AI ASSISTED CODING

ASSIGMENT 2.2

PROGRAM: B.TECH/CSE

SPECIALIZATION :AIML

NAME : CHALLA SRAVYA

ENROLLMENT NO:2403A52033

COURSE : AI ASSISTED CODE

DATE:19-08-2025

BATCH :02

#TASK DESCRIPTION 1:

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

PROMPT:

Give me a code of function that filters out all negative numbers from a given list

EXPECTED OUTPUT 1:

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

```python
def filter_negative_numbers(input_list):
    """Filters out negative numbers from a list."""
    return [num for num in input_list if num >= 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 (without negative numbers): {filtered_list}")
```

```
Original list: [1, -2, 3, -4, 5, 0, -6]
Filtered list (without negative numbers): [1, 3, 5, 0]
```

EXPLANATION 1:

Please explain this code:

```
<> def filter_negative_numbers(input_list):
```

This code defines a function called `filter_negative_numbers` that takes a list as input. It uses a list comprehension to create a new list containing only the numbers from the input list that are greater than or equal to zero (i.e., not negative). The code then provides an example of how to use this function with a sample list and prints both the original and the filtered lists.

👍 👎

#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
PROMPT:

Give me a code to generate a text file and counts the frequency of each word

EXPECTED OUTPUT 2:

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

```python
# Read the text file
with open("sample_text_file.txt", "r") as f:
    text = f.read()

print("Content of the file:")
print(text)
```

```
Content of the file:
This is a sample text file.
This file is for demonstrating word frequency counting.
This text file has some words repeated.
```

```python
[10] # Generate a text file
file_content = """This is a sample text file.
This file is for demonstrating word frequency counting.
This text file has some words repeated.
"""

with open("sample_text_file.txt", "w") as f:
    f.write(file_content)

print("sample_text_file.txt generated successfully.")
```

```
sample_text_file.txt generated successfully.
```

```python
# Read the text file
with open("sample_text_file.txt", "r") as f:
    text = f.read()

print("Content of the file:")
print(text)
```

```
Content of the file:
This is a sample text file.
This file is for demonstrating word frequency counting.
This text file has some words repeated.
```

EXPLANATION 2:

`<> # Generate a text file`

This code snippet generates a text file named `sample_text_file.txt` with some predefined content.

- `file_content = """..."""`: This multiline string variable `file_content` holds the text that will be written to the file.
- `with open("sample_text_file.txt", "w") as f:`: This line opens a file named `sample_text_file.txt` in write mode (`"w"`). If the file doesn't exist, it will be created. If it exists, its contents will be overwritten. The `with` statement ensures the file is properly closed afterward. The opened file object is assigned to the variable `f`.
- `f.write(file_content)`: This line writes the content of the `file_content` variable into the opened file.
- `print("sample_text_file.txt generated successfully.")`: This line prints a confirmation message to the console indicating that the file has been created.

#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

PROMPT:

Give me a code cursor AI to generate python class called book with attributes title,author and method summary() that prints the details

EXPECTED OUTPUT 3:

Screenshot comparisons and student commentary on code clarity and performance

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

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

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

EXPLANATION 3:

This code defines a Python class called `Book`.

- `class Book:` : This line starts the definition of the class named `Book`.
- `def __init__(self, title, author):` : This is the constructor method. It's called when you create a new `Book` object. It takes `self` (which refers to the instance of the class), `title`, and `author` as arguments and initializes the `title` and `author` attributes of the object.
- `self.title = title` : This line assigns the value passed for `title` to the `title` attribute of the `Book` object.
- `self.author = author` : This line assigns the value passed for `author` to the `author` attribute of the `Book` object.
- `def summary(self):` : This defines a method named `summary` within the `Book` class. It takes `self` as an argument.
- `print(f"Title: {self.title}\nAuthor: {self.author}")` : This line inside the `summary` method prints the title and author of the book in a formatted string.
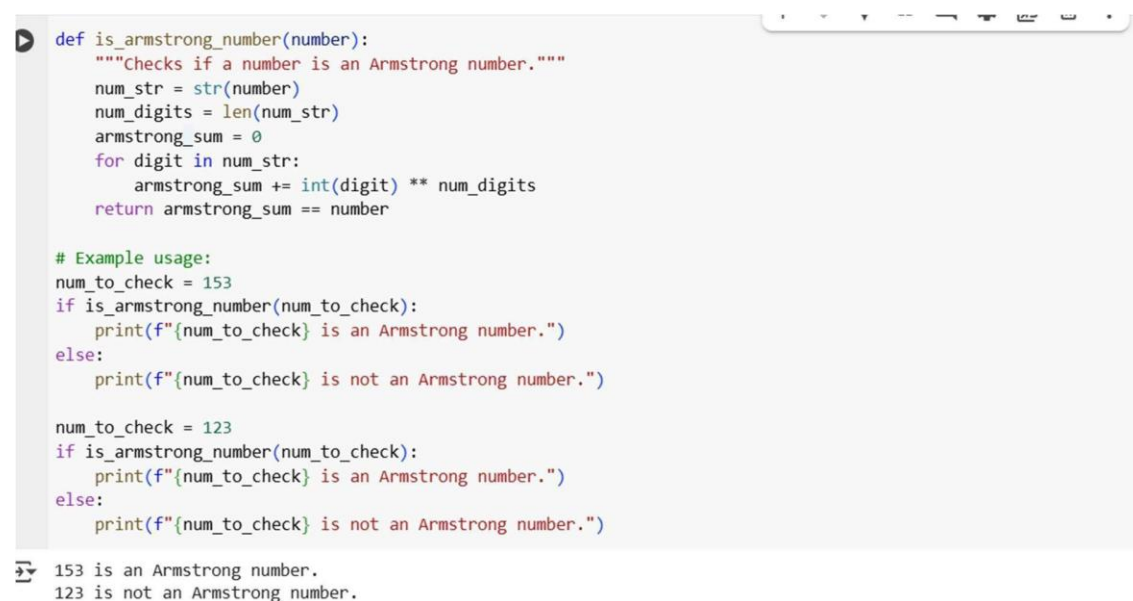
#TASK DESCRIPTION 4:

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

PROMPT:

Give me a code to generate program that checks whether a

number is an Armstrong number, and then modify it using Cursor AI to improve performance or structure

EXPECTED OUTPUT 4:

Two versions of the code with screenshots, and a summary of what changes were made by Cursor

```python
def is_armstrong_number(number):
    """Checks if a number is an Armstrong 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

# 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.")
```

```
153 is an Armstrong number.
123 is not an Armstrong number.
```

EXPLANATION 4:

1. `def is_armstrong_number(number):` : This line defines the function `is_armstrong_number` that takes one argument, `number`.
2. `"""Checks if a number is an Armstrong number."""` : This is a docstring that explains what the function does.
3. `num_str = str(number)` : The input `number` is converted to a string so that each digit can be easily accessed.
4. `num_digits = len(num_str)` : The number of digits in the input number is calculated.
5. `armstrong_sum = 0` : A variable `armstrong_sum` is initialized to 0. This variable will store the sum of each digit raised to the power of the number of digits.
6. `for digit in num_str:` : This loop iterates through each character (digit) in the `num_str` string.
7. `armstrong_sum += int(digit) ** num_digits` : Inside the loop, each `digit` (which is a string character) is converted back to an integer using `int(digit)`. This integer is then raised to the power of `num_digits` using the `**` operator, and the result is added to `armstrong_sum`.
8. `return armstrong_sum == number` : After the loop finishes, the function checks if the calculated `armstrong_sum` is equal to the original `number`. If they are equal, the function returns `True`.

#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)

PROMPT:

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

EXPECTED OUTPUT 5:

Screenshot comparisons and student commentary on code clarity and performance

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

sorted_list = sorted(list_of_dicts, key=lambda x: x['age'])

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

EXPLANATION OF CODE

key 'age'.

- `list_of_dicts = [...]`: This line initializes a list named `list_of_dicts` containing three dictionaries. Each dictionary represents a person with 'name' and 'age'.
- `sorted_list = sorted(list_of_dicts, key=lambda x: x['age'])`: This is the core of the sorting operation.
    - `sorted()` is a built-in Python function that returns a new sorted list from the items in an iterable.
    - `list_of_dicts` is the list that we want to sort.
    - `key=lambda x: x['age']` is the key function that determines the sorting order. The `lambda` function is a small anonymous function that takes one argument (`x`, which represents each dictionary in the list) and returns the value associated with the key 'age' (`x['age']`). The `sorted()` function uses these 'age' values to compare and sort the dictionaries.

- `print(sorted_list)`: This line prints the newly created `sorted_list` to the console. The list will be sorted in ascending order based on the 'age' values.