

-----AI Assisted Coding-----

----- Lab Assignment – 2.2-----

TASK DESCRIPTION 1 :

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

USED PROMPT :

Write a python program to take a list as input from the user and filter out all the negative numbers from the list and print all non negative numbers

```
[ ] # Get input from the user as a string
input_list_str = input("Enter a list of numbers separated by commas (e.g., 1,2,3,-4): ")

# Split the input string by commas and convert to a list of numbers
input_list = [float(num) for num in input_list_str.split(',')]

# Filter out negative numbers without using a function
filtered_list = []
for number in input_list:
    if number >= 0:
        filtered_list.append(number)

print("Original list:", input_list)
print("Filtered list (non-negative numbers):", filtered_list)
```

```
Enter a list of numbers separated by commas (e.g., 1,2,3,-4): 4,6,7,-7,-2
Original list: [4.0, 6.0, 7.0, -7.0, -2.0]
Filtered list (non-negative numbers): [4.0, 6.0, 7.0]
```

The above takes a list as input from the user and checks from negative numbers in the list if any negative numbers are found it separates the negative numbers and prints only the non-negative numbers in the form of a list.

TASK DESCRIPTION 2 :

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

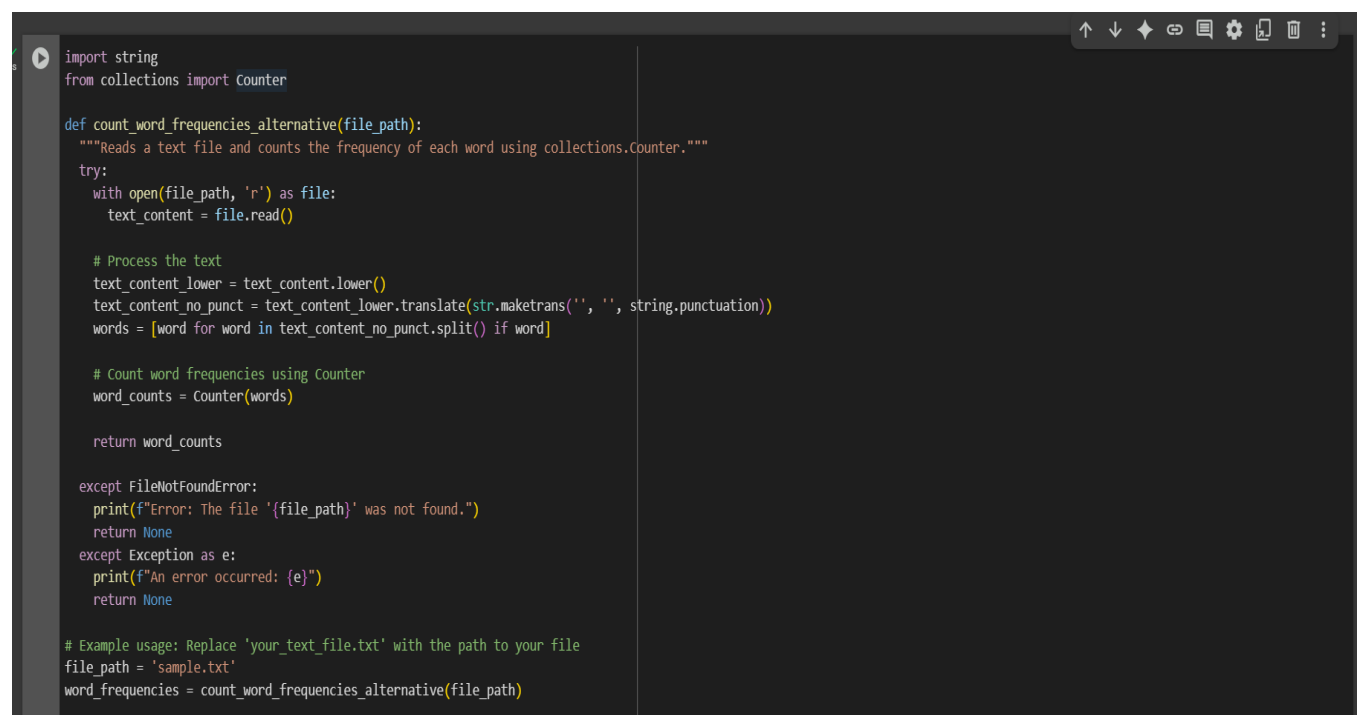
USED PROMPT :

generate a python code that reads a text file containing a paragraph and then count and print the frequency of every word in the file.

```
[3] # Create a dummy file for testing
file_path = 'sample.txt'
file_content = "This is a sample text file.\nIt contains some words for testing.\nWord word word."

with open(file_path, 'w') as file:
    file.write(file_content)

print(f"Created a dummy file named '{file_path}' with content:\n{file_content}")
```

A screenshot of a code editor with a dark theme. The editor shows a Python script that defines a function to count word frequencies in a text file. The script includes imports for 'string' and 'Counter' from 'collections'. The function 'count_word_frequencies_alternative' takes a file path as input, reads the file content, processes it by converting to lowercase and removing punctuation, and then uses 'Counter' to count the words. It also includes exception handling for 'FileNotFoundError' and a general 'Exception'. At the bottom, there is an example usage section that sets 'file_path' to 'sample.txt' and calls the function.

```
import string
from collections import Counter

def count_word_frequencies_alternative(file_path):
    """Reads a text file and counts the frequency of each word using collections.Counter."""
    try:
        with open(file_path, 'r') as file:
            text_content = file.read()

        # Process the text
        text_content_lower = text_content.lower()
        text_content_no_punct = text_content_lower.translate(str.maketrans('', '', string.punctuation))
        words = [word for word in text_content_no_punct.split() if word]

        # Count word frequencies using Counter
        word_counts = Counter(words)

        return word_counts

    except FileNotFoundError:
        print(f"Error: The file '{file_path}' was not found.")
        return None
    except Exception as e:
        print(f"An error occurred: {e}")
        return None

# Example usage: Replace 'your_text_file.txt' with the path to your file
file_path = 'sample.txt'
word_frequencies = count_word_frequencies_alternative(file_path)
```

C

```
if word_frequencies:
    print(f"Word frequencies in '{file_path}' (using Counter):")
    for word, count in word_frequencies.items():
        print(f"Word: {word}, Count: {count}")
```

```
Word frequencies in 'sample.txt' (using Counter):
Word: this, Count: 1
Word: is, Count: 1
Word: a, Count: 1
Word: sample, Count: 1
Word: text, Count: 1
Word: file, Count: 1
Word: it, Count: 1
Word: contains, Count: 1
Word: some, Count: 1
Word: words, Count: 1
Word: for, Count: 1
Word: testing, Count: 1
Word: word, Count: 3
```

CODE EXPLANATION :

- **Function Definition:** It defines a function `count_word_frequencies(file_path)` that is designed to take the path of a text file as input and return the frequency of each word found in that file.
- **File Reading (with error handling):** It attempts to open the specified `file_path` in read mode ('r') using a `try...except` block to gracefully handle potential `FileNotFoundError` or other exceptions during file operations. If the file is opened successfully, its entire content is read into a string variable `text_content`.
- **Text Cleaning (Lowercase & Punctuation Removal):** The `text_content` is converted to lowercase to ensure case-insensitivity in word counting. Punctuation characters are then removed from the text using the `translate()` method and `string.punctuation`.
- **Tokenization (Splitting into Words):** The cleaned text is split into a list of individual words (tokens) based on whitespace. A list comprehension is used to efficiently create this list, filtering out any empty strings that might result from multiple spaces.
- **Word Frequency Counting:** An empty dictionary `word_counts` is initialized. The code then iterates through the list of words. For each word, it checks if the word is already a key in the dictionary; if it is, the corresponding count is incremented, otherwise, the word is added as a new key with a count of 1.
- **Returning Results:** The function returns the `word_counts` dictionary containing the frequency of each unique word. If a `FileNotFoundError` or other exception occurred, it prints an error message and returns `None`.
- **Example Usage and Output:** The code then sets the `file_path` to 'sample.txt' and calls the `count_word_frequencies` function. If the function returns a dictionary

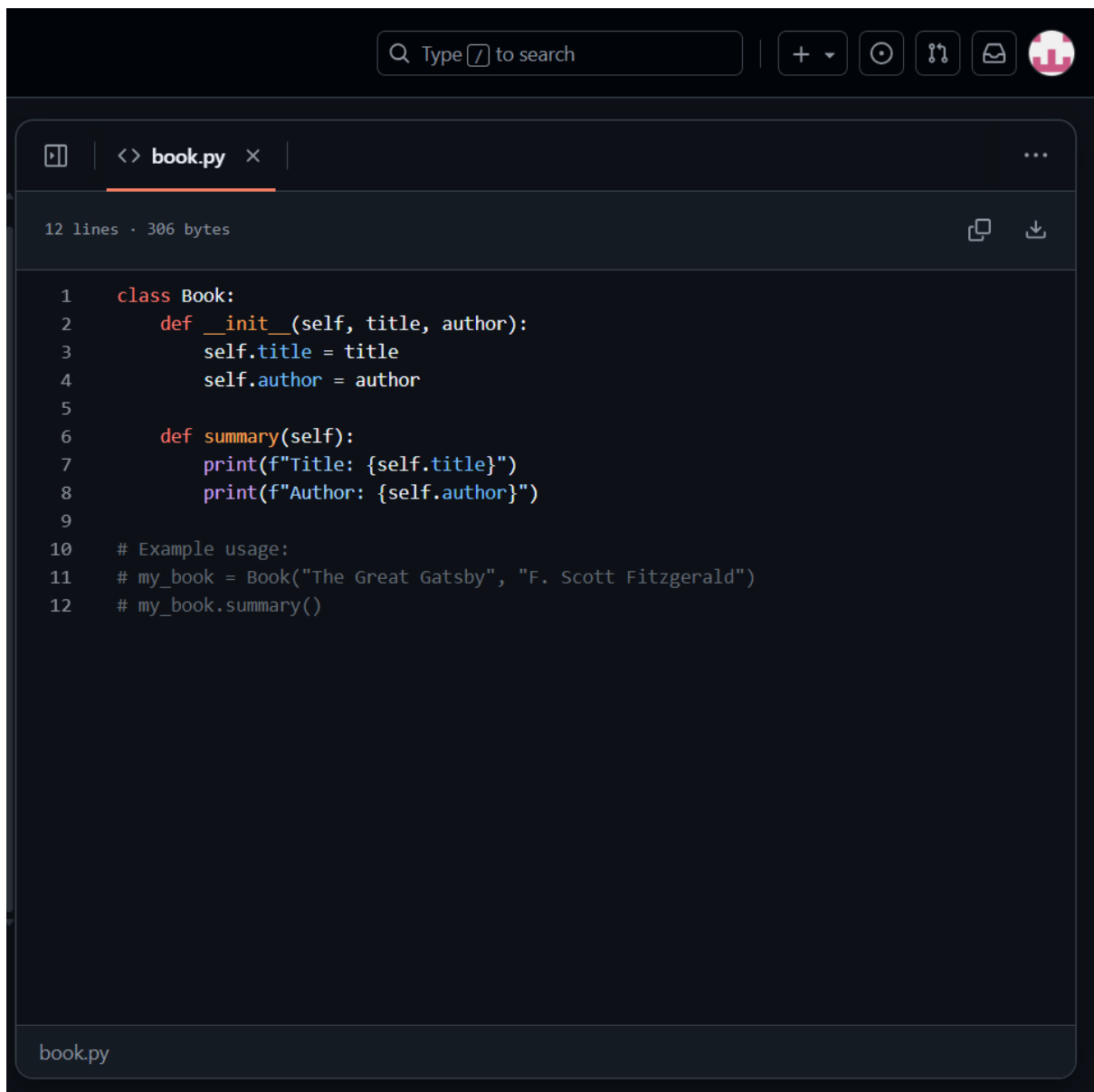
(meaning no file error), it iterates through the dictionary and prints each word along with its calculated frequency.

TASK DESCRIPTION 3 :

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

USED PROMPT :

generate a python class .name it as book with attributes title , author and a method summary() that prints the details



The screenshot shows the Cursor AI code editor interface. At the top, there is a search bar with the text "Type [7] to search" and several icons for file management and settings. Below the search bar, the editor window displays a file named "book.py" with 12 lines of code and 306 bytes. The code defines a Python class named "Book" with two methods: "__init__" and "summary". The "__init__" method takes "title" and "author" as arguments and assigns them to "self.title" and "self.author". The "summary" method prints the title and author. Below the class definition, there is a comment "# Example usage:" followed by two lines of code: "# my_book = Book('The Great Gatsby', 'F. Scott Fitzgerald')" and "# my_book.summary()". The editor window has a dark theme and a sidebar on the left showing the file structure.

```
1  class Book:
2      def __init__(self, title, author):
3          self.title = title
4          self.author = author
5
6      def summary(self):
7          print(f"Title: {self.title}")
8          print(f"Author: {self.author}")
9
10 # Example usage:
11 # my_book = Book("The Great Gatsby", "F. Scott Fitzgerald")
12 # my_book.summary()
```

- The code defines a **Book** class. A class is like a blueprint for creating objects that represent books.
- When you create a new book, you give it a **title** and an **author**. These are saved inside the object.
- The **summary** method prints out the book's title and author.
- At the bottom, there's an example showing how you could use the class:
 - You create a book called "**The Great Gatsby**" by "**F. Scott Fitzgerald**".
 - Then, you call the **summary** method to print out its details

TASK DESCRIPTION 4 :

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.

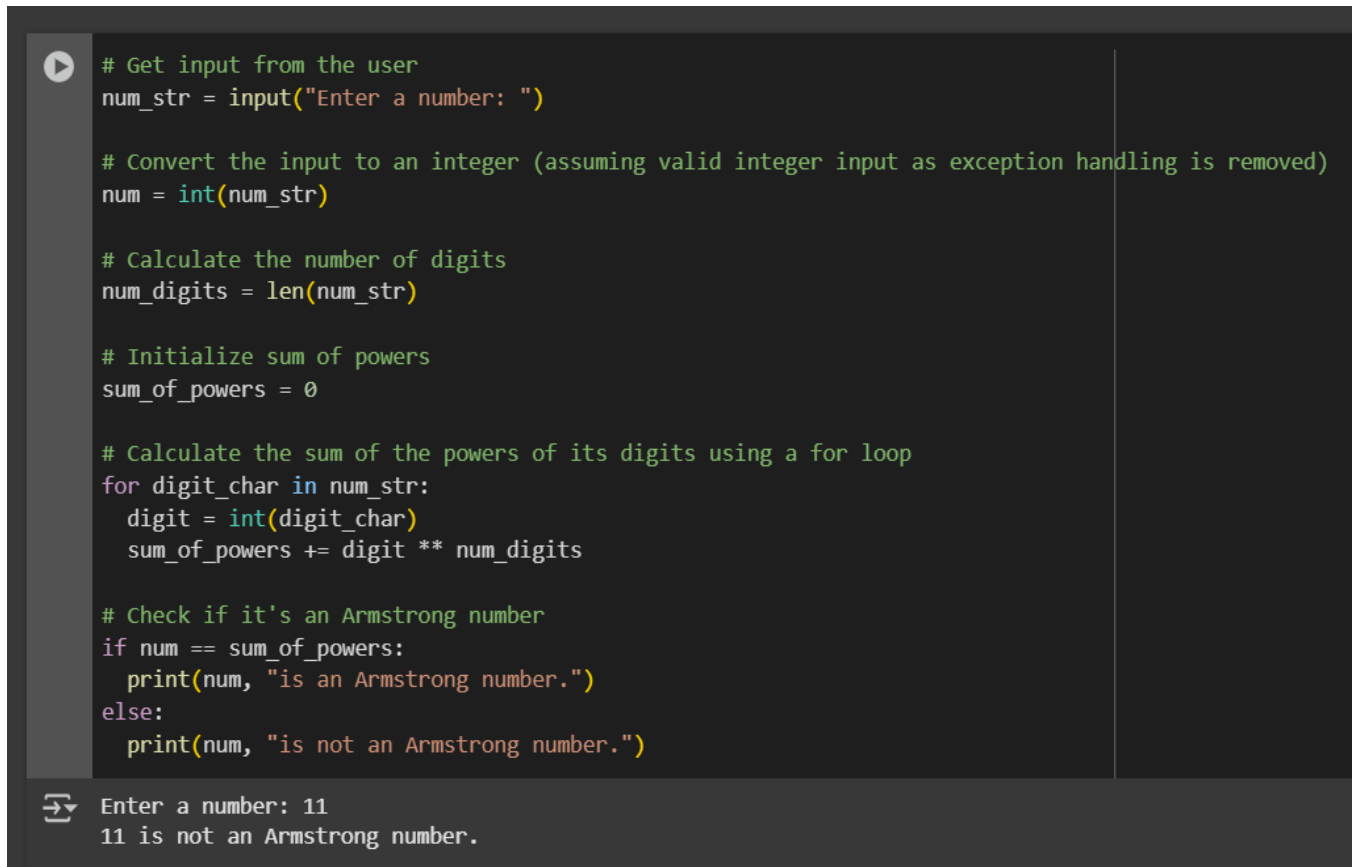
USED PROMPT :

Generate a python code to take one number as an input from the user and check if the given number is an Armstrong number or not.

CODE EXPLANATION :

- The code asks the user to enter a number.
- It calculates how many digits are in the number.
- Then, for each digit, it raises it to the power of the number of digits and sums up these values.
- If this sum equals the original number, it prints that the number is an Armstrong number; otherwise, it says it is not.

Armstrong numbers are numbers that are equal to the sum of their own digits raised to the power of the number of digits. (For example: $153 = 1^3 + 5^3 + 3^3$)



```
# Get input from the user
num_str = input("Enter a number: ")

# Convert the input to an integer (assuming valid integer input as exception handling is removed)
num = int(num_str)

# Calculate the number of digits
num_digits = len(num_str)

# Initialize sum of powers
sum_of_powers = 0

# Calculate the sum of the powers of its digits using a for loop
for digit_char in num_str:
    digit = int(digit_char)
    sum_of_powers += digit ** num_digits

# Check if it's an Armstrong number
if num == sum_of_powers:
    print(num, "is an Armstrong number.")
else:
    print(num, "is not an Armstrong number.")
```

Enter a number: 11
11 is not an Armstrong number.

TASK DESCRIPTION 5 :

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

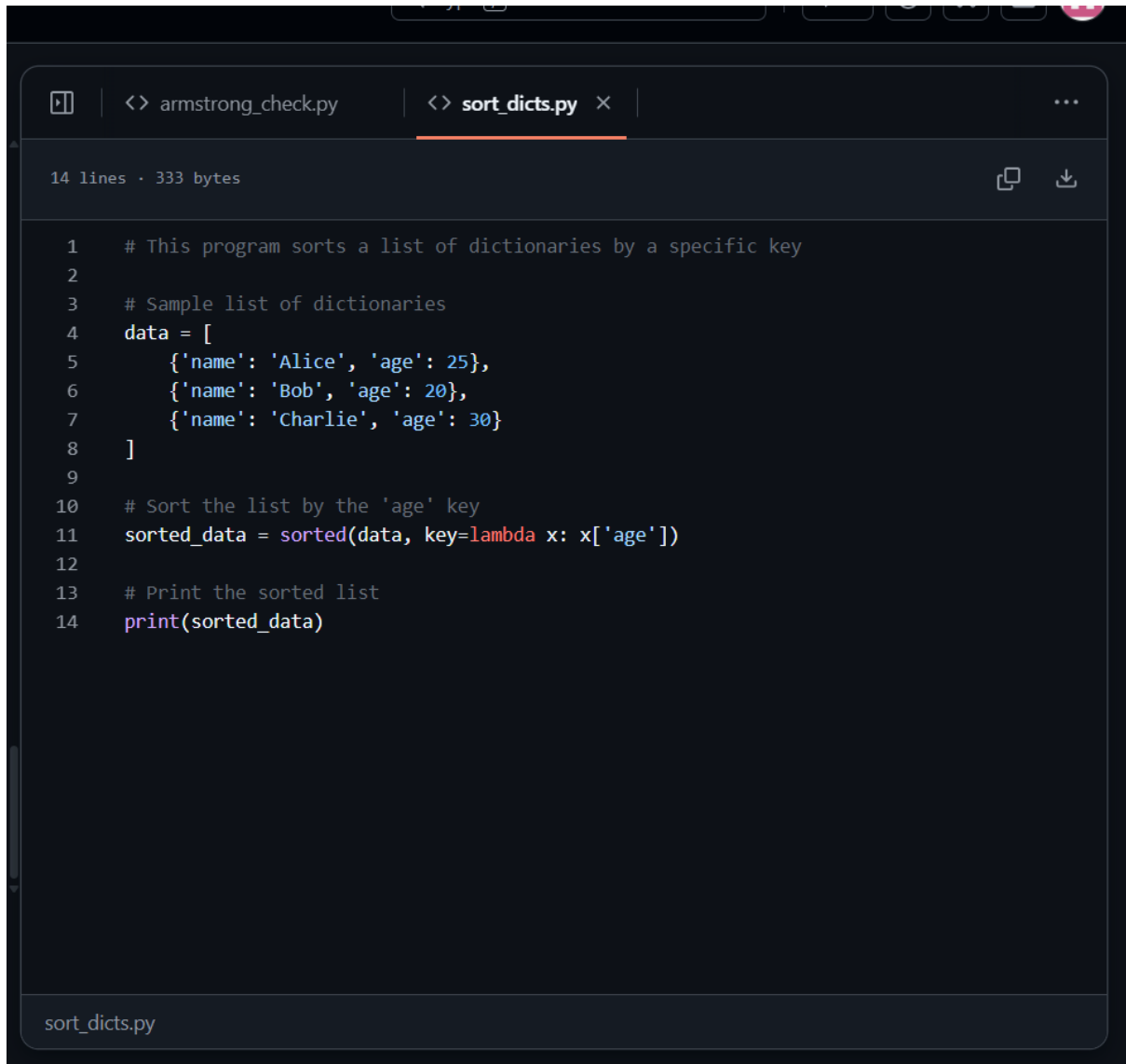
USED PROMPT :

generate a python code to sort a set of dictionaries by a specific key.

CODE EXPLANATION :

- We have a list called data that contains several dictionaries, each with a 'name' and 'age'.

- The `sorted()` function is used to sort the list of dictionaries by the value of the 'age' key.
- The `key=lambda x: x['age']` part tells Python to use the 'age' value in each dictionary for sorting.
- Finally, the sorted list is printed



```
1  # This program sorts a list of dictionaries by a specific key
2
3  # Sample list of dictionaries
4  data = [
5      {'name': 'Alice', 'age': 25},
6      {'name': 'Bob', 'age': 20},
7      {'name': 'Charlie', 'age': 30}
8  ]
9
10 # Sort the list by the 'age' key
11 sorted_data = sorted(data, key=lambda x: x['age'])
12
13 # Print the sorted list
14 print(sorted_data)
```

sort_dicts.py

+ Code+ Text

↑↓⚡🔗💬⚙️📄🗑️⋮

▶

```
# Example list of dictionaries
list_of_dictionaries = [
    {'name': 'Alice', 'age': 30},
    {'name': 'Bob', 'age': 25},
    {'name': 'Charlie', 'age': 35},
    {'name': 'David', 'age': 25}
]

# Specify the key to sort by
sort_key = 'age'

# Sort the list of dictionaries by the specified 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"Sorted list by '{sort_key}': {sorted_list}")

# You can also sort in reverse order
sorted_list_reverse = sorted(list_of_dictionaries, key=lambda x: x[sort_key], reverse=True)
print(f"Sorted list by '{sort_key}' in reverse: {sorted_list_reverse}")
```

↺

Original list: [{'name': 'Alice', 'age': 30}, {'name': 'Bob', 'age': 25}, {'name': 'Charlie', 'age': 35}, {'name': 'David', 'age': 25}]
Sorted list by 'age': [{'name': 'Bob', 'age': 25}, {'name': 'David', 'age': 25}, {'name': 'Alice', 'age': 30}, {'name': 'Charlie', 'age': 35}]
Sorted list by 'age' in reverse: [{'name': 'Charlie', 'age': 35}, {'name': 'Alice', 'age': 30}, {'name': 'Bob', 'age': 25}, {'name': 'David', 'age': 25}]