps are obvious, and even the slightly more compact calculation step is a common way to do things efficiently in Python.
- Performance: It's like a quick little calculator for this specific job. For the numbers you'd usually test, it gives you the answer almost instantly. It handles bigger numbers gracefully; it just takes a tiny bit longer as the number gets longer, but it's not like it suddenly grinds to a halt. For performance it is Gemini AI which performed in very good manner.

TASK DESCRIPTION 5:

QUESTION:

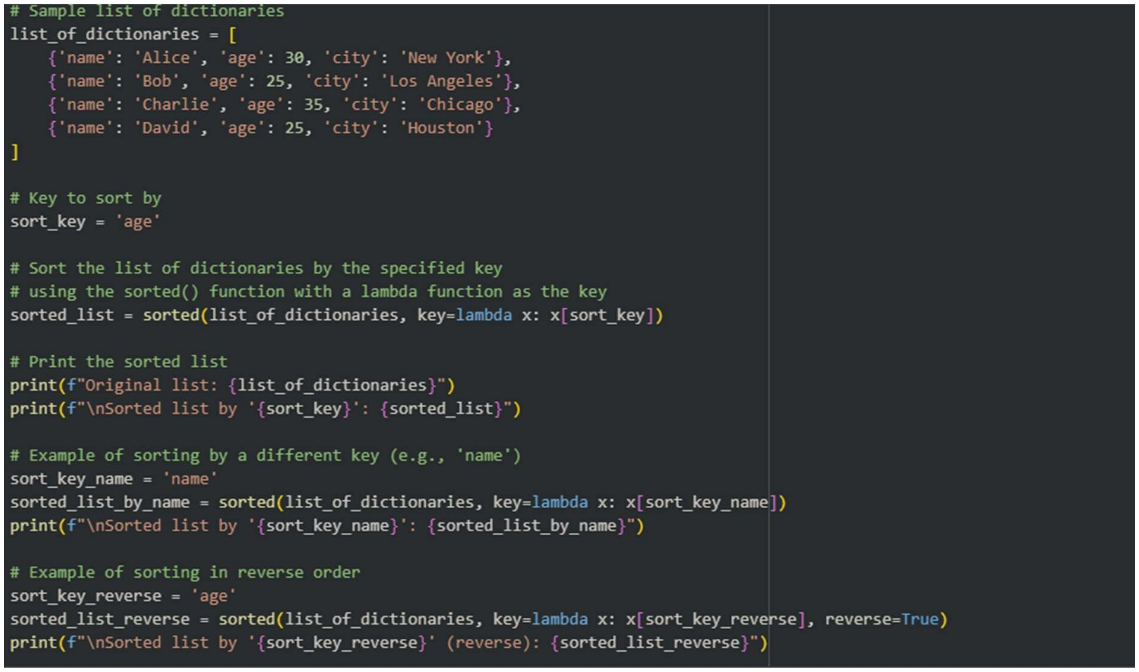
Use both Gemini and Cursor AI to generate code for sorting a list of dictionaries by a specific key (e.g., age)

PROMPT IN GEMINI: generate a code for sorting a list of dictionaries by a specific key.

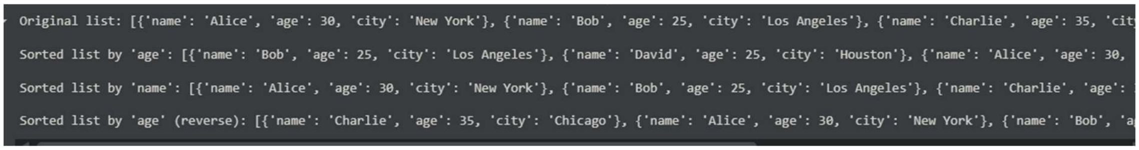
EXPECTED OUTPUT:

Screenshot comparisons and student commentary on code clarity and performance

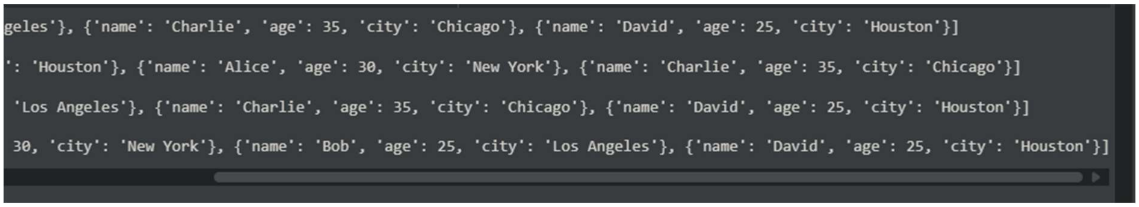
CODE WITH OUTPUT IN GEMINI:



```python
# Sample list of dictionaries
list_of_dictionaries = [
    {'name': 'Alice', 'age': 30, 'city': 'New York'},
    {'name': 'Bob', 'age': 25, 'city': 'Los Angeles'},
    {'name': 'Charlie', 'age': 35, 'city': 'Chicago'},
    {'name': 'David', 'age': 25, 'city': 'Houston'}
]

# Key to sort by
sort_key = 'age'

# Sort the list of dictionaries by the specified key
# using the sorted() function with a lambda function as the key
sorted_list = sorted(list_of_dictionaries, key=lambda x: x[sort_key])

# Print the sorted list
print(f"Original list: {list_of_dictionaries}")
print(f"\nSorted list by '{sort_key}': {sorted_list}")

# Example of sorting by a different key (e.g., 'name')
sort_key_name = 'name'
sorted_list_by_name = sorted(list_of_dictionaries, key=lambda x: x[sort_key_name])
print(f"\nSorted list by '{sort_key_name}': {sorted_list_by_name}")

# Example of sorting in reverse order
sort_key_reverse = 'age'
sorted_list_reverse = sorted(list_of_dictionaries, key=lambda x: x[sort_key_reverse], reverse=True)
print(f"\nSorted list by '{sort_key_reverse}' (reverse): {sorted_list_reverse}")
```

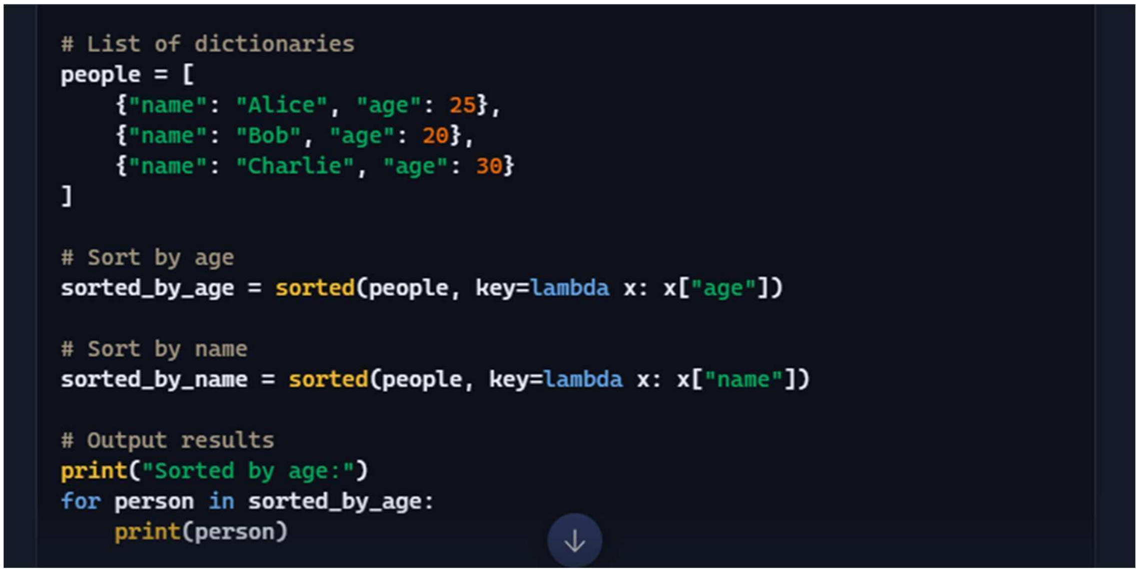


```
Original list: [{'name': 'Alice', 'age': 30, 'city': 'New York'}, {'name': 'Bob', 'age': 25, 'city': 'Los Angeles'}, {'name': 'Charlie', 'age': 35, 'cit

Sorted list by 'age': [{'name': 'Bob', 'age': 25, 'city': 'Los Angeles'}, {'name': 'David', 'age': 25, 'city': 'Houston'}, {'name': 'Alice', 'age': 30,

Sorted list by 'name': [{'name': 'Alice', 'age': 30, 'city': 'New York'}, {'name': 'Bob', 'age': 25, 'city': 'Los Angeles'}, {'name': 'Charlie', 'age':

Sorted list by 'age' (reverse): [{'name': 'Charlie', 'age': 35, 'city': 'Chicago'}, {'name': 'Alice', 'age': 30, 'city': 'New York'}, {'name': 'Bob', 'a
```

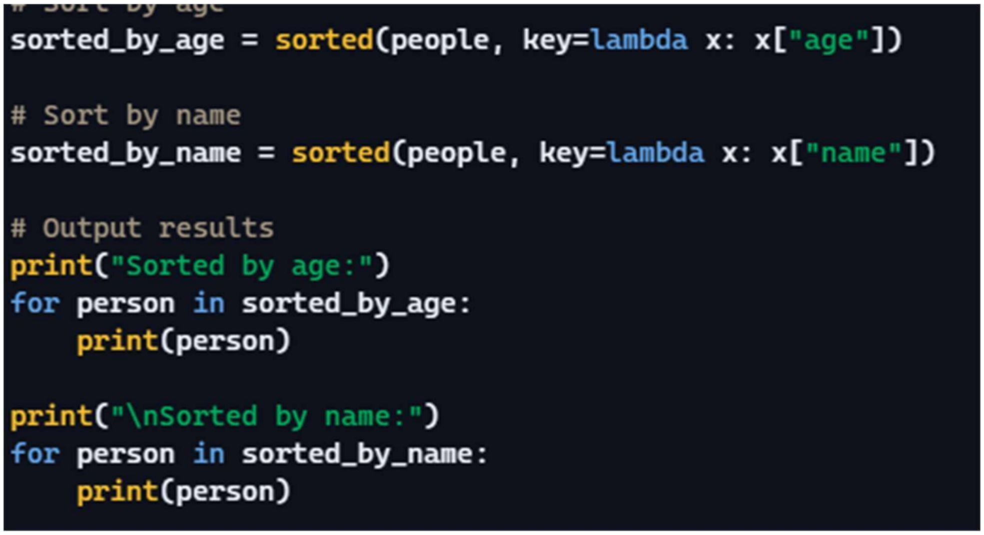


```
geles'}, {'name': 'Charlie', 'age': 35, 'city': 'Chicago'}, {'name': 'David', 'age': 25, 'city': 'Houston'}]

': 'Houston'}, {'name': 'Alice', 'age': 30, 'city': 'New York'}, {'name': 'Charlie', 'age': 35, 'city': 'Chicago'}]

 'Los Angeles'}, {'name': 'Charlie', 'age': 35, 'city': 'Chicago'}, {'name': 'David', 'age': 25, 'city': 'Houston'}]

 30, 'city': 'New York'}, {'name': 'Bob', 'age': 25, 'city': 'Los Angeles'}, {'name': 'David', 'age': 25, 'city': 'Houston'}]
```

CODE AND OUTPUT WITH COPILOT:



```python
# List of dictionaries
people = [
    {"name": "Alice", "age": 25},
    {"name": "Bob", "age": 20},
    {"name": "Charlie", "age": 30}
]

# Sort by age
sorted_by_age = sorted(people, key=lambda x: x["age"])

# Sort by name
sorted_by_name = sorted(people, key=lambda x: x["name"])

# Output results
print("Sorted by age:")
for person in sorted_by_age:
    print(person)
```



```python
# Sort by age
sorted_by_age = sorted(people, key=lambda x: x["age"])

# Sort by name
sorted_by_name = sorted(people, key=lambda x: x["name"])

# Output results
print("Sorted by age:")
for person in sorted_by_age:
    print(person)

print("\nSorted by name:")
for person in sorted_by_name:
    print(person)
```

```python
# Sort by age
sorted_by_age = sorted(people, key=lambda x: x["age"])

# Sort by name
sorted_by_name = sorted(people, key=lambda x: x["name"])

# Output results
print("Sorted by age:")
for person in sorted_by_age:
    print(person)

print("\nSorted by name:")
for person in sorted_by_name:
    print(person)
```

SUMMARY:

- Clarity: COPILOT is like having a simple instruction: "Take this stack of papers (your list of dictionaries) and sort them based on the age written on each paper (the 'age' key)." It's direct and easy to follow. The lambda is just a quick way to peek at the right piece of information on each paper for sorting.
- Performance: It's like having a really efficient sorting machine. I give it the stack, and it quickly arranges them in the order I asked for, whether it's by age or name. It doesn't waste time and can handle quite a few papers before slowing down noticeably.