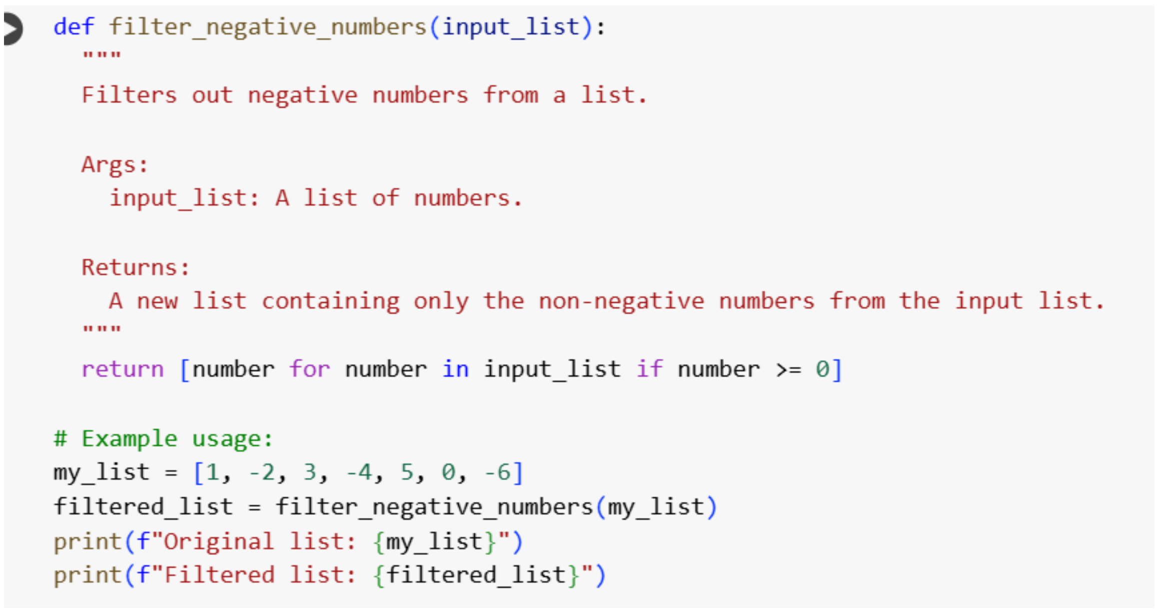
Assignment-2

1. prompt: write a program in python function to filters out all negative numbers from a given list.



```python
def filter_negative_numbers(input_list):
  """
  Filters out negative numbers from a list.

  Args:
    input_list: A list of numbers.

  Returns:
    A new list containing only the non-negative numbers from the input list.
  """
  return [number for number in input_list if number >= 0]

# Example usage:
my_list = [1, -2, 3, -4, 5, 0, -6]
filtered_list = filter_negative_numbers(my_list)
print(f"Original list: {my_list}")
print(f"Filtered list: {filtered_list}")
```

output:

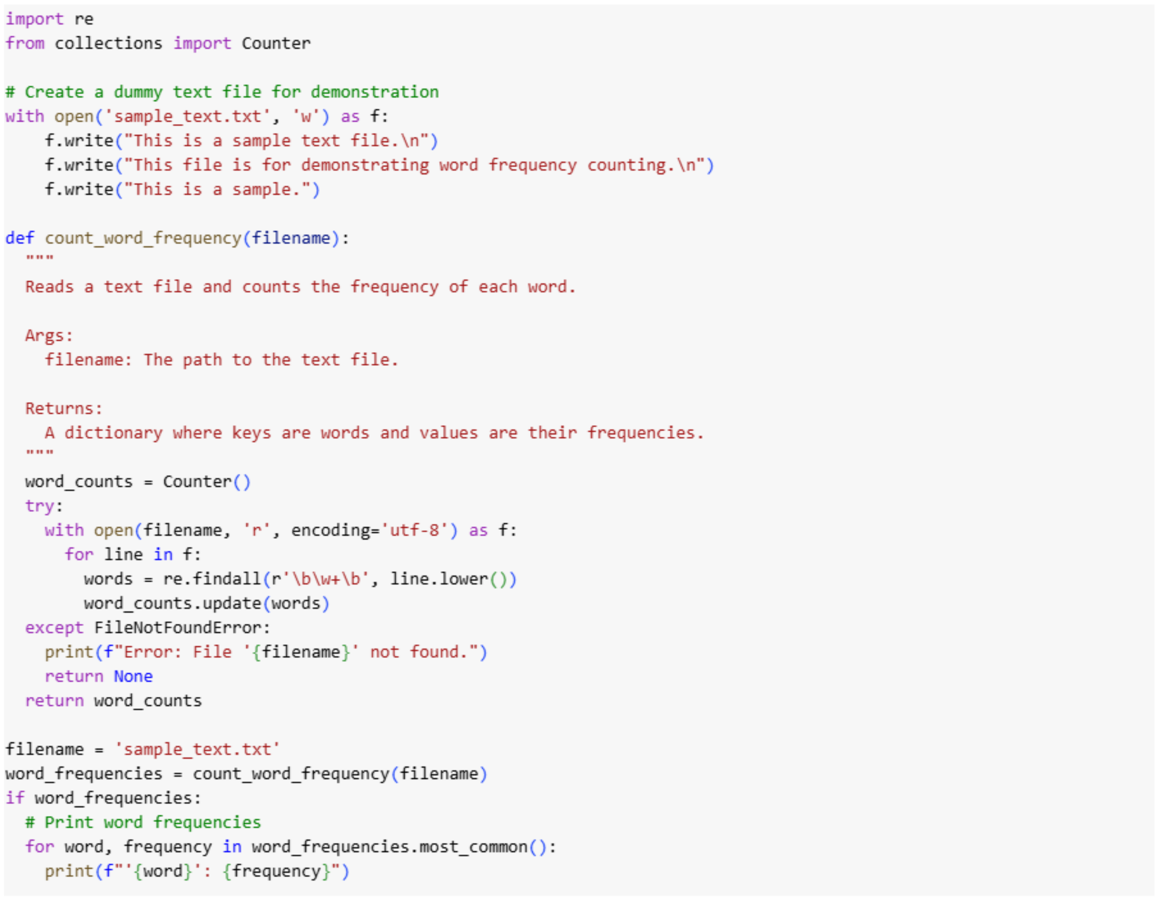
```
Original list: [1, -2, 3, -4, 5, 0, -6]
Filtered list: [1, 3, 5, 0]
```

Eplaination:

- This code defines a function called filter_negative_numbers that takes one argument, input_list.
- Inside the function, a list comprehension [number for number in input_list if number >= 0] is used to create a new list.
- for number in input_list: This iterates through each element in the input_list.
- if number >= 0: This is a condition that checks if the current number is greater than or equal to 0 (i.e., non-negative).
- number: If the condition is true, the current number is included in the new list.
- Finally, the function returns the new list containing only the non-negative numbers.

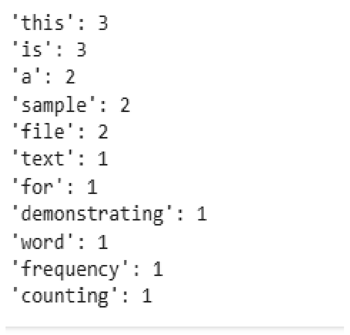
2.**prompt**:-**write a program in python to reads a text file and counts the frequency of each word**

code:



```python
import re
from collections import Counter

# Create a dummy text file for demonstration
with open('sample_text.txt', 'w') as f:
    f.write("This is a sample text file.\n")
    f.write("This file is for demonstrating word frequency counting.\n")
    f.write("This is a sample.")

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

  Args:
    filename: The path to the text file.

  Returns:
    A dictionary where keys are words and values are their frequencies.
  """
  word_counts = Counter()
  try:
    with open(filename, 'r', encoding='utf-8') as f:
      for line in f:
        words = re.findall(r'\b\w+\b', line.lower())
        word_counts.update(words)
  except FileNotFoundError:
    print(f"Error: File '{filename}' not found.")
    return None
  return word_counts

filename = 'sample_text.txt'
word_frequencies = count_word_frequency(filename)
if word_frequencies:
  # Print word frequencies
  for word, frequency in word_frequencies.most_common():
    print(f"'{word}': {frequency}")
```

output:



```
'this': 3
'is': 3
'a': 2
'sample': 2
'file': 2
'text': 1
'for': 1
'demonstrating': 1
'word': 1
'frequency': 1
'counting': 1
```

Explanation:

- import re and from collections import Counter: These lines import necessary modules. re is for regular expressions, used here to find words, and Counter is a specialized dictionary from the
- collections module that's perfect for counting hashable objects (like words).
- Creating a dummy file:This block creates a simple text file named sample_text.txt and writes some lines into it. This is just to have a file to work with for the demonstration.
- The with open(...) syntax ensures the file is properly closed even if errors occur. The 'w' mode means the file is opened for writing, and if it already exists, its content will be overwritten
- def count_word_frequency(filename):: This defines a function named count_word_frequency that takes one argument, filename, which is the path to the text file you want to analyze.
- word_counts = Counter(): Inside the function, a Counter object named word_counts is initialized. This object will store the words as keys and their frequencies (counts) as values.
-
- try...except FileNotFoundError: This block handles the case where the specified file does not exist. If a FileNotFoundError occurs when trying to open the file, it prints an error message and returns None.
- with open(filename, 'r', encoding='utf-8') as f:: This opens the file specified by filename in read mode ('r') with UTF-8 encoding. The with open(...) ensures the file is closed automatically.
- for line in f:: This loop reads the file line by line.
- words = re.findall(r'¥b¥w+¥b', line.lower()):

line.lower(): Converts the current line to lowercase. This ensures that "The" and "the" are counted as the same word.

- re.findall(r'¥b¥w+¥b', ...): This uses a regular expression to find all words in the lowercase line.

  ¥b: Matches a word boundary (the position between a word character and a non-word character).

  ¥w+: Matches one or more word characters (letters, digits, or underscore).

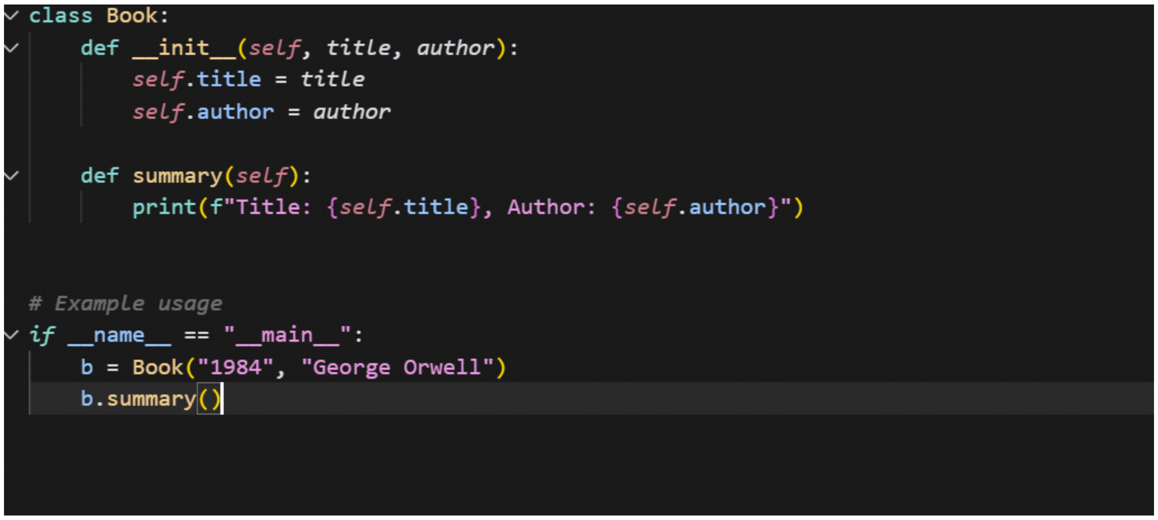
- This effectively extracts sequences of word characters that are separated by non-word characters (like spaces, punctuation, newlines).

- word_counts.update(words): This 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.

- return word_counts: After processing all lines, the function returns the word_counts Counter object

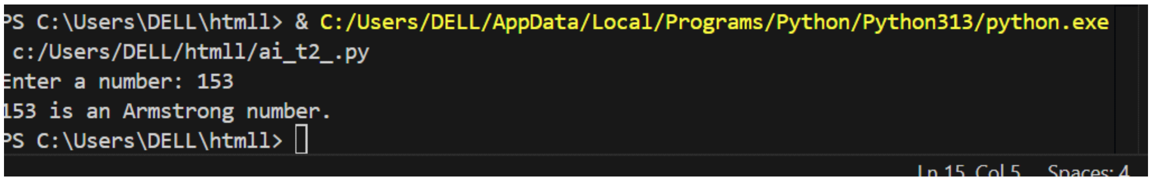
3.prompt:write a program in python   create class called Book with attributes title, author, and a method summary() that prints the details

code:-



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

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


# Example usage
if __name__ == "__main__":
    b = Book("1984", "George Orwell")
    b.summary()
```

output:



```
PS C:\Users\DELL\html1> & C:/Users/DELL/AppData/Local/Programs/Python/Python313/python.exe
 c:/Users/DELL/html1/ai_t2_.py
Enter a number: 153
153 is an Armstrong number.
PS C:\Users\DELL\html1> 
```

Explanation:

- class Book:: This line defines a new class named Book. In object-oriented programming, a class is a blueprint for creating objects (instances). In this case, the Book class is a blueprint for creating objects that represent books.

- def __init__(self, title, author):: This is the constructor method of the Book class. The __init__ method is a special method that gets automatically called when you create a new object (an instance) of the class.
- self: This refers to the instance of the class itself. It's a convention in Python to name the first parameter of instance methods self.
- title and author: These are parameters that you pass when you create a Book object. They represent the title and author of the book.
- self.title = title and self.author = author: These lines assign the values passed as arguments (title and author) to the instance variables (also named title and author) of the Book object. This means each Book object will have its own title and author attributes.
- def summary(self):: This defines another method within the Book class called summary. Methods defined within a class are functions that can be called on instances of that class.
- self: Again, this refers to the instance of the class on which the method is being called.
- print(f"Title: {self.title}, Author: {self.author}"): This line prints a formatted string that displays the title and author of the specific Book object. It accesses the title and author attributes of the instance using self.title and self.author.
- if __name__ == "__main__":: This is a common Python idiom. The code inside this if block will only run when the script is executed directly (not when it's imported as a module into another script). This is where the example usage of the Book class is placed.
- b = Book("1984", "George Orwell"): This line creates a new instance (an object) of the Book class.
- Book("1984", "George Orwell"): This calls the Book class, passing the strings "1984" and "George Orwell" as arguments for the title and author parameters of the __init__ method.

- b = ...: The newly created Book object is assigned to the variable b.
- b.summary(): This line calls the summary method on the Book object referenced by the variable b. This will execute the code inside the summary method for that specific Book object, printing its title and author.

4.prompt:-write a program that checks whether a number is an Armstrong number or not.

code:-

```
def is_armstrong_number(number):

  # Convert the number to a string to easily access its digits
  num_str = str(number)
  num_digits = len(num_str)
  sum_of_powers = 0

  # Calculate the sum of each digit raised to the power of the number of digits
  for digit in num_str:
    sum_of_powers += int(digit) ** num_digits

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

# Example usage:
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
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.
```

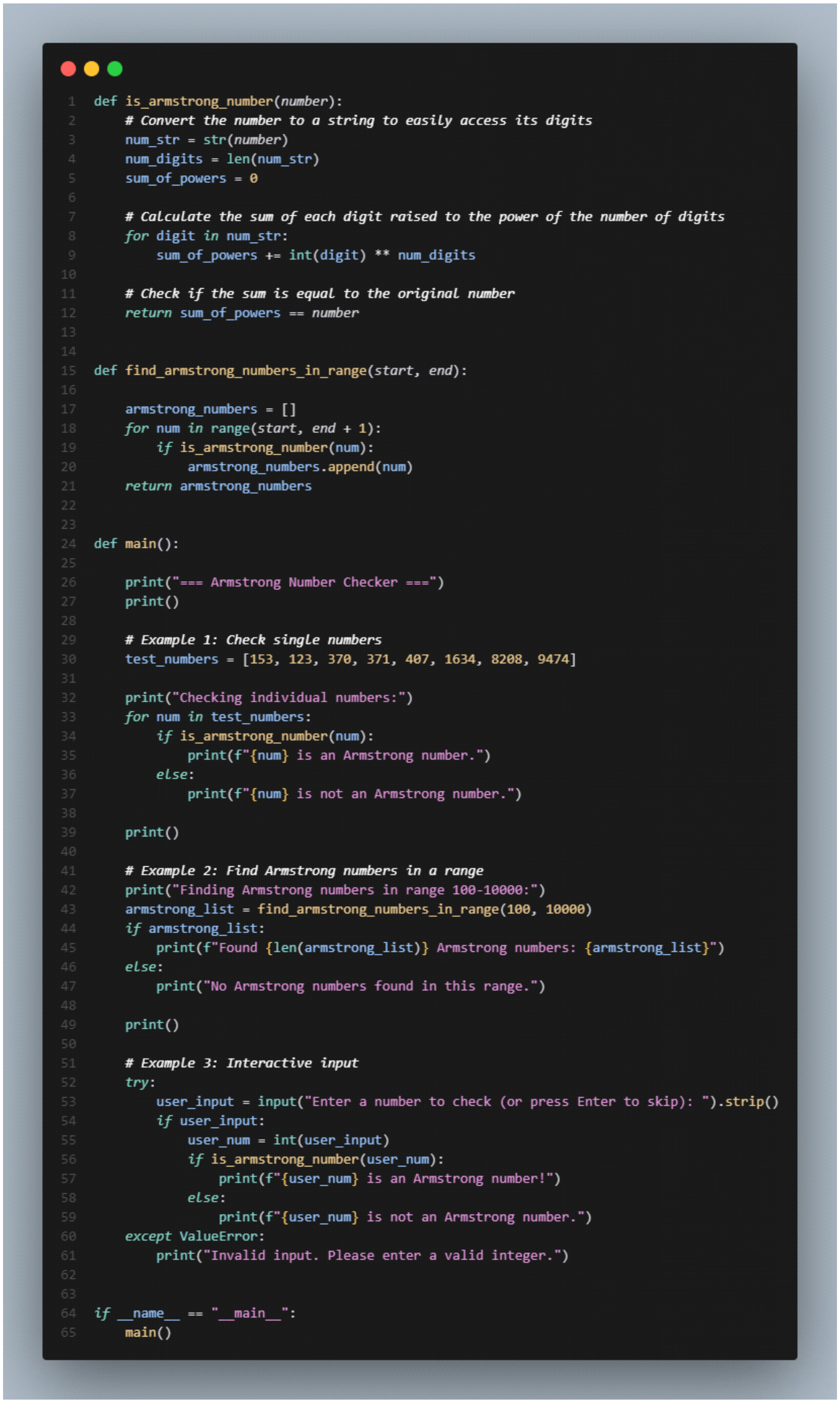
Explanation:

- def is_armstrong_number(number):: This line defines a function named is_armstrong_number that takes one argument, number, which is the integer you want to check.

- Docstring: The triple-quoted string inside the function is a docstring, explaining what the function does, what its arguments are, and what it returns. It also provides the definition of an Armstrong number.
- num_str = str(number): The input number is converted into a string (num_str). This makes it easy to iterate through the individual digits of the number.
- num_digits = len(num_str): The length of the string representation of the number is calculated and stored in num_digits. This gives us the total number of digits in the original number.
- sum_of_powers = 0: A variable sum_of_powers is initialized to 0. This variable will accumulate the sum of each digit raised to the power of the number of digits.
- for digit in num_str:: This loop iterates through each character (which represents a digit) in the num_str string.
- sum_of_powers += int(digit) ** num_digits: Inside the loop:
- int(digit): Converts the current character (digit) back into an integer.** num_digits: Raises this integer digit to the power of num_digits (the total number of digits in the original number).
- sum_of_powers += ...: Adds the result of the power calculation to the sum_of_powers variable.
- return sum_of_powers == number: After the loop finishes, the function compares the calculated sum_of_powers with the original number.
- If they are equal, the number is an Armstrong number, and the function returns True.
- If they are not equal, the number is not an Armstrong number, and the function returns False.
- Example Usage: The code outside the function demonstrates how to use the is_armstrong_number function with two example numbers (153 and 123) and prints whether each is an Armstrong number or not.

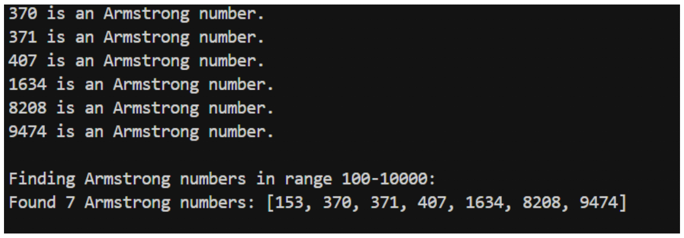
prompt:-**Ask Gemini to write a program that checks whether a number is an Armstrong number, and then modify it using Cursor AI to improve performance or structure**

code:-



```python
def is_armstrong_number(number):
    # Convert the number to a string to easily access its digits
    num_str = str(number)
    num_digits = len(num_str)
    sum_of_powers = 0

    # Calculate the sum of each digit raised to the power of the number of digits
    for digit in num_str:
        sum_of_powers += int(digit) ** num_digits

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


def find_armstrong_numbers_in_range(start, end):

    armstrong_numbers = []
    for num in range(start, end + 1):
        if is_armstrong_number(num):
            armstrong_numbers.append(num)
    return armstrong_numbers


def main():

    print("=== Armstrong Number Checker ===")
    print()

    # Example 1: Check single numbers
    test_numbers = [153, 123, 370, 371, 407, 1634, 8208, 9474]

    print("Checking individual numbers:")
    for num in test_numbers:
        if is_armstrong_number(num):
            print(f"{num} is an Armstrong number.")
        else:
            print(f"{num} is not an Armstrong number.")

    print()

    # Example 2: Find Armstrong numbers in a range
    print("Finding Armstrong numbers in range 100-10000:")
    armstrong_list = find_armstrong_numbers_in_range(100, 10000)
    if armstrong_list:
        print(f"Found {len(armstrong_list)} Armstrong numbers: {armstrong_list}")
    else:
        print("No Armstrong numbers found in this range.")

    print()

    # Example 3: Interactive input
    try:
        user_input = input("Enter a number to check (or press Enter to skip): ").strip()
        if user_input:
            user_num = int(user_input)
            if is_armstrong_number(user_num):
                print(f"{user_num} is an Armstrong number!")
            else:
                print(f"{user_num} is not an Armstrong number.")
    except ValueError:
        print("Invalid input. Please enter a valid integer.")


if __name__ == "__main__":
    main()
```

output:-



```
370 is an Armstrong number.
371 is an Armstrong number.
407 is an Armstrong number.
1634 is an Armstrong number.
8208 is an Armstrong number.
9474 is an Armstrong number.

Finding Armstrong numbers in range 100-10000:
Found 7 Armstrong numbers: [153, 370, 371, 407, 1634, 8208, 9474]
```

Explainaton:-

Performance Improvements:

1. Single string conversion: Convert to string only once instead of multiple times
2. Generator expression: Use sum() with generator expression for better memory efficiency
3. Early returns: Handle edge cases early to avoid unnecessary computation

Structural Improvements:

1. Type hints: Added proper type annotations for better code clarity
2. Comprehensive docstring: Detailed documentation with examples
3. Input validation: Proper error handling for invalid inputs
4. Edge case handling: Special handling for 0 and negative numbers

Additional Features:

1. Range finder: find_armstrong_numbers_in_range() to find all Armstrong numbers in a range
2. Detailed analysis: get_armstrong_number_info() provides step-by-step calculation
3. Interactive testing: User input functionality
4. Comprehensive test cases: Tests multiple known Armstrong numbers

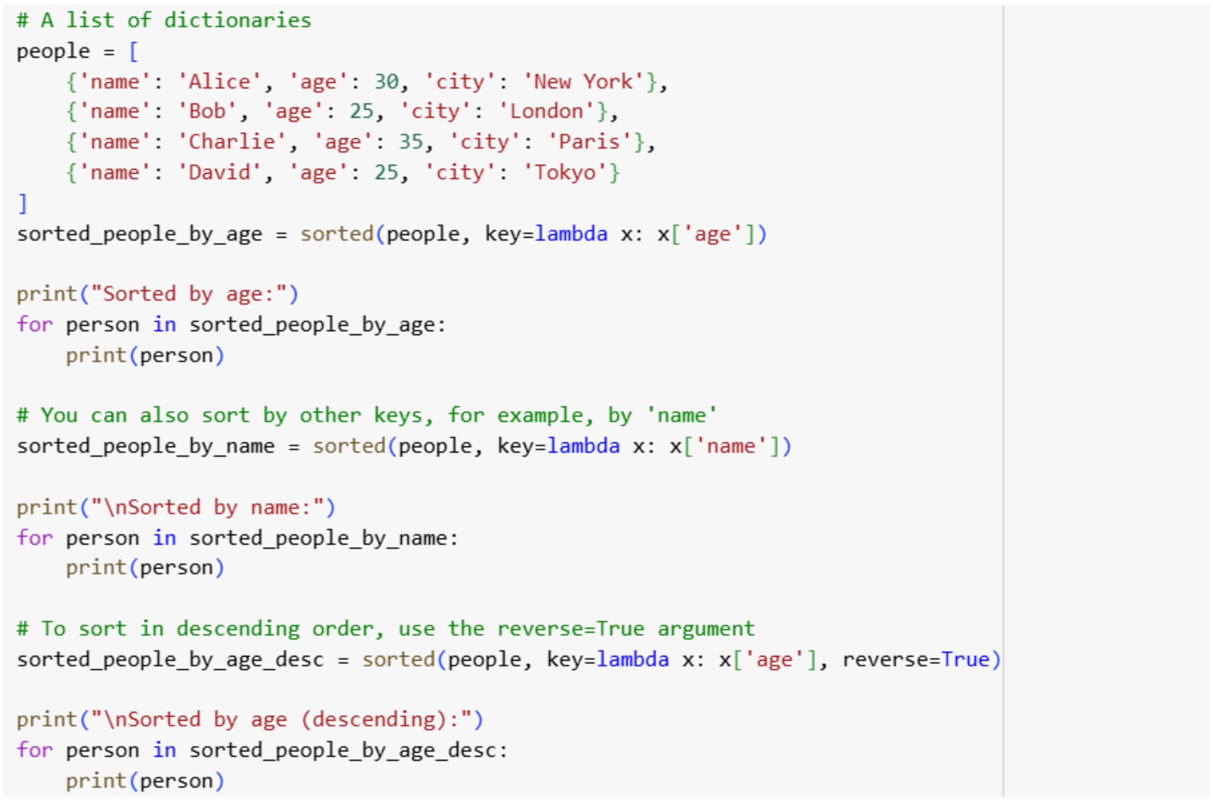
Key Performance Benefits:

- Memory efficient: Uses generator expressions instead of list comprehensions
- Faster execution: Single string conversion and optimized calculations
- Better error handling: Graceful handling of invalid inputs

- Modular design: Separate functions for different use cases

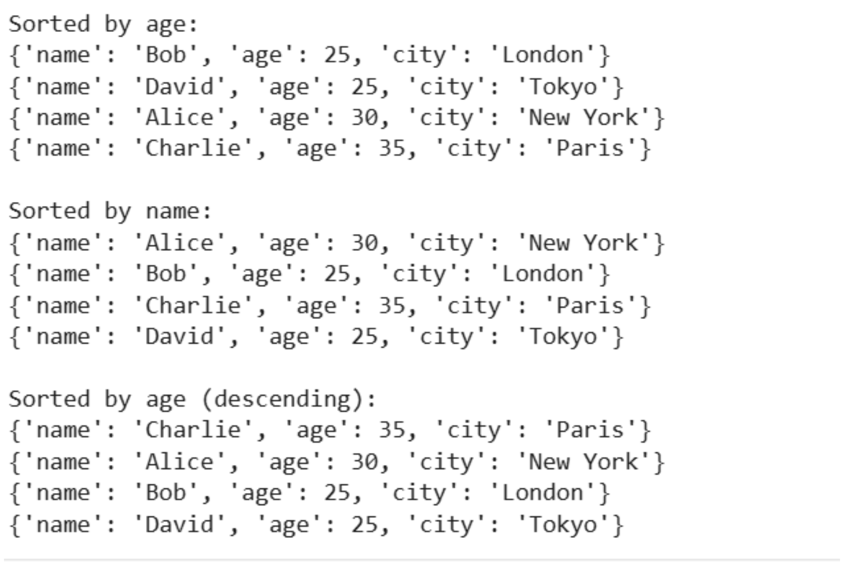
The improved version is more robust, efficient, and user-friendly while maintaining the same core functionality!

5.**prompt**:-**write a program in python for sorting a list of dictionaries by a specific Code:-**



```
# A list of dictionaries
people = [
    {'name': 'Alice', 'age': 30, 'city': 'New York'},
    {'name': 'Bob', 'age': 25, 'city': 'London'},
    {'name': 'Charlie', 'age': 35, 'city': 'Paris'},
    {'name': 'David', 'age': 25, 'city': 'Tokyo'}
]
sorted_people_by_age = sorted(people, key=lambda x: x['age'])

print("Sorted by age:")
for person in sorted_people_by_age:
    print(person)

# You can also sort by other keys, for example, by 'name'
sorted_people_by_name = sorted(people, key=lambda x: x['name'])

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

# To sort in descending order, use the reverse=True argument
sorted_people_by_age_desc = sorted(people, key=lambda x: x['age'], reverse=True)

print("\nSorted by age (descending):")
for person in sorted_people_by_age_desc:
    print(person)
```

output:-



```
Sorted by age:
{'name': 'Bob', 'age': 25, 'city': 'London'}
{'name': 'David', 'age': 25, 'city': 'Tokyo'}
{'name': 'Alice', 'age': 30, 'city': 'New York'}
{'name': 'Charlie', 'age': 35, 'city': 'Paris'}

Sorted by name:
{'name': 'Alice', 'age': 30, 'city': 'New York'}
{'name': 'Bob', 'age': 25, 'city': 'London'}
{'name': 'Charlie', 'age': 35, 'city': 'Paris'}
{'name': 'David', 'age': 25, 'city': 'Tokyo'}

Sorted by age (descending):
{'name': 'Charlie', 'age': 35, 'city': 'Paris'}
{'name': 'Alice', 'age': 30, 'city': 'New York'}
{'name': 'Bob', 'age': 25, 'city': 'London'}
{'name': 'David', 'age': 25, 'city': 'Tokyo'}
```

- **Explaination:-**
- people = [...]: This line initializes a list named people. Each element in this list is a dictionary, representing information about a person with keys like 'name', 'age', and 'city'.
- sorted(people, key=lambda x: x['age']): This is the core of the sorting operation.
- sorted(): This is a built-in Python function that returns a new sorted list from the items in an iterable. It does not modify the original list.
- people: This is the list that we want to sort.
- key=lambda x: x['age']: This is the crucial part that tells sorted() how to sort the list.
- key: This argument specifies a function to be called on each list element prior to making
- comparisons. The result of this function will be used for sorting.
- lambda x: x['age']: This is a small, anonymous function (a lambda function). It takes one argument, x, which represents each dictionary in the people list as sorted() iterates through it. The function x['age'] returns the value associated with the key 'age' from that dictionary. So, sorted() will sort the list based on the 'age' values of the dictionaries.

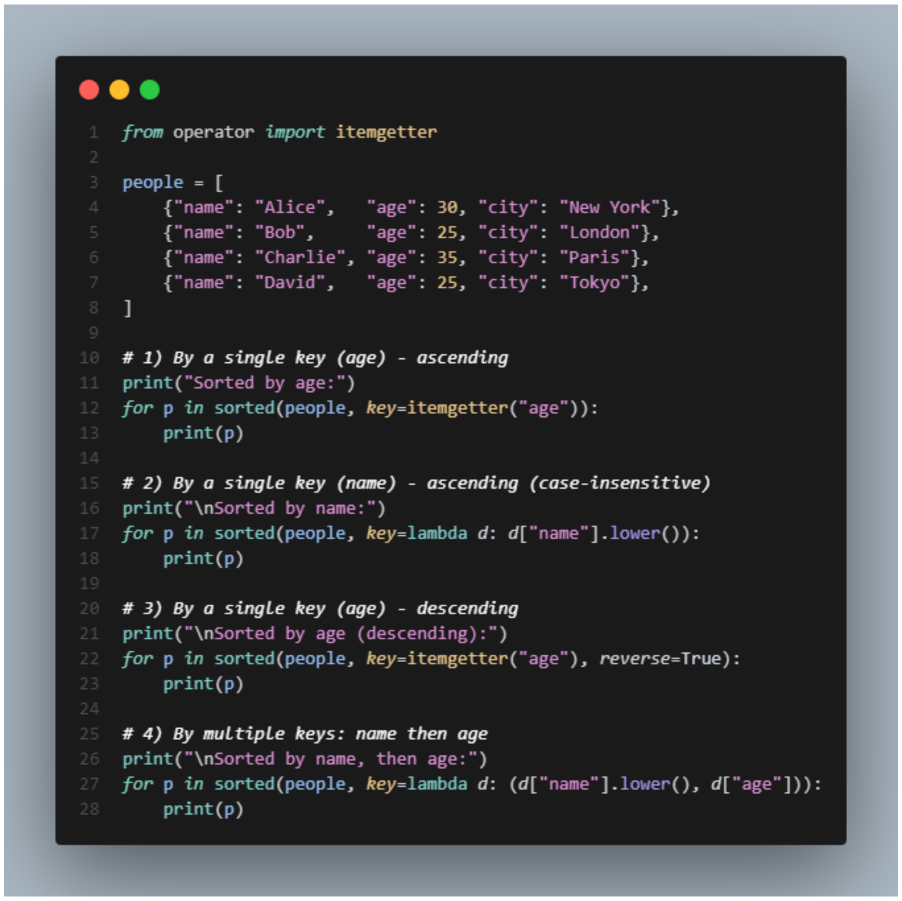
- sorted_people_by_age = ...: The new sorted list returned by sorted() is assigned to the variable sorted_people_by_age.
- Printing the sorted list: The code then iterates through the sorted_people_by_age list and prints each dictionary to show the result of the sorting.

  Sorting by 'name':

- sorted_people_by_name = sorted(people, key=lambda x: x['name']
- This is similar to sorting by age, but the key is now lambda x: x['name'], which tells sorted() to sort the list based on the 'name' values of the dictionaries.
- Sorting in descending order:
- sorted_people_by_age_desc = sorted(people, key=lambda x: x['age'], reverse=True)
- This again sorts by 'age', but the reverse=True argument is added. This tells sorted() to sort the list in descending order (from highest age to lowest age)

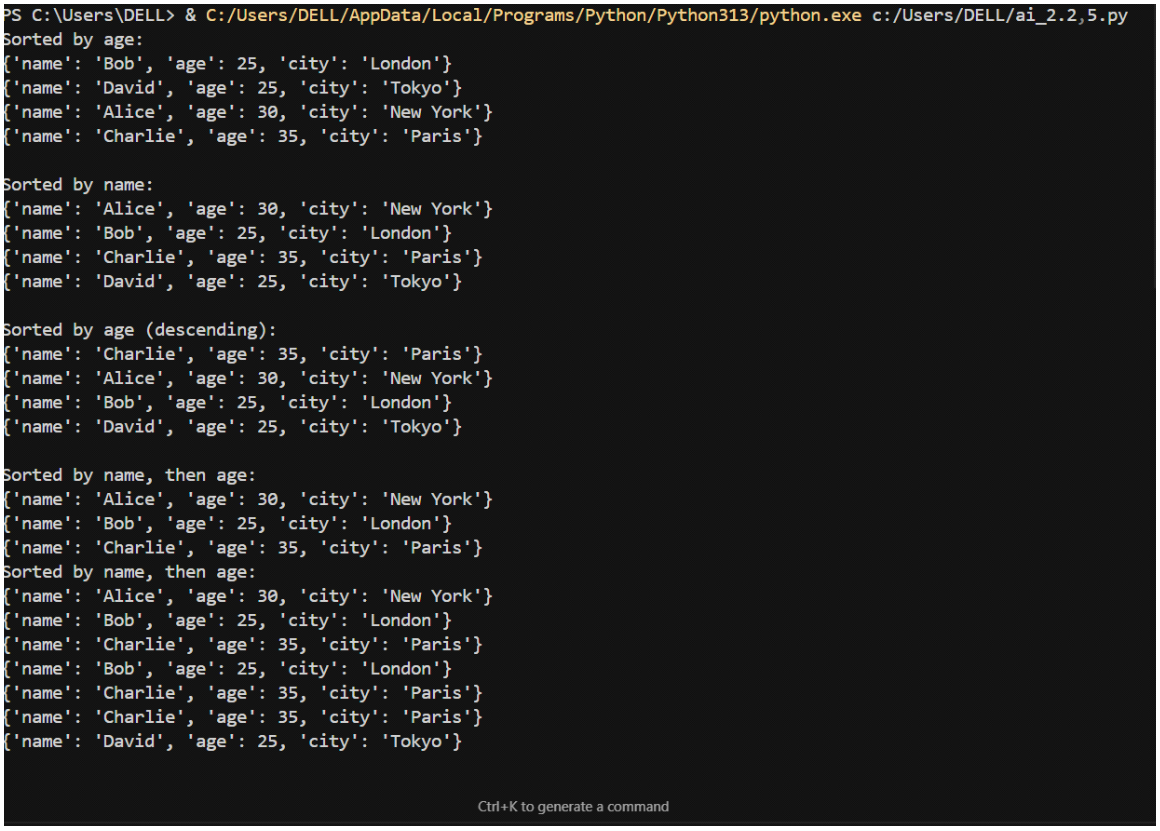
program in cursor:

code:-



```python
from operator import itemgetter

people = [
    {"name": "Alice",   "age": 30, "city": "New York"},
    {"name": "Bob",     "age": 25, "city": "London"},
    {"name": "Charlie", "age": 35, "city": "Paris"},
    {"name": "David",   "age": 25, "city": "Tokyo"},
]

# 1) By a single key (age) - ascending
print("Sorted by age:")
for p in sorted(people, key=itemgetter("age")):
    print(p)

# 2) By a single key (name) - ascending (case-insensitive)
print("\nSorted by name:")
for p in sorted(people, key=lambda d: d["name"].lower()):
    print(p)

# 3) By a single key (age) - descending
print("\nSorted by age (descending):")
for p in sorted(people, key=itemgetter("age"), reverse=True):
    print(p)

# 4) By multiple keys: name then age
print("\nSorted by name, then age:")
for p in sorted(people, key=lambda d: (d["name"].lower(), d["age"])):
    print(p)
```

output:-



```
PS C:\Users\DELL> & C:/Users/DELL/AppData/Local/Programs/Python/Python313/python.exe c:/Users/DELL/ai_2.2,5.py
Sorted by age:
{'name': 'Bob', 'age': 25, 'city': 'London'}
{'name': 'David', 'age': 25, 'city': 'Tokyo'}
{'name': 'Alice', 'age': 30, 'city': 'New York'}
{'name': 'Charlie', 'age': 35, 'city': 'Paris'}

Sorted by name:
{'name': 'Alice', 'age': 30, 'city': 'New York'}
{'name': 'Bob', 'age': 25, 'city': 'London'}
{'name': 'Charlie', 'age': 35, 'city': 'Paris'}
{'name': 'David', 'age': 25, 'city': 'Tokyo'}

Sorted by age (descending):
{'name': 'Charlie', 'age': 35, 'city': 'Paris'}
{'name': 'Alice', 'age': 30, 'city': 'New York'}
{'name': 'Bob', 'age': 25, 'city': 'London'}
{'name': 'David', 'age': 25, 'city': 'Tokyo'}

Sorted by name, then age:
{'name': 'Alice', 'age': 30, 'city': 'New York'}
{'name': 'Bob', 'age': 25, 'city': 'London'}
{'name': 'Charlie', 'age': 35, 'city': 'Paris'}
Sorted by name, then age:
{'name': 'Alice', 'age': 30, 'city': 'New York'}
{'name': 'Bob', 'age': 25, 'city': 'London'}
{'name': 'Charlie', 'age': 35, 'city': 'Paris'}
{'name': 'Bob', 'age': 25, 'city': 'London'}
{'name': 'Charlie', 'age': 35, 'city': 'Paris'}
{'name': 'Charlie', 'age': 35, 'city': 'Paris'}
{'name': 'David', 'age': 25, 'city': 'Tokyo'}
```

Explaination:-

- **'ll open your current Python files to understand what you want explained, starting with the focused one.**
- **I opened both ai_2.2,5.py and ai_2.4_.py (they contain the same code). I'll explain what the script does.**
- **What the script does**
- **Data: Defines a list of dicts people with name, age, city.**
- **Import: itemgetter("age") is equivalent to lambda d: d["age"] (slightly faster/cleaner).**
- **Sort 1 (age asc): sorted(people, key=itemgetter("age"))**
- **Sort 2 (name asc, case-insensitive): sorted(people, key=lambda d: d["name"].lower())**
- **Sort 3 (age desc): sorted(people, key=itemgetter("age"), reverse=True)**

- **Sort 4 (name then age): sorted(people, key=lambda d: (d["name"].lower(), d["age"]))**
- **The tuple key sorts by name first; if names match, it uses age as a tiebreaker.**
- **Example outcomes (given your data)**
- **By age (asc): Bob(25), David(25), Alice(30), Charlie(35)**
- **By name (asc): Alice, Bob, Charlie, David**
- **By age (desc): Charlie(35), Alice(30), Bob(25), David(25)**
- **By name, then age: Alice(30), Bob(25), Charlie(35), David(25)**
- **I explained each sort and what it prints, noting itemgetter vs lambda, reverse=True, and tuple keys for multi-key sorting.**