# Task-1:

ASSIGNMENT-2.2

MD ZIAUDDIN 2403A51271

Prompt given to gemini colab:

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

Code:

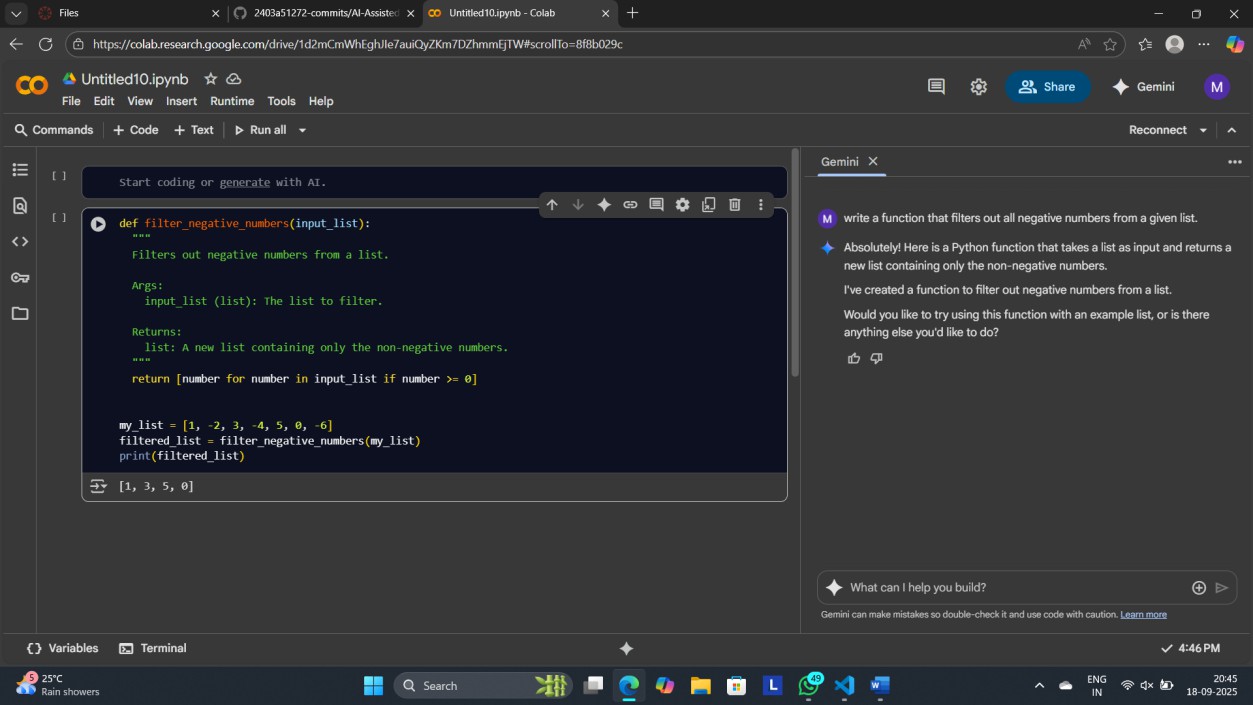
def filter\_negative\_numbers(input\_list): # Example usage:

# 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

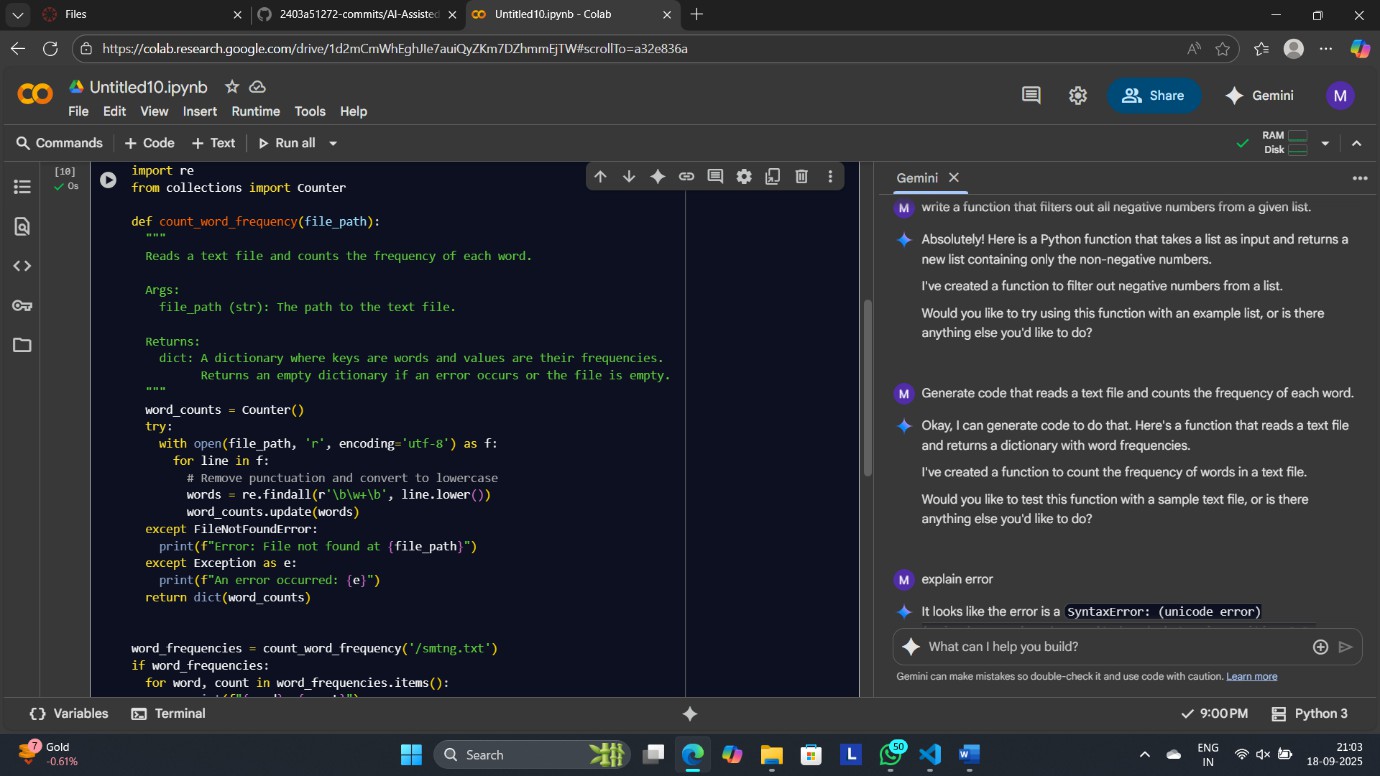
to: 1

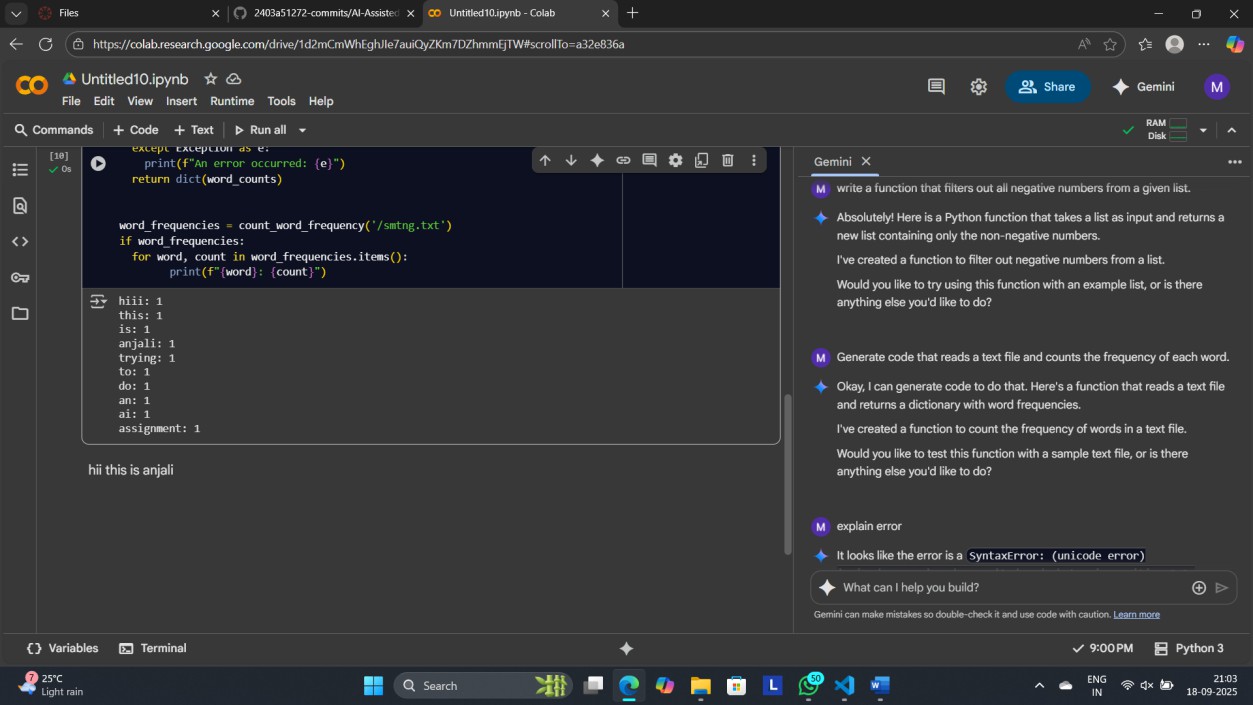
do: 1

an: 1

ai: 1

assignment: 1





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

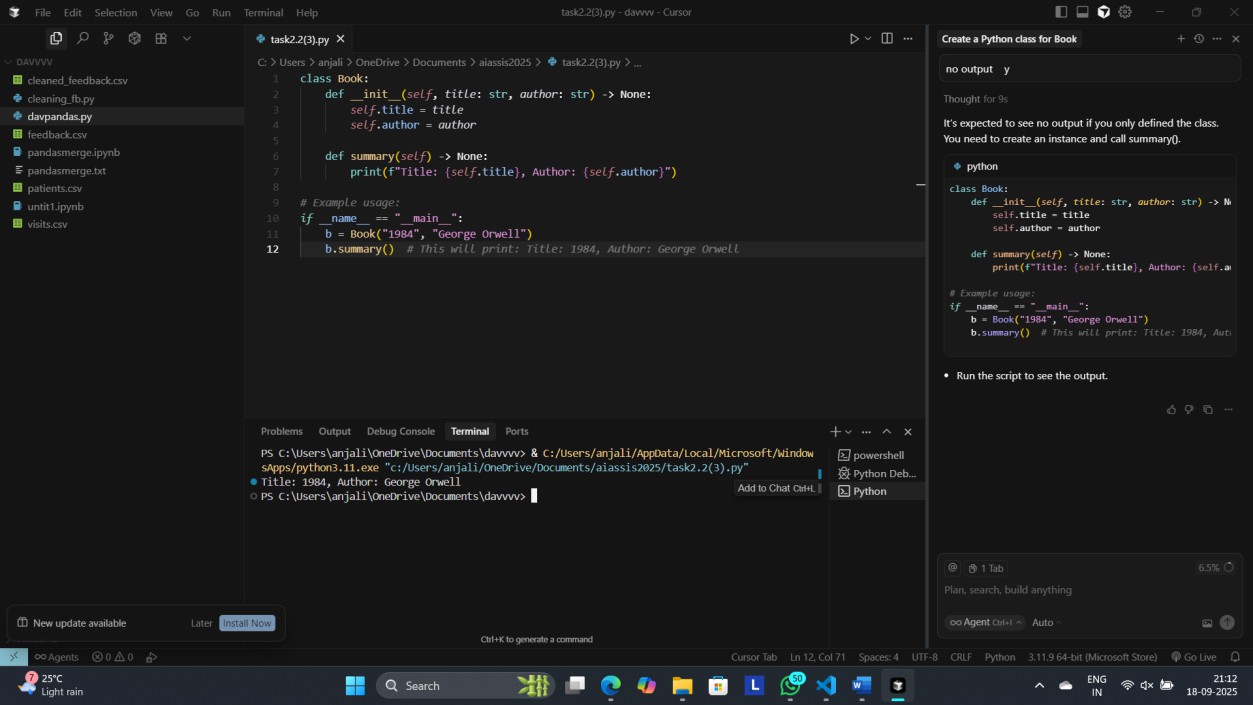
def summary(self) -> None:

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.")

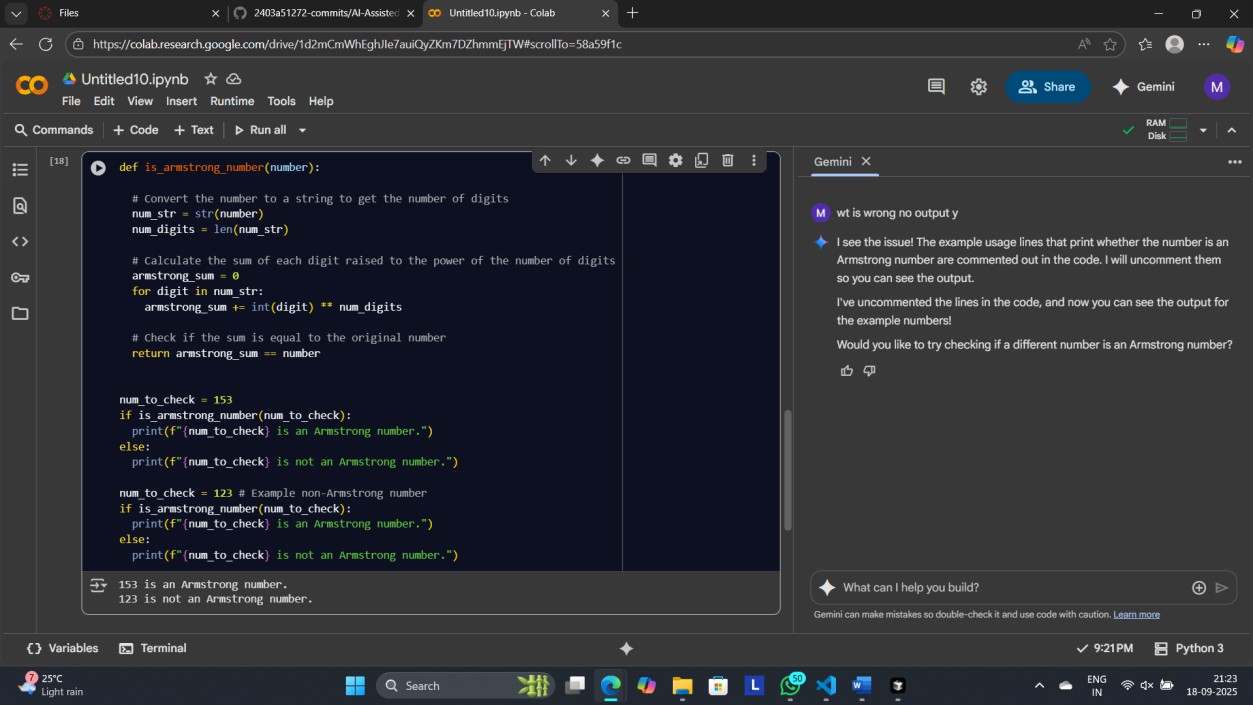
num\_to\_check = 123 # Example non-Armstrong number if is\_armstrong\_number(num\_to\_check):

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

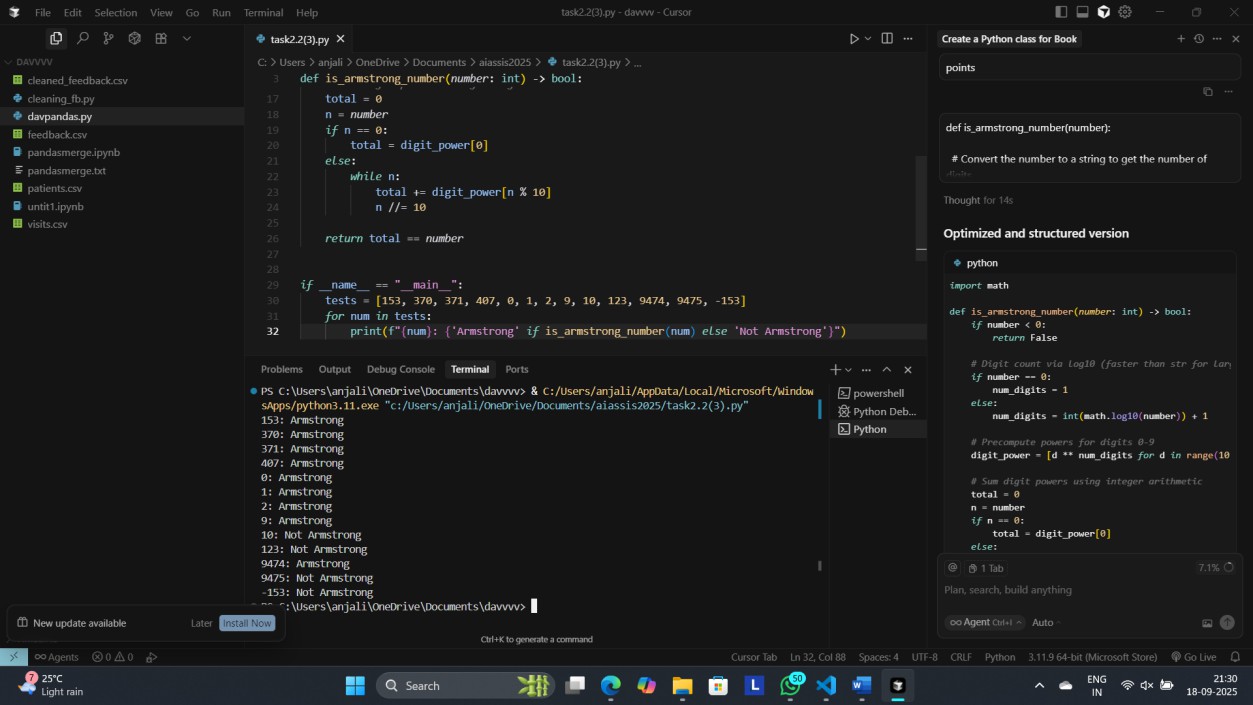
10: Not Armstrong

123: Not Armstrong

9474: Armstrong

9475: Not Armstrong

-153: Not Armstrong



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 log10 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},

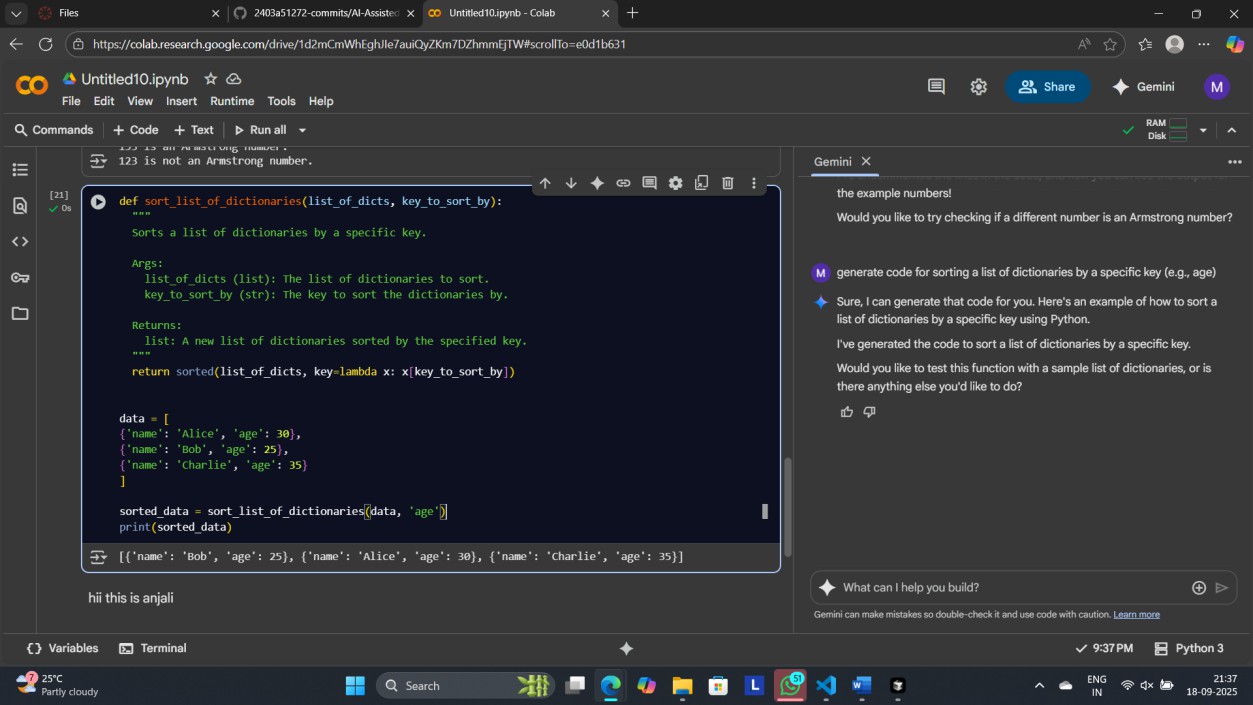
{'name': 'Charlie', 'age': 35}

]

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)

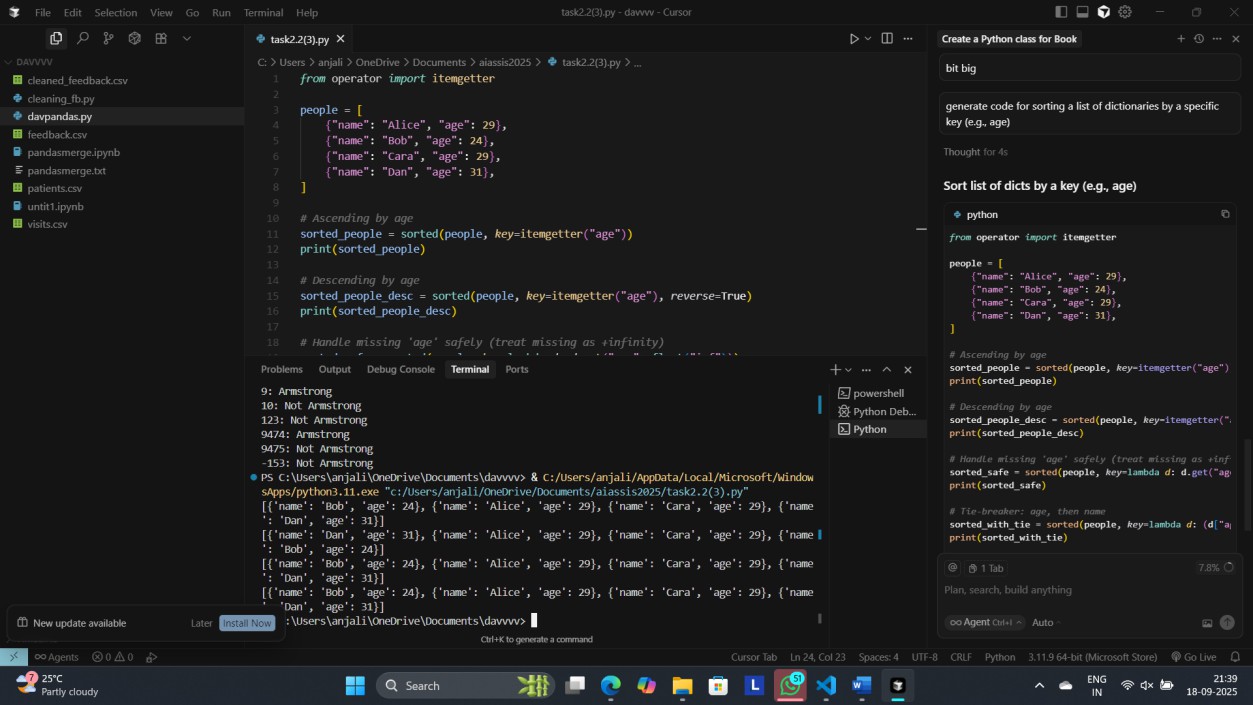
Output:

[{'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}]



Summary of comparision:

* **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.