

ASSIGNMENT-2.2

A.Dhanalaxmi

2403A51269

Task-1:

Prompt given to gemini colab:

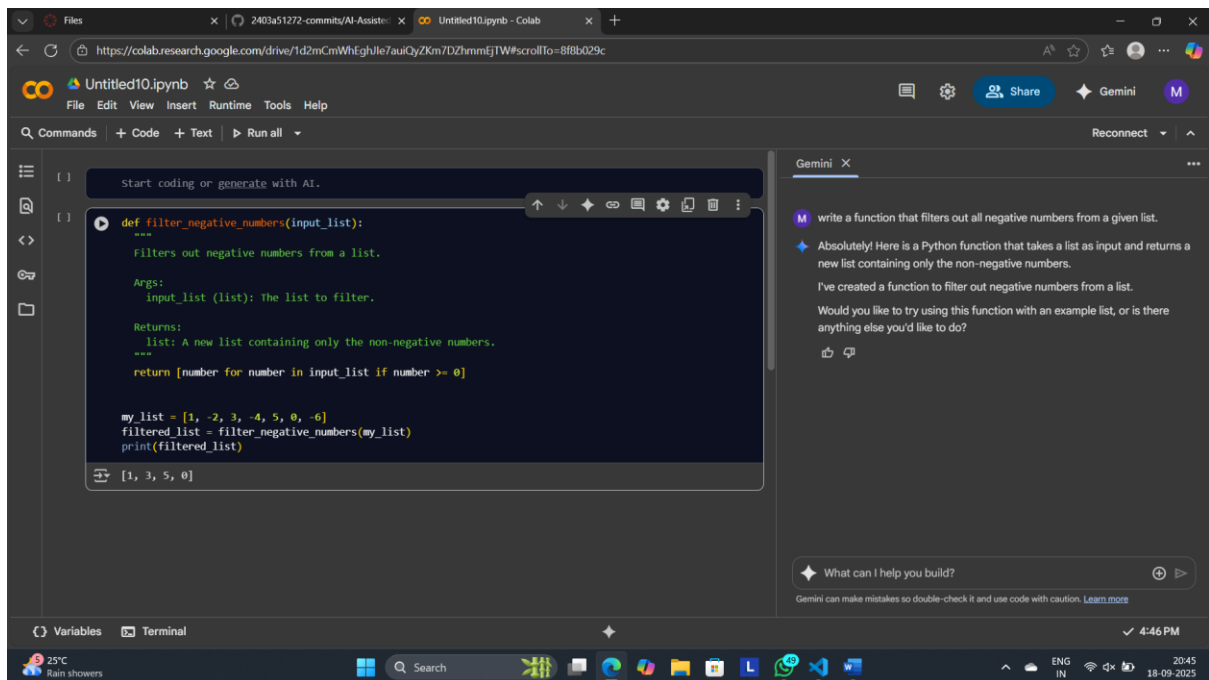
Write a function that filters out all negative numbers from a given list.

Code:

```
def filter_negative_numbers(input_list):  
  
# Example usage:  
  
# my_list = [1, -2, 3, -4, 5, 0, -6]  
  
# filtered_list = filter_negative_numbers(my_list)  
  
# print(filtered_list)
```

Output:

[1, 3, 5, 0]



Task-2:

Prompt:

Generate code that reads a text file and counts the frequency of each word.

Code:

```
import re
```

```
from collections import Counter
```

```
def count_word_frequency(file_path):
```

```
    """
```

```
    Reads a text file and counts the frequency of each word.
```

```
    Args:
```

```
        file_path (str): The path to the text file.
```

```
    Returns:
```

```
        dict: A dictionary where keys are words and values are their frequencies.
```

```
        Returns an empty dictionary if an error occurs or the file is empty.
```

```
    """
```

```
    word_counts = Counter()
```

```
    try:
```

```
        with open(file_path, 'r', encoding='utf-8') as f:
```

```
            for line in f:
```

```
                # Remove punctuation and convert to lowercase
```

```
                words = re.findall(r'\b\w+\b', line.lower())
```

```
                word_counts.update(words)
```

```
    except FileNotFoundError:
```

```
        print(f"Error: File not found at {file_path}")
```

```
except Exception as e:  
    print(f"An error occurred: {e}")  
return dict(word_counts)
```

```
word_frequencies = count_word_frequency('/smtng.txt')  
  
if word_frequencies:  
    for word, count in word_frequencies.items():  
        print(f"{word}: {count}")
```

Output:

hiii: 1

this: 1

is: 1

anjali: 1

trying: 1

to: 1

do: 1

an: 1

ai: 1

assignment: 1

Untitled10.ipynb

```

import re
from collections import Counter

def count_word_frequency(file_path):
    """
    Reads a text file and counts the frequency of each word.

    Args:
        file_path (str): The path to the text file.

    Returns:
        dict: A dictionary where keys are words and values are their frequencies.
        Returns an empty dictionary if an error occurs or the file is empty.
    """
    word_counts = Counter()
    try:
        with open(file_path, 'r', encoding='utf-8') as f:
            for line in f:
                # Remove punctuation and convert to lowercase
                words = re.findall(r'\b\w+\b', line.lower())
                word_counts.update(words)
    except FileNotFoundError:
        print(f"Error: File not found at {file_path}")
    except Exception as e:
        print(f"An error occurred: {e}")
    return dict(word_counts)

word_frequencies = count_word_frequency('/satng.txt')
if word_frequencies:
    for word, count in word_frequencies.items():
        print(f"{word}: {count}")

```

Gemini

write a function that filters out all negative numbers from a given list.

Absolutely! Here is a Python function that takes a list as input and returns a new list containing only the non-negative numbers.

I've created a function to filter out negative numbers from a list.

Would you like to try using this function with an example list, or is there anything else you'd like to do?

Generate code that reads a text file and counts the frequency of each word.

Okay, I can generate code to do that. Here's a function that reads a text file and returns a dictionary with word frequencies.

I've created a function to count the frequency of words in a text file.

Would you like to test this function with a sample text file, or is there anything else you'd like to do?

explain error

It looks like the error is a `SyntaxError: (unicode error)`

What can I help you build?

Gemini can make mistakes so double-check it and use code with caution. [Learn more](#)

Untitled10.ipynb

```

except Exception as e:
    print(f"An error occurred: {e}")
return dict(word_counts)

word_frequencies = count_word_frequency('/satng.txt')
if word_frequencies:
    for word, count in word_frequencies.items():
        print(f"{word}: {count}")

```

```

hihi: 1
this: 1
is: 1
anjali: 1
trying: 1
to: 1
do: 1
an: 1
ai: 1
assignment: 1

```

hi this is anjali

Gemini

write a function that filters out all negative numbers from a given list.

Absolutely! Here is a Python function that takes a list as input and returns a new list containing only the non-negative numbers.

I've created a function to filter out negative numbers from a list.

Would you like to try using this function with an example list, or is there anything else you'd like to do?

Generate code that reads a text file and counts the frequency of each word.

Okay, I can generate code to do that. Here's a function that reads a text file and returns a dictionary with word frequencies.

I've created a function to count the frequency of words in a text file.

Would you like to test this function with a sample text file, or is there anything else you'd like to do?

explain error

It looks like the error is a `SyntaxError: (unicode error)`

What can I help you build?

Gemini can make mistakes so double-check it and use code with caution. [Learn more](#)

Explanation by gemini:

- `import re`: This line imports the `re` module, which provides regular expression operations. It's used here to find words in each line of the text file.
- `from collections import Counter`: This line imports the `Counter` class from the `collections` module. `Counter` is a specialized dictionary subclass for counting hashable objects. It's very useful for counting the frequency of items in a list.

- `def count_word_frequency(file_path)::` This line defines a function named `count_word_frequency` that takes one argument, `file_path`, which is expected to be a string representing the path to the text file.
- `""" ... """`: This is a docstring, which explains what the function does, its arguments, and what it returns.
- `word_counts = Counter():` This line initializes an empty `Counter` object named `word_counts`. This object will store the words and their frequencies as key-value pairs.
- `try::` This starts a try block, which is used for error handling. Code within the try block is executed, and if an error occurs, the code in the corresponding except block is executed.
- `with open(file_path, 'r', encoding='utf-8') as f::` This line opens the file specified by `file_path` in read mode ('r') with UTF-8 encoding. The with statement ensures that the file is automatically closed even if errors occur. The opened file object is assigned to the variable `f`.
- `for line in f::` This loop iterates through each line in the opened file.
- `words = re.findall(r'\b\w+\b', line.lower()):` This is a key line for processing.
 - `line.lower():` Converts the current line to lowercase. This ensures that words like "The" and "the" are counted as the same word.
 - `re.findall(r'\b\w+\b', ...):` This uses the `re.findall` function to find all occurrences of a pattern in the lowercase line.
 - `r'\b\w+\b'`: This is the regular expression pattern.
 - `r''`: Denotes a raw string, which is useful for regular expressions to avoid issues with backslashes.
 - `\b`: Matches a word boundary. This ensures that only whole words are matched and not parts of words.
 - `\w+`: Matches one or more word characters (letters, numbers, and underscore).
 - The result of `re.findall` is a list of all the words found in the line, which is assigned to the `words` variable.
- `word_counts.update(words):` This line updates the `word_counts` `Counter` with the words found in the current line. If a word is already in the `Counter`, its count is incremented; otherwise, the word is added with a count of 1.

- `except FileNotFoundError::` This block is executed if a `FileNotFoundError` occurs within the try block (i.e., if the specified file does not exist).
- `print(f"Error: File not found at {file_path}"):` This line prints an error message indicating that the file was not found, including the file path.
- `except Exception as e::` This block catches any other type of exception that might occur during the execution of the try block.
- `print(f"An error occurred: {e}"):` This line prints a generic error message along with the specific error `e` that occurred.
- `return dict(word_counts):` After processing all lines (or if an error occurred), this line converts the `word_counts` Counter object into a regular dictionary and returns it.
- `word_frequencies = count_word_frequency(r'C:\Users\anjali\OneDrive\Documents\smtng.txt'):` This line calls the `count_word_frequency` function with the specified file path and stores the returned dictionary of word frequencies in the `word_frequencies` variable. The `r` before the string indicates a raw string to handle the backslashes in the path.
- `if word_frequencies::` This line checks if the `word_frequencies` dictionary is not empty. The if condition is true if the dictionary contains any items (meaning words were counted successfully).
- `for word, count in word_frequencies.items()::` If `word_frequencies` is not empty, this loop iterates through each key-value pair (word and its count) in the dictionary.
- `print(f"{word}: {count}"):` This line prints each word followed by a colon and its corresponding count.

Task-3:

Prompt:

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

Code:

`class Book:`

`def __init__(self, title: str, author: str) -> None:`

```
self.title = title
```

```
self.author = author
```

```
def summary(self) -> None:
```

```
    print(f"Title: {self.title}, Author: {self.author}")
```

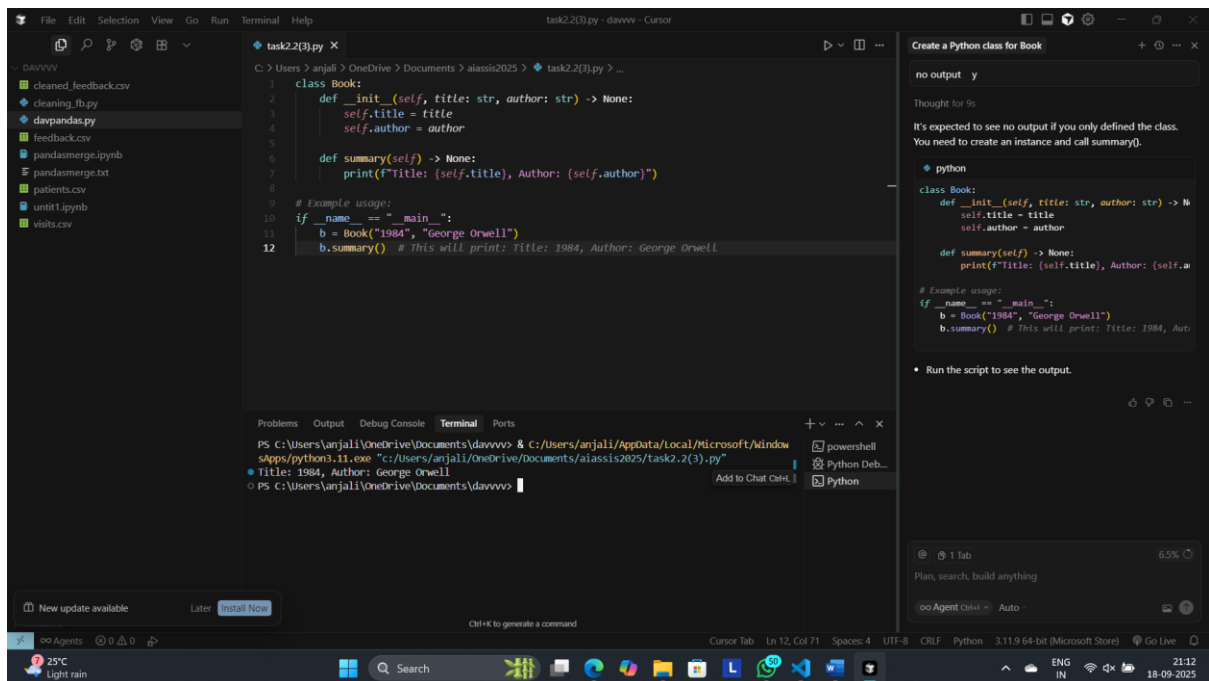
```
if __name__ == "__main__":
```

```
    b = Book("1984", "George Orwell")
```

```
    b.summary()
```

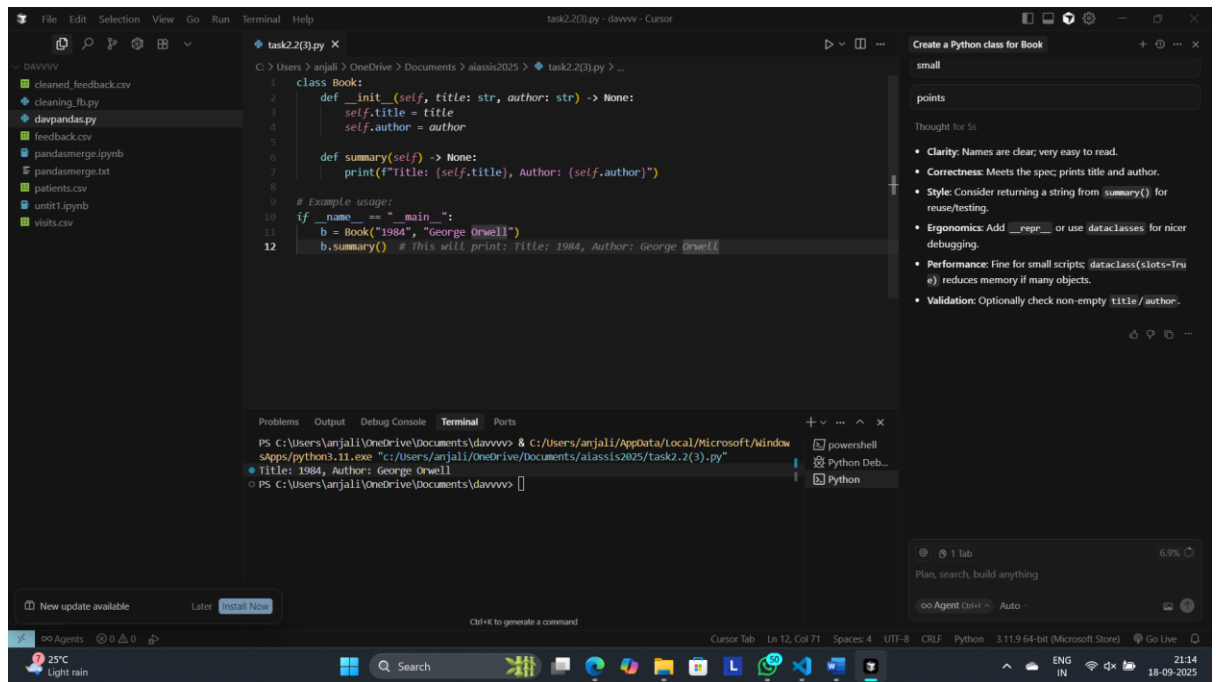
Output:

Title: 1984, Author: George Orwell



Opinion:

- **Clarity:** Names are clear; very easy to read.
- **Correctness:** Meets the spec; prints title and author.
- **Style:** Consider returning a string from summary() for reuse/testing.
- **Ergonomics:** Add `__repr__` or use dataclasses for nicer debugging.
- **Performance:** Fine for small scripts; `dataclass(slots=True)` reduces memory if many objects.
- **Validation:** Optionally check non-empty title/author.



Task-4:

Prompt:

Write a program that checks whether a number is an Armstrong number.

Code by gemini:

```
def is_armstrong_number(number):
```

```
# Convert the number to a string to get the number of digits
```

```
num_str = str(number)
```

```
num_digits = len(num_str)
```

```
# Calculate the sum of each digit raised to the power of the number of digits
```

```
armstrong_sum = 0
```

```
for digit in num_str:
```

```
    armstrong_sum += int(digit) ** num_digits
```



```
# Check if the sum is equal to the original number  
return armstrong_sum == number
```

```
num_to_check = 153
```

```
if is_armstrong_number(num_to_check):  
    print(f"{num_to_check} is an Armstrong number.")  
else:  
    print(f"{num_to_check} is not an Armstrong number.")
```

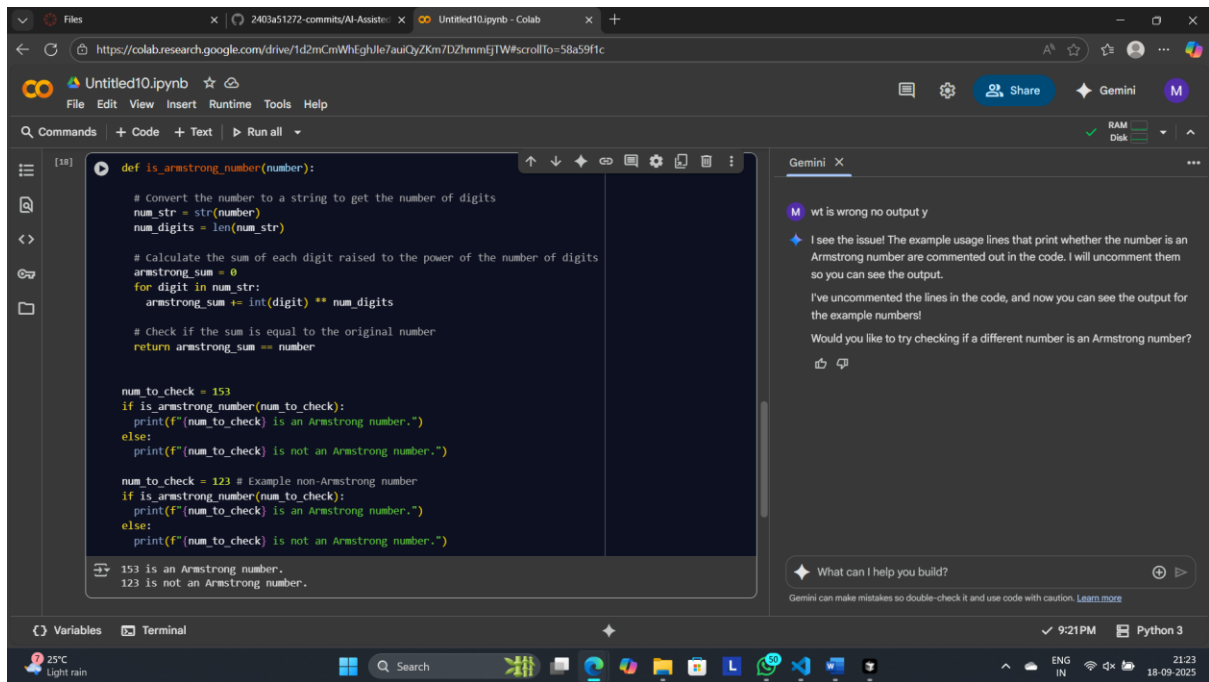
```
num_to_check = 123 # Example non-Armstrong number
```

```
if is_armstrong_number(num_to_check):  
    print(f"{num_to_check} is an Armstrong number.")  
else:  
    print(f"{num_to_check} is not an Armstrong number.")
```

Output:

153 is an Armstrong number.

123 is not an Armstrong number.



Modified Code by cursor:

```
import math
```

```
def is_armstrong_number(number: int) -> bool:
```

```
    if number < 0:
```

```
        return False
```

```
    # Digit count via log10 (faster than str for large ints)
```

```
    if number == 0:
```

```
        num_digits = 1
```

```
    else:
```

```
        num_digits = int(math.log10(number)) + 1
```

```
    # Precompute powers for digits 0-9
```

```
    digit_power = [d ** num_digits for d in range(10)]
```

```
    # Sum digit powers using integer arithmetic
```

```
total = 0

n = number

if n == 0:

    total = digit_power[0]

else:

    while n:

        total += digit_power[n % 10]

        n //= 10

return total == number
```

```
if __name__ == "__main__":

    tests = [153, 370, 371, 407, 0, 1, 2, 9, 10, 123, 9474, 9475, -153]

    for num in tests:

        print(f"{num}: {'Armstrong' if is_armstrong_number(num) else 'Not Armstrong'}")
```

Output:

153: Armstrong

370: Armstrong

371: Armstrong

407: Armstrong

0: Armstrong

1: Armstrong

2: Armstrong

9: Armstrong

10: Not Armstrong

123: Not Armstrong

9474: Armstrong

9475: Not Armstrong

-153: Not Armstrong

The screenshot shows the Cursor IDE interface. The main editor displays a Python file named `task2.2(3).py` with the following code:

```
def is_armstrong_number(number: int) -> bool:
    total = 0
    n = number
    if n == 0:
        total = digit_power[0]
    else:
        while n:
            total += digit_power[n % 10]
            n //= 10
    return total == number

if __name__ == "__main__":
    tests = [153, 370, 371, 407, 0, 1, 2, 9, 10, 123, 9474, 9475, -153]
    for num in tests:
        print(f"{num}: {'Armstrong' if is_armstrong_number(num) else 'Not Armstrong'}")
```

The terminal output shows the results of running the script:

```
153: Armstrong
370: Armstrong
371: Armstrong
407: Armstrong
0: Armstrong
1: Armstrong
2: Armstrong
9: Armstrong
10: Not Armstrong
123: Not Armstrong
9474: Armstrong
9475: Not Armstrong
-153: Not Armstrong
```

On the right side, a sidebar titled "Create a Python class for Book" shows an "Optimized and structured version" of the code, which includes type hints, docstrings, and precomputed powers for digits 0-9.

Summary of modifications by cursor:

- **Negative/zero handling:** Returns False for negatives; treats 0 as a 1-digit Armstrong number for correctness.
- **Digit count optimization:** Uses \log_{10} to compute digit count in $O(1)$ vs converting to string; special-cases 0.
- **Integer-only digit loop:** Replaces per-digit str/int conversions with $\% 10$ and $// 10$, reducing allocations.
- **Power precomputation:** Caches 0–9 to the `num_digits` power once, avoiding repeated exponentiation in the loop.
- **Type hints:** Adds annotations for function signature, improving readability and tooling support.
- **Structure:** Moves ad-hoc prints into an `if __name__ == "__main__":` test block with a small test set.
- **Performance impact:** Fewer temporary objects and exponentiations; tighter loop; more noticeable for large numbers.
- **Behavior:** Output/decision remains the same for valid inputs; added guard for negatives.

Task-5:

Prompt:

Generate code for sorting a list of dictionaries by a specific key (e.g., age)

Code by gemini:

```
def sort_list_of_dictionaries(list_of_dicts, key_to_sort_by):
```

```
    """
```

```
    Sorts a list of dictionaries by a specific key.
```

Args:

list_of_dicts (list): The list of dictionaries to sort.

key_to_sort_by (str): The key to sort the dictionaries by.

Returns:

list: A new list of dictionaries sorted by the specified key.

```
    """
```

```
    return sorted(list_of_dicts, key=lambda x: x[key_to_sort_by])
```

```
data = [
```

```
{'name': 'Alice', 'age': 30},
```

```
{'name': 'Bob', 'age': 25},
```

```
{'name': 'Charlie', 'age': 35}
```

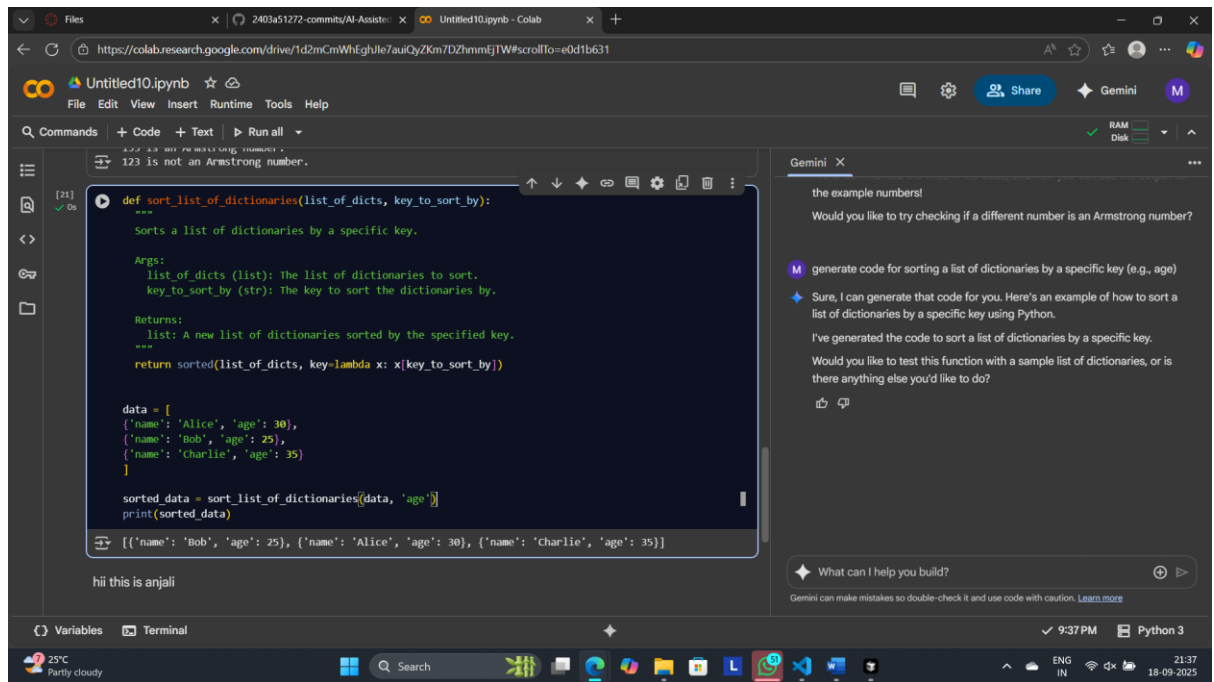
```
]
```

```
sorted_data = sort_list_of_dictionaries(data, 'age')
```

```
print(sorted_data)
```

Output:

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



Code by Cursor:

```
from operator import itemgetter
```

```
people = [  
    {"name": "Alice", "age": 29},  
    {"name": "Bob", "age": 24},  
    {"name": "Cara", "age": 29},  
    {"name": "Dan", "age": 31},  
]
```

```
# Ascending by age
```

```
sorted_people = sorted(people, key=itemgetter("age"))  
print(sorted_people)
```

```
# Descending by age
```

```
sorted_people_desc = sorted(people, key=itemgetter("age"), reverse=True)  
print(sorted_people_desc)
```

```
# Handle missing 'age' safely (treat missing as +infinity)
```

```
sorted_safe = sorted(people, key=lambda d: d.get("age", float("inf")))
```

```
print(sorted_safe)
```

```
# Tie-breaker: age, then name
```

```
sorted_with_tie = sorted(people, key=lambda d: (d["age"], d["name"]))
```

```
print(sorted_with_tie)
```

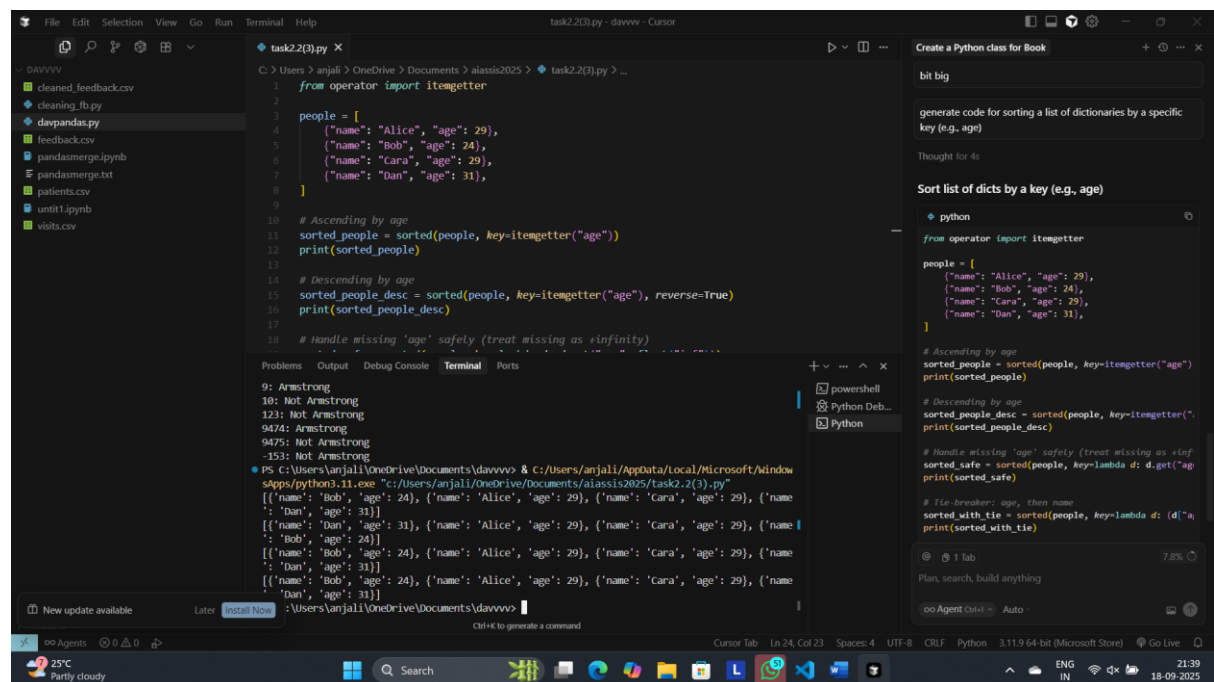
Output:

```
{'name': 'Bob', 'age': 24}, {'name': 'Alice', 'age': 29}, {'name': 'Cara', 'age': 29}, {'name': 'Dan', 'age': 31}]
```

```
{'name': 'Dan', 'age': 31}, {'name': 'Alice', 'age': 29}, {'name': 'Cara', 'age': 29}, {'name': 'Bob', 'age': 24}]
```

```
{'name': 'Bob', 'age': 24}, {'name': 'Alice', 'age': 29}, {'name': 'Cara', 'age': 29}, {'name': 'Dan', 'age': 31}]
```

```
{'name': 'Bob', 'age': 24}, {'name': 'Alice', 'age': 29}, {'name': 'Cara', 'age': 29}, {'name': 'Dan', 'age': 31}]
```



```
task2.2(3).py X
C:\Users\anjali> OneDrive\Documents> aiaiss2025> task2.2(3).py ...
1 from operator import itemgetter
2
3 people = [
4     {"name": "Alice", "age": 29},
5     {"name": "Bob", "age": 24},
6     {"name": "Cara", "age": 29},
7     {"name": "Dan", "age": 31},
8 ]
9
10 # Ascending by age
11 sorted_people = sorted(people, key=itemgetter("age"))
12 print(sorted_people)
13
14 # Descending by age
15 sorted_people_desc = sorted(people, key=itemgetter("age"), reverse=True)
16 print(sorted_people_desc)
17
18 # Handle missing 'age' safely (treat missing as infinity)
19 sorted_safe = sorted(people, key=lambda d: d.get("age", float("inf")))
20 print(sorted_safe)
21
22 # Tie-breaker: age, then name
23 sorted_with_tie = sorted(people, key=lambda d: (d["age"], d["name"]))
24 print(sorted_with_tie)
```

9: Armstrong
10: Not Armstrong
123: Not Armstrong
9474: Armstrong
9475: Not Armstrong
-153: Not Armstrong

```
PS C:\Users\anjali\OneDrive\Documents\davvvv> C:\Users\anjali\AppData\Local\Microsoft\Windows\
scripts\python3.11.exe "C:\Users\anjali\OneDrive\Documents\aiasss2025\task2.2(3).py"
[{'name': 'Bob', 'age': 24}, {'name': 'Alice', 'age': 29}, {'name': 'Cara', 'age': 29}, {'name': 'Dan', 'age': 31}]
[{'name': 'Dan', 'age': 31}, {'name': 'Alice', 'age': 29}, {'name': 'Cara', 'age': 29}, {'name': 'Bob', 'age': 24}]
[{'name': 'Bob', 'age': 24}, {'name': 'Alice', 'age': 29}, {'name': 'Cara', 'age': 29}, {'name': 'Dan', 'age': 31}]
[{'name': 'Bob', 'age': 24}, {'name': 'Alice', 'age': 29}, {'name': 'Cara', 'age': 29}, {'name': 'Dan', 'age': 31}]
```

Summary of comparison:

- **Clarity:** Gemini's function + docstring are clean and readable. Cursor showed multiple concrete examples (asc/desc, tie-breakers), which aids understanding.

- **Performance:** Both are $O(n \log n)$. Cursor's itemgetter is a bit faster than a lambda key; difference is small.
- **Robustness:** Gemini's version raises `KeyError` if a key is missing. Cursor included a safe pattern with `dict.get(..., default)` and tie-breakers.
- **Ergonomics:** Gemini = concise baseline. Cursor = ready-to-use variants for real data quirks.
- **Bottom line:** Use Gemini's structure + docstring; swap in itemgetter and optional default/tie-breaker from Cursor.