AI ASSISTED CODING

ASSIGNMENT - 6.3

Name: Varshitha. G

HtNo: 2303A51103

Task Description #1 (Loops – Automorphic Numbers in a Range)

• Task: Prompt AI to generate a function that displays all Automorphic numbers between 1 and 1000 using a for loop.

• Instructions:

o Get AI-generated code to list Automorphic numbers using a for loop.

o Analyze the correctness and efficiency of the generated logic.

o Ask AI to regenerate using a while loop and compare both implementations.

Expected Output #1:

• Correct implementation that lists Automorphic numbers using both loop types, with explanation.

```python
#Generate a function that displays all automorphic numbers between 1 to 1000 using a for loop
def is_automorphic(num):
    square=str(num**2)
    return square.endswith(str(num))
l=[]
def display_automorphic_numbers():
    for i in range(1, 1001):
        if is_automorphic(i):
            l.append(i)
display_automorphic_numbers()
print("Automorphic numbers between 1 and 1000 are:", l)

# using while loop
def is_automorphic(num):
    square=str(num**2)
    return square.endswith(str(num))
l=[]
def display_automorphic_numbers():
    i = 1
    while i <= 1000:
        if is_automorphic(i):
            l.append(i)
        i += 1
display_automorphic_numbers()
print("Automorphic numbers between 1 and 1000 are:", l)

```

```
PROBLEMS  OUTPUT  TERMINAL  PORTS
TERMINAL
625
PS C:\Users\varsh\OneDrive\ドキュメント\third\AIAC> & C:/Users/varsh/AppData/Local/Programs/Python/Pyth
Automorphic numbers between 1 and 1000 are: [1, 5, 6, 25, 76, 376, 625]
Automorphic numbers between 1 and 1000 are: [1, 5, 6, 25, 76, 376, 625]
PS C:\Users\varsh\OneDrive\ドキュメント\third\AIAC>
```

Both versions work correctly and produce the same output by finding all automorphic numbers from 1 to 1000.
Both run equally fast because they loop 1000 times, use the same logic, and store results in a list.

Time Complexity: Overall: O(n). Since n = 1000 is fixed, it behaves like O(1) for this problem.
The for loop is better because it is cleaner, easier to read, safer (no infinite loop risk), and clearly shows a fixed number of iterations.

Task Description #2 (Conditional Statements – Online Shopping Feedback Classification)

• Task: Ask AI to write nested if-elif-else conditions to classify online shopping feedback as Positive, Neutral, or Negative based on a numerical rating (1–5).

• Instructions:

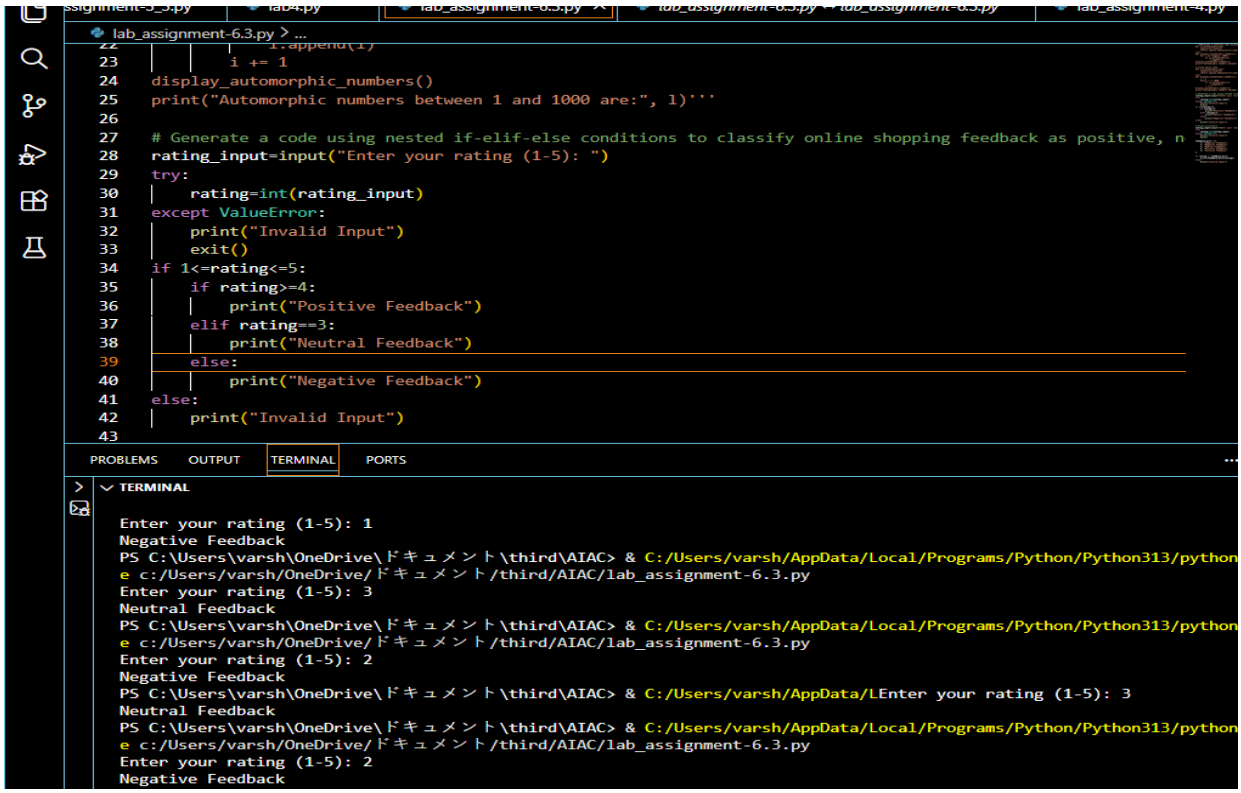
o Generate initial code using nested if-elif-else.

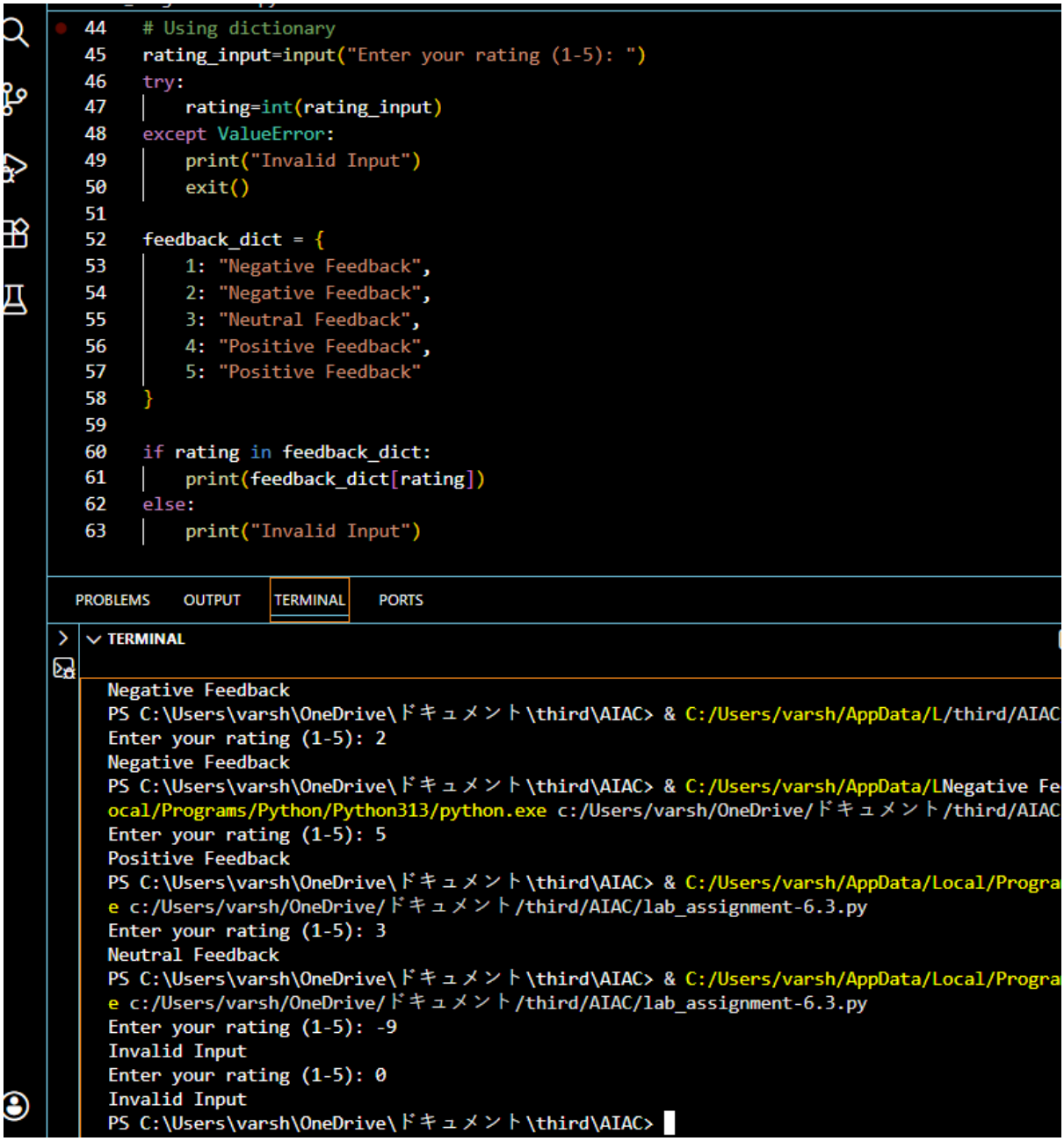
o Analyze correctness and readability.

o Ask AI to rewrite using dictionary-based or match-case structure.

Expected Output #2:

• Feedback classification function with explanation and an alternative approach.





Dictionary-based approach is more suitable for this program.

The dictionary approach is cleaner and easier to read than long if-elif-else chains.

It is faster because dictionary works in O(1) time.

It is easy to maintain—adding or changing values only needs updating the dictionary.

Task 3: Statistical_operations

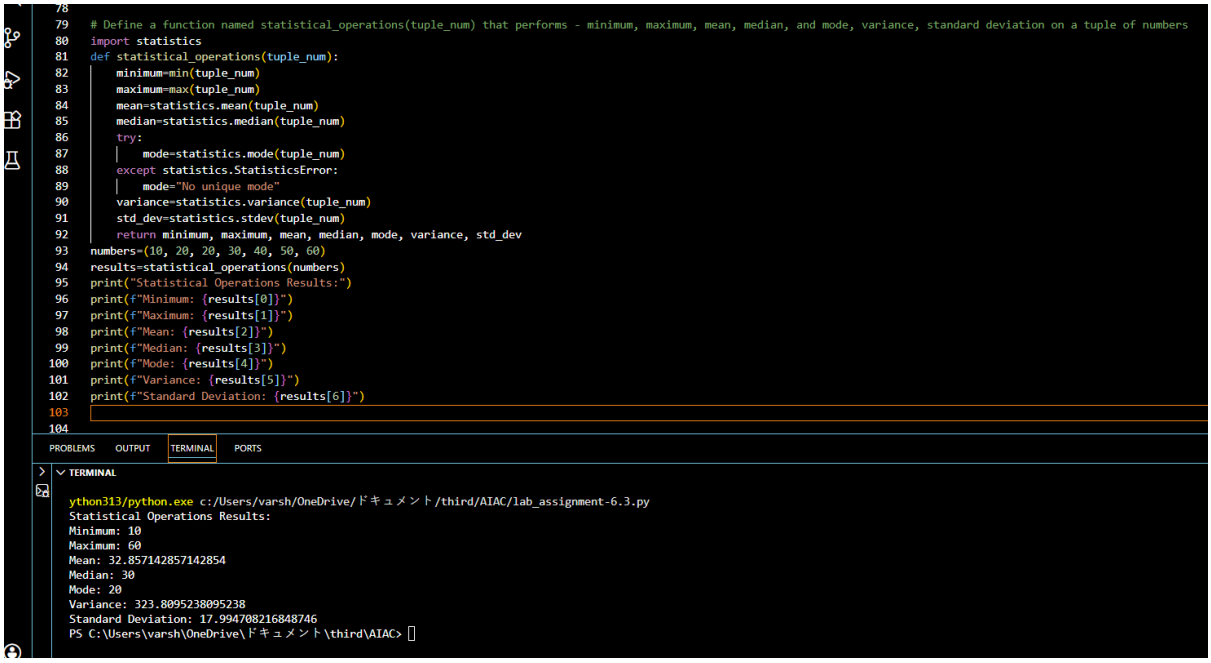
Define a function named statistical_operations(tuple_num) that performs the following statistical operations on a tuple of numbers:

• Minimum, Maximum

• Mean, Median, Mode

• Variance, Standard Deviation

While writing the function, observe the code suggestions provided by GitHub Copilot.Make decisions to accept, reject, or modify the suggestions based on their relevance and correctness



```python
# Define a function named statistical_operations(tuple_num) that performs - minimum, maximum, mean, median, and mode, variance, standard deviation on a tuple of numbers
import statistics
def statistical_operations(tuple_num):
    minimum=min(tuple_num)
    maximum=max(tuple_num)
    mean=statistics.mean(tuple_num)
    median=statistics.median(tuple_num)
    try:
        mode=statistics.mode(tuple_num)
    except statistics.StatisticsError:
        mode="No unique mode"
    variance=statistics.variance(tuple_num)
    std_dev=statistics.stdev(tuple_num)
    return minimum, maximum, mean, median, mode, variance, std_dev
numbers=(10, 20, 20, 30, 40, 50, 60)
results=statistical_operations(numbers)
print("Statistical Operations Results:")
print(f"Minimum: {results[0]}")
print(f"Maximum: {results[1]}")
print(f"Mean: {results[2]}")
print(f"Median: {results[3]}")
print(f"Mode: {results[4]}")
print(f"Variance: {results[5]}")
print(f"Standard Deviation: {results[6]}")

```

```
PROBLEMS  OUTPUT  TERMINAL  PORTS

TERMINAL
ython313/python.exe c:/Users/varsh/OneDrive/ドキュメント/third/AIAC/lab_assignment-6.3.py
Statistical Operations Results:
Minimum: 10
Maximum: 60
Mean: 32.857142857142854
Median: 30
Mode: 20
Variance: 323.8095238095238
Standard Deviation: 17.994708216848746
PS C:\Users\varsh\OneDrive\ドキュメント\third\AIAC>
```

Task 4: Teacher Profile

• Prompt: Create a class Teacher with attributes teacher_id, name, subject, and experience. Add a method to display teacher details.

• Expected Output: Class with initializer, method, and object creation.

```python

# Create a class teacher with attributes teacher_id, name, subject, and experience. Add a method to display teacher details.
class Teacher:
    def __init__(self, teacher_id, name, subject, experience):
        self.teacher_id = teacher_id
        self.name = name
        self.subject = subject
        self.experience = experience

    def display_details(self):
        print(f"Teacher ID: {self.teacher_id}")
        print(f"Name: {self.name}")
        print(f"Subject: {self.subject}")
        print(f"Experience: {self.experience} years")
teacher1 = Teacher(1, "Alice Smith", "Mathematics", 10)
teacher1.display_details()
teacher2 = Teacher(2, "Bob Johnson", "Science", 8)
teacher2.display_details()
teacher3 = Teacher(3, "Cathy Brown", "English", 5)
teacher3.display_details()
```

```
PROBLEMS  OUTPUT  TERMINAL  PORTS

TERMINAL

PS C:\Users\varsh\OneDrive\ドキュメント\third\AIAC> & C:/Users/varsh/AppData/Local/Programs/Python/Python313/python.exe c:/Users/varsh/OneDrive/
ird/AIAC/lab_assignment-6.3.py
Teacher ID: 1
Name: Alice Smith
Subject: Mathematics
Experience: 10 years
Teacher ID: 2
Name: Bob Johnson
Subject: Science
Experience: 8 years
Teacher ID: 3
Name: Cathy Brown
Subject: English
Experience: 5 years
PS C:\Users\varsh\OneDrive\ドキュメント\third\AIAC>
```

Task #5 – Zero-Shot Prompting with Conditional Validation

Use zero-shot prompting to instruct an AI tool to generate a function that validates an Indian mobile number.

Requirements: The function must ensure the mobile number: o Starts with 6, 7, 8, or 9 , Contains exactly 10 digits

Expected Output: A valid Python function that performs all required validations without using any input-output examples in the prompt.

```python

# Generate a function that validates an Indian mobile number (10 digits, starts with 6, 7, 8, or 9)
import re
def validate_mobile_number(mobile_number):
    pattern = r'^[6-9]\d{9}$'
    if re.match(pattern, mobile_number):
        return "Valid Indian Mobile Number"
    else:
        return "Invalid Indian Mobile Number"
mobile_number_input = input("Enter an Indian mobile number: ")
result = validate_mobile_number(mobile_number_input)
print(result)
```

```
PROBLEMS  OUTPUT  TERMINAL  PORTS

TERMINAL

Enter an Indian mobile number: 6123045879
Valid Indian Mobile Number
PS C:\Users\varsh\OneDrive\ドキュメント\third\AIAC> & C:/Users/varsh/AppData/Local/Programs/Python/Python313/python.exe c:/U
Drive/ドキュメント/third/AIAC/lab_assignment-6.3.py
Enter an Indian mobile number: 00000000000
Invalid Indian Mobile Number
PS C:\Users\varsh\OneDrive\ドキュメント\third\AIAC> & C:/Users/varsh/AppData/Local/Programs/Python/Python313/python.exe c:/U
Drive/ドキュメント/third/AIAC/lab_assignment-6.3.py
Enter an Indian mobile number: 5
Invalid Indian Mobile Number
PS C:\Users\varsh\OneDrive\ドキュメント\third\AIAC>
```

Task Description #6 (Loops – Armstrong Numbers in a Range)

Task: Write a function using AI that finds all Armstrong numbers in a user-specified range (e.g., 1 to 1000).

Instructions:

• Use a for loop and digit power logic.

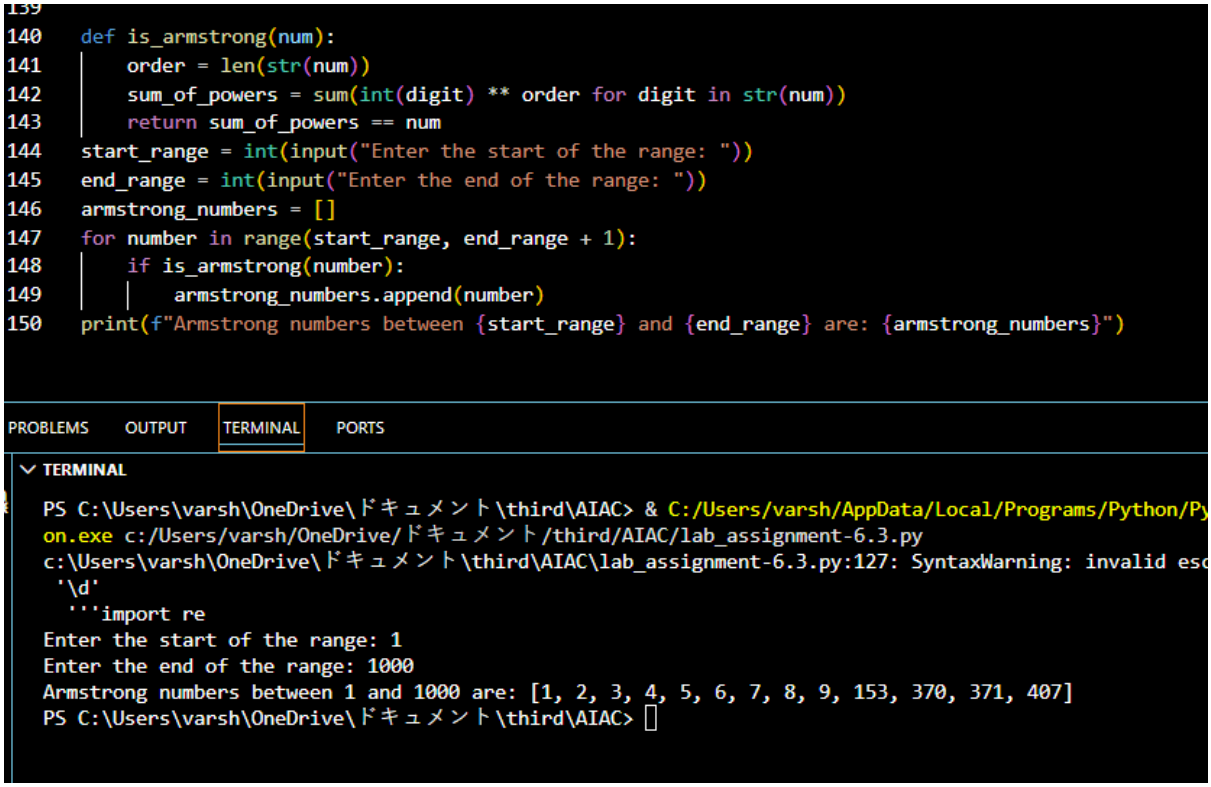
• Validate correctness by checking known Armstrong numbers (153, 370,etc.).

• Ask AI to regenerate an optimized version (using list comprehensions).

Expected Output #7:

• Python program listing Armstrong numbers in the range.

• Optimized version with explanation.



```python
def find_armstrong_numbers_generator(start, end):
    """
    Memory-efficient version using generator.
    Yields Armstrong numbers one at a time instead of storing all in memory.
    """
    for num in range(start, end + 1):
        if is_armstrong(num):
            yield num

# Input with validation
while True:
    try:
        start_range = int(input("Enter the start of the range: "))
        end_range = int(input("Enter the end of the range: "))
        if start_range > end_range:
            print("Error: Start range should be less than or equal to end range.")
            continue
        break
    except ValueError:
        print("Error: Please enter valid integers.")

# Using optimized list comprehension approach
armstrong_numbers = find_armstrong_numbers(start_range, end_range)
print(f"Armstrong numbers between {start_range} and {end_range} are: {armstrong_numbers}")

# Alternative: Memory-efficient generator approach (uncomment to use)
# armstrong_numbers_gen = find_armstrong_numbers_generator(start_range, end_range)
# print(f"Armstrong numbers between {start_range} and {end_range} are: {list(armstrong_numbers_gen)}")
```

```
PROBLEMS  OUTPUT  TERMINAL  PORTS

TERMINAL
    '''import re
Enter the start of the range: 1
Enter the end of the range: 1000
Armstrong numbers between 1 and 1000 are: [1, 2, 3, 4, 5, 6, 7, 8, 9, 153, 370, 371, 407]
PS C:\Users\varsh\OneDrive\ドキュメント\third\AIAC>
```

Input validation: Avoids errors when the range is invalid or reversed.

Separate function: find_armstrong_numbers() improves reusability and keeps code organized.

Error handling: try-except ensures the program doesn't go wrong on wrong input.

Comments make the logic easy to understand and maintain.

Task Description #7 (Loops – Happy Numbers in a Range)

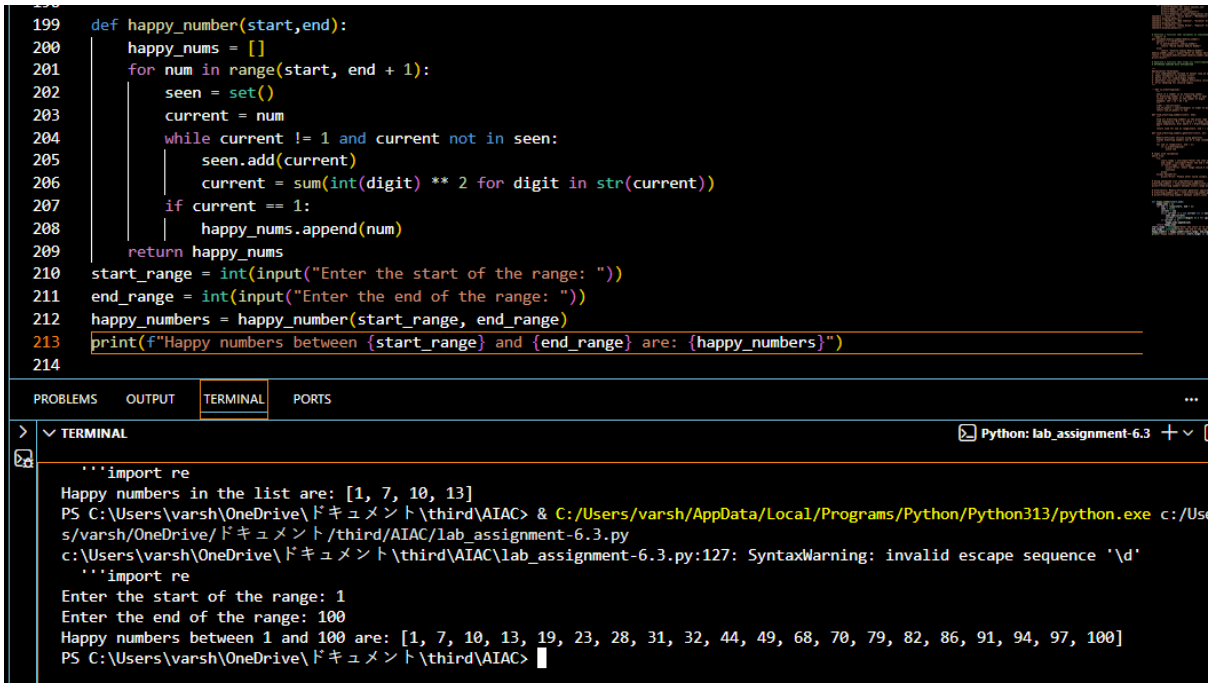
Task: Generate a function using AI that displays all Happy Numbers within a user-specified range (e.g., 1 to 500).

Instructions:

• Implement the logic using a loop: repeatedly replace a number with the sum of the squares of its digits until the result is either 1 (Happy Number) or enters a cycle (Not Happy).

• Validate correctness by checking known Happy Numbers (e.g., 1, 7, 10, 13, 19, 23, 28…).

• Ask AI to regenerate an optimized version (e.g., by using a set to detect cycles instead of infinite loops).



```
def happy_number(start,end):
    happy_nums = []
    for num in range(start, end + 1):
        seen = set()
        current = num
        while current != 1 and current not in seen:
            seen.add(current)
            current = sum(int(digit) ** 2 for digit in str(current))
        if current == 1:
            happy_nums.append(num)
    return happy_nums
start_range = int(input("Enter the start of the range: "))
end_range = int(input("Enter the end of the range: "))
happy_numbers = happy_number(start_range, end_range)
print(f"Happy numbers between {start_range} and {end_range} are: {happy_numbers}")

PROBLEMS  OUTPUT  TERMINAL  PORTS
TERMINAL                                   Python: lab_assignment-6.3
  '''import re
Happy numbers in the list are: [1, 7, 10, 13]
PS C:\Users\varsh\OneDrive\ドキュメント\third\AIAC> & C:/Users/varsh/AppData/Local/Programs/Python/Python313/python.exe c:/Us
s/varsh/OneDrive/ドキュメント/third/AIAC/lab_assignment-6.3.py
c:\Users\varsh\OneDrive\ドキュメント\third\AIAC\lab_assignment-6.3.py:127: SyntaxWarning: invalid escape sequence '\d'
  '''import re
Enter the start of the range: 1
Enter the end of the range: 100
Happy numbers between 1 and 100 are: [1, 7, 10, 13, 19, 23, 28, 31, 32, 44, 49, 68, 70, 79, 82, 86, 91, 94, 97, 100]
PS C:\Users\varsh\OneDrive\ドキュメント\third\AIAC>
```

Expected Output #8

• Python program that prints all Happy Numbers within a range.

• Optimized version using cycle detection with explanation.

```python
def is_happy(num, seen=None):
    """
    Check if a number is happy using set-based cycle detection.

    Args:
        num: The number to check
        seen: Set tracking visited numbers in sequence

    Returns:
        True if happy (reaches 1), False if unhappy (cycle detected)
    Time Complexity: O(log n) - sequence converges quickly
    Space Complexity: O(1) - set size is bounded (typically < 20 elements)
    """
    if seen is None:
        seen = set()
    # Base case: reached 1 (happy number)
    if num == 1:
        return True
    # Cycle detected: number appears again in sequence (unhappy)
    if num in seen:
        return False
    # Add current number to set to track visited numbers
    seen.add(num)
    # Compute sum of squares of digits
    next_num = sum(int(digit) ** 2 for digit in str(num))
    # Recursively check next number with same set
    return is_happy(next_num, seen)
def find_happy_numbers_optimized(numbers_list):
    """
    Find all happy numbers in a list using list comprehension and cycle detection.
    Args:
        numbers_list: List of numbers to check
    Returns:
        List of happy numbers
    Time Complexity: O(n * log m) where n = list size, m = avg number value
    Space Complexity: O(k) where k = happy numbers found
    """
    return [num for num in numbers_list if is_happy(num)]
# Input validation: Get range from user
while True:
    try:
        start_range = int(input("Enter the start of the range: "))
        end_range = int(input("Enter the end of the range: "))

        # Validate range
        if start_range > end_range:
            print("Error: Start range should be less than or equal to end range.")
            continue
        if start_range < 1:
            print("Error: Range must contain positive integers.")
            continue
        break
    except ValueError:
        print("Error: Please enter valid integers.")
# Generate list from range
numbers_list = list(range(start_range, end_range + 1))
# Find happy numbers using optimized function
happy_numbers = find_happy_numbers_optimized(numbers_list)
print(f"\nHappy numbers between {start_range} and {end_range} are: {happy_numbers}")
# Example trace for unhappy number (2):
print("\nExample - Cycle detection for 2:")
print("2 → 4 → 16 → 37 → 58 → 89 → 145 → 42 → 20 → 4 (CYCLE DETECTED! Set prevents infinite loop)")
print("\nExample - Happy number sequence for 7:")
print("7 → 49 → 97 → 130 → 10 → 1 (HAPPY! Reaches 1)")
```

Cycle detection using a set: Already-visited numbers are stored, if a number repeats, it means a loop and the number is not happy.

Using a separate is_happy() function makes the code cleaner and easier to read.

Shorter and faster way to build lists compared to manual append().

Ensures only valid, positive numbers and correct ranges are processed.

Each number finishes quickly because digit-square sums reduce fast, avoiding long loops.

Task Description #8 (Loops – Strong Numbers in a Range)

Task: Generate a function using AI that displays all Strong Numbers (sum of factorial of digits equals the number, e.g., 145 = 1!+4!+5!) within a given range.

Instructions:

• Use loops to extract digits and calculate factorials.

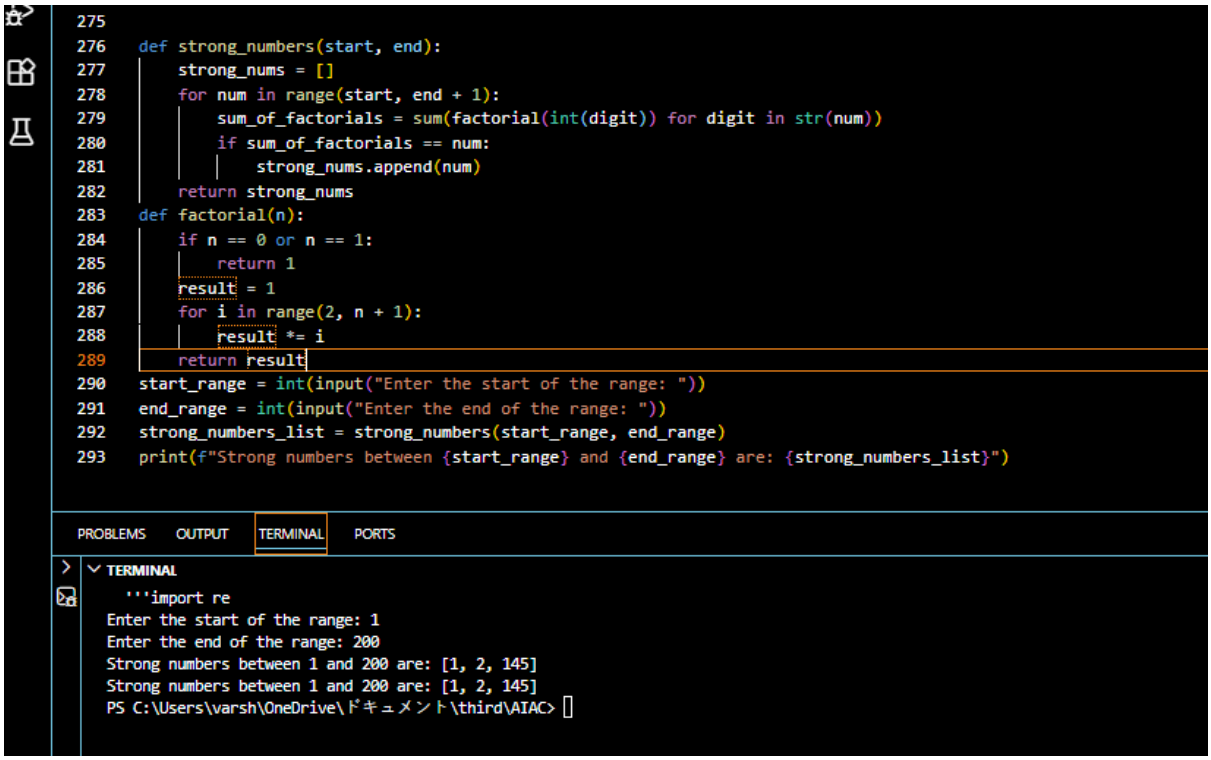
• Validate with examples (1, 2, 145).

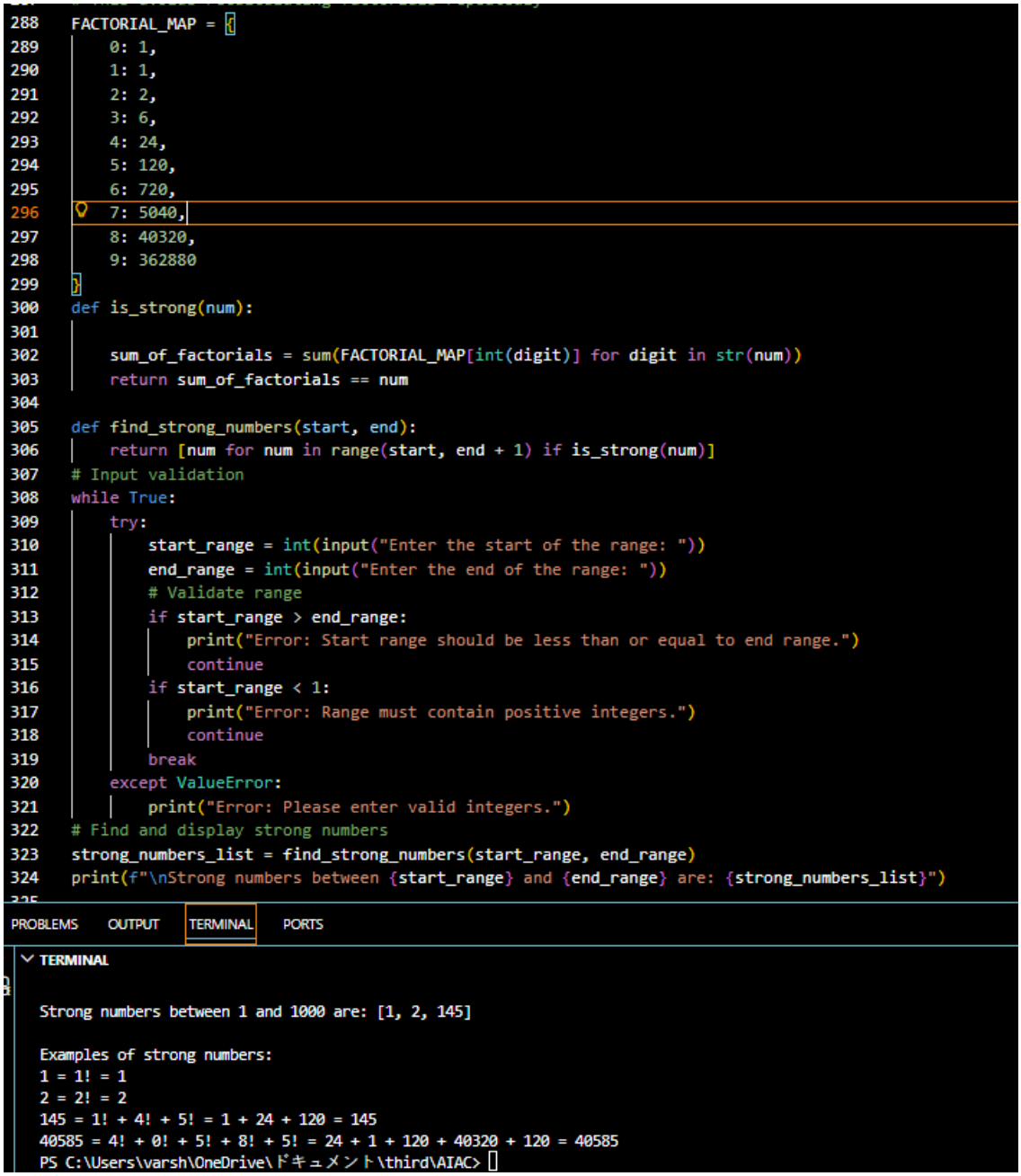
• Ask AI to regenerate an optimized version (precompute digit factorials).

Expected Output #9:

• Python program that lists Strong Numbers.

• Optimized version with explanation.





```
FACTORIAL_MAP = {
    0: 1,
    1: 1,
    2: 2,
    3: 6,
    4: 24,
    5: 120,
    6: 720,
    7: 5040,
    8: 40320,
    9: 362880
}
def is_strong(num):

    sum_of_factorials = sum(FACTORIAL_MAP[int(digit)] for digit in str(num))
    return sum_of_factorials == num

def find_strong_numbers(start, end):
    return [num for num in range(start, end + 1) if is_strong(num)]
# Input validation
while True:
    try:
        start_range = int(input("Enter the start of the range: "))
        end_range = int(input("Enter the end of the range: "))
        # Validate range
        if start_range > end_range:
            print("Error: Start range should be less than or equal to end range.")
            continue
        if start_range < 1:
            print("Error: Range must contain positive integers.")
            continue
        break
    except ValueError:
        print("Error: Please enter valid integers.")
# Find and display strong numbers
strong_numbers_list = find_strong_numbers(start_range, end_range)
print(f"\nStrong numbers between {start_range} and {end_range} are: {strong_numbers_list}")
```

PROBLEMS OUTPUT TERMINAL PORTS

TERMINAL

```
Strong numbers between 1 and 1000 are: [1, 2, 145]

Examples of strong numbers:
1 = 1! = 1
2 = 2! = 2
145 = 1! + 4! + 5! = 1 + 24 + 120 = 145
40585 = 4! + 0! + 5! + 8! + 5! = 24 + 1 + 120 + 40320 + 120 = 40585
PS C:\Users\varsh\OneDrive\ドキュメント\third\AIAC>
```

Precomputed factorials: Factorials of digits 0–9 are calculated once and stored, avoiding repeated work.

Factorials are fetched directly from a dictionary in O(1) time instead of recalculating.

Cleaner and faster way to build the result list than using loops.

Checks the range and handles errors to prevent incorrect inputs.

Task #9 – Few-Shot Prompting for Nested Dictionary Extraction

Objective: Use few-shot prompting (2–3 examples) to instruct the AI to create a function that parses a nested dictionary representing student information.

Requirements: • The function should extract and return:Full Name, Branch, SGPA

Expected Output: A reusable Python function that correctly navigates and extracts values from nested dictionaries based on the provided examples

```
'''

'''
{
    "name": {"first": "Aarav", "last": "Sharma"},
    "branch": "CSE",
    "sgpa": 9.1
}
display ("Aarav Sharma", "CSE", 9.1)

{
    "student": {
        "name": {"first": "Neha", "last": "Patel"},
        "details": {
            "branch": "ECE",
            "sgpa": 8.6
        }
    }
}
display ("Neha Patel", "ECE", 8.6)'''
def student_info(student_dict):
    try:
        if "student" in student_dict:
            first_name = student_dict["student"]["name"]["first"]
            last_name = student_dict["student"]["name"]["last"]
            branch = student_dict["student"]["details"]["branch"]
            sgpa = student_dict["student"]["details"]["sgpa"]
        else:
            first_name = student_dict["name"]["first"]
            last_name = student_dict["name"]["last"]
            branch = student_dict["branch"]
            sgpa = student_dict["sgpa"]
        full_name = f"{first_name} {last_name}"
        print(f"Name: {full_name}, Branch: {branch}, SGPA: {sgpa}")
    except KeyError as e:
        print(f"Missing key in dictionary: {e}")
student1 = {
    "name": {"first": "Aarav", "last": "Sharma"},
    "branch": "CSE",
    "sgpa": 9.1
}
student2 = {
    "student": {
        "name": {"first": "Neha", "last": "Patel"},
        "details": {
            "branch": "ECE",
            "sgpa": 8.6
        }
    }
}
student_info(student1)
```

```
PROBLEMS  OUTPUT  TERMINAL  PORTS
TERMINAL

Name: Aarav Sharma, Branch: CSE, SGPA: 9.1
Name: Neha Patel, Branch: ECE, SGPA: 8.6
Name: Neha Patel, Branch: ECE, SGPA: 8.6
```

Task Description #10 (Loops – Perfect Numbers in a Range)

Task: Generate a function using AI that displays all Perfect Numbers within a user-specified range (e.g., 1 to 1000).

Instructions:

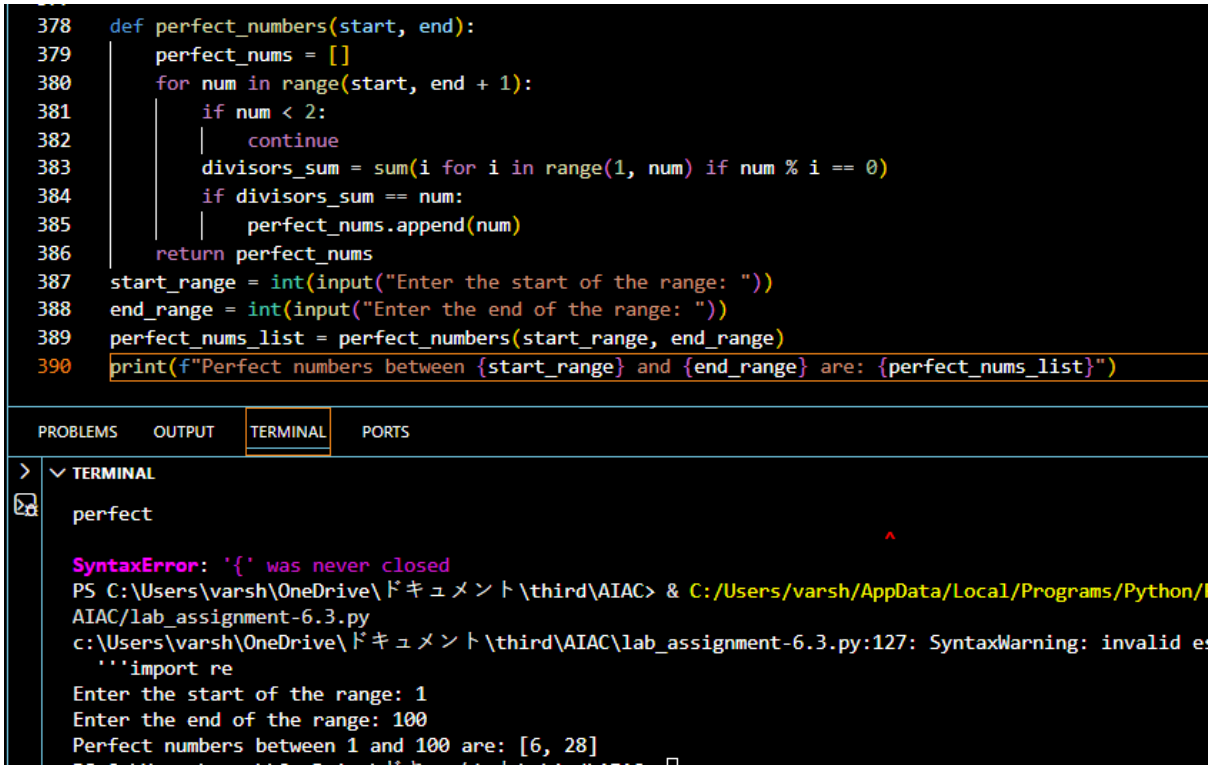
• A Perfect Number is a positive integer equal to the sum of its proper divisors (excluding itself).

o Example: 6 = 1 + 2 + 3, 28 = 1 + 2 + 4 + 7 + 14.

• Use a for loop to find divisors of each number in the range.

• Validate correctness with known Perfect Numbers (6, 28, 496…).

• Ask AI to regenerate an optimized version (using divisor check only up to √n).



```python
def is_perfect(num):
    if num < 2:
        return False
    divisors_sum = 1
    i = 2
    while i * i <= num:
        if num % i == 0:
            divisors_sum += i
            if i != num // i:
                divisors_sum += num // i
        i += 1
    return divisors_sum == num
def find_perfect_numbers(start, end):
    return [num for num in range(start, end + 1) if is_perfect(num)]
while True:
    try:
        start_range = int(input("Enter the start of the range: "))
        end_range = int(input("Enter the end of the range: "))

        # Validate range
        if start_range > end_range:
            print("Error: Start range should be less than or equal to end range.")
            continue
        if start_range < 1:
            print("Error: Range must contain positive integers.")
            continue
        break
    except ValueError:
        print("Error: Please enter valid integers.")
# Find and display perfect numbers
perfect_nums_list = find_perfect_numbers(start_range, end_range)
print(f"\nPerfect numbers between {start_range} and {end_range} are: {perfect_nums_list}")
```

```
PROBLEMS  OUTPUT  TERMINAL  PORTS

TERMINAL
Enter the start of the range: 1
Enter the end of the range: 1000

Perfect numbers between 1 and 1000 are: [6, 28, 496]
PS C:\Users\varsh\OneDrive\ドキュメント\third\AIAC>
```

Square root optimization: Divisors are checked only up to √n, making the code much faster.

When a number divides n, both the divisor and its pair (n / i) are counted together.

Ensures the square root is added only once for perfect squares.

Cleaner and faster way to collect results than manual loops.

Handles invalid inputs safely and avoids runtime errors.