AI ASSISTED CODING

ASSIGNMENT-6.3

NAME: G.V.N Hasini

H.T.NO: 2303A51099

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:

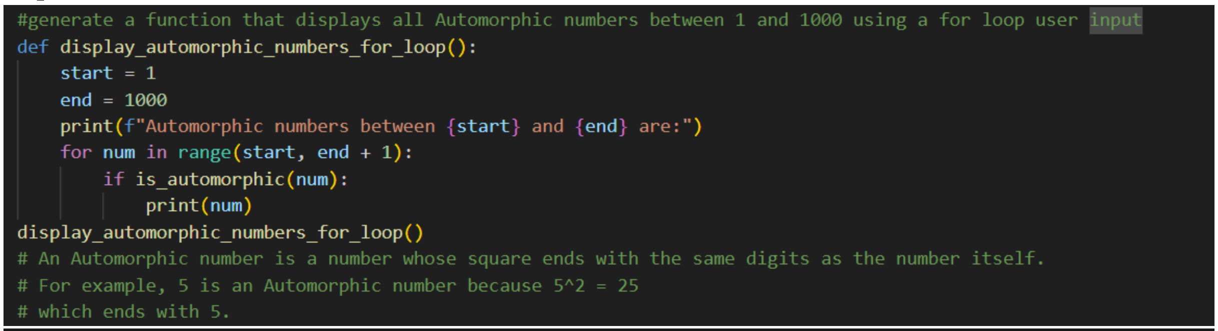
o Get AI-generated code to list Automorphic numbers using a for loop.

o Analyze the correctness and efficiency of the generated logic.

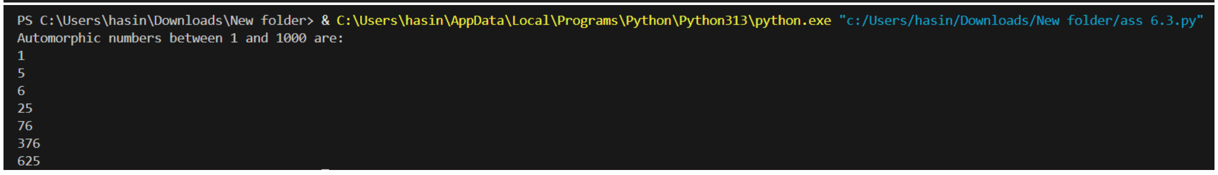
o Ask AI to regenerate using a while loop and compare both implementations.

Expected Output #1:

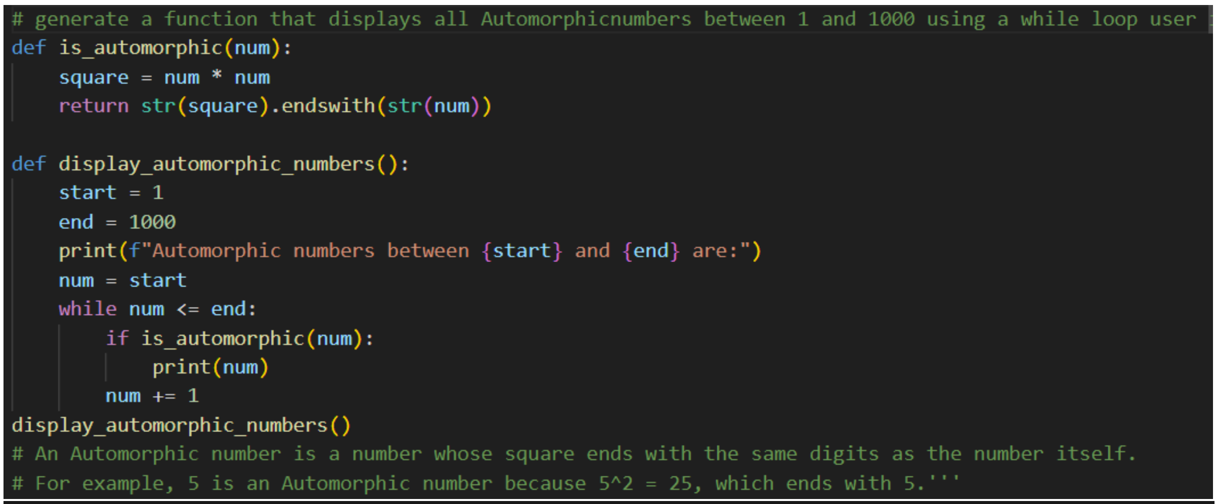
• Correct implementation that lists Automorphic numbers using both loop types, with explanation.



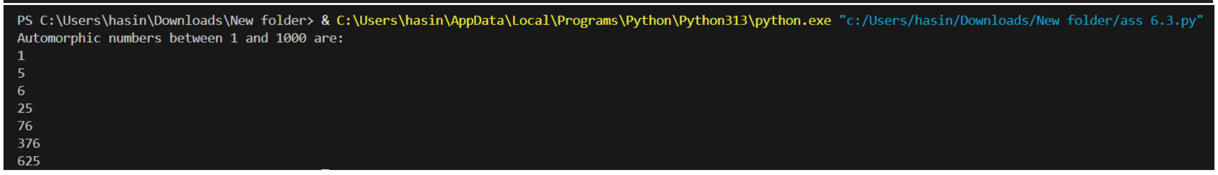
```python
#generate a function that displays all Automorphic numbers between 1 and 1000 using a for loop user input
def display_automorphic_numbers_for_loop():
    start = 1
    end = 1000
    print(f"Automorphic numbers between {start} and {end} are:")
    for num in range(start, end + 1):
        if is_automorphic(num):
            print(num)
display_automorphic_numbers_for_loop()
# An Automorphic number is a number whose square ends with the same digits as the number itself.
# For example, 5 is an Automorphic number because 5^2 = 25
# which ends with 5.
```



```
PS C:\Users\hasin\Downloads\New folder> & C:\Users\hasin\AppData\Local\Programs\Python\Python313\python.exe "c:/Users/hasin/Downloads/New folder/ass 6.3.py"
Automorphic numbers between 1 and 1000 are:
1
5
6
25
76
376
625
```



```python
# generate a function that displays all Automorphicnumbers between 1 and 1000 using a while loop user
def is_automorphic(num):
    square = num * num
    return str(square).endswith(str(num))

def display_automorphic_numbers():
    start = 1
    end = 1000
    print(f"Automorphic numbers between {start} and {end} are:")
    num = start
    while num <= end:
        if is_automorphic(num):
            print(num)
        num += 1
display_automorphic_numbers()
# An Automorphic number is a number whose square ends with the same digits as the number itself.
# For example, 5 is an Automorphic number because 5^2 = 25, which ends with 5.'''
```



```
PS C:\Users\hasin\Downloads\New folder> & C:\Users\hasin\AppData\Local\Programs\Python\Python313\python.exe "c:/Users/hasin/Downloads/New folder/ass 6.3.py"
Automorphic numbers between 1 and 1000 are:
1
5
6
25
76
376
625
```

1. Both methods take about the same time because each number is checked once and string conversion adds a little extra work, so time complexity is O(n log n).
2. They use very little memory since only a small list of automorphic numbers is stored.
3. A for loop is faster because it is optimized inside Python.
4. A for loop makes the code cleaner and easier to understand because no manual counter is needed.
5. A for loop is safer and less error-prone since it avoids skipping numbers or running into infinite loops.

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:

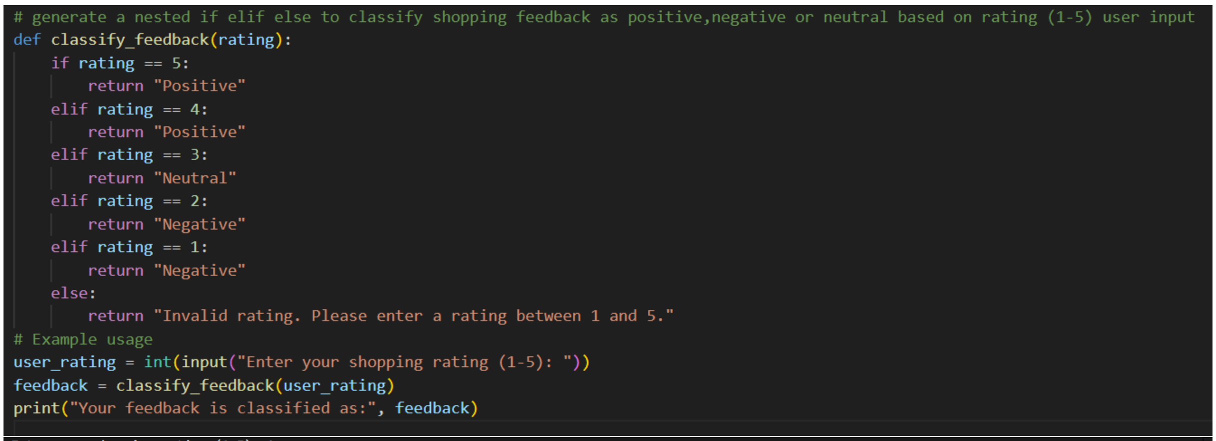
o Generate initial code using nested if-elif-else.

o Analyze correctness and readability.

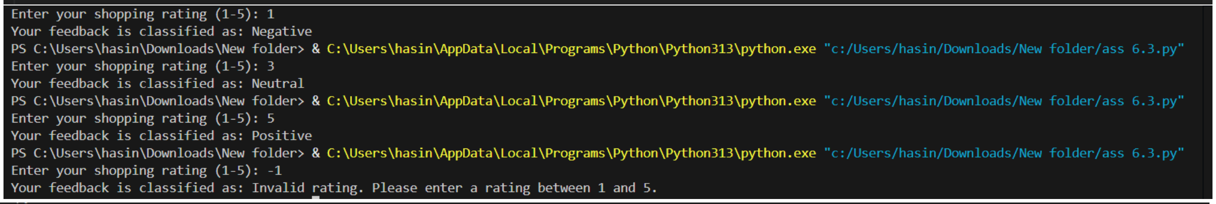
o Ask AI to rewrite using dictionary-based or match-case structure.

Expected Output #2:

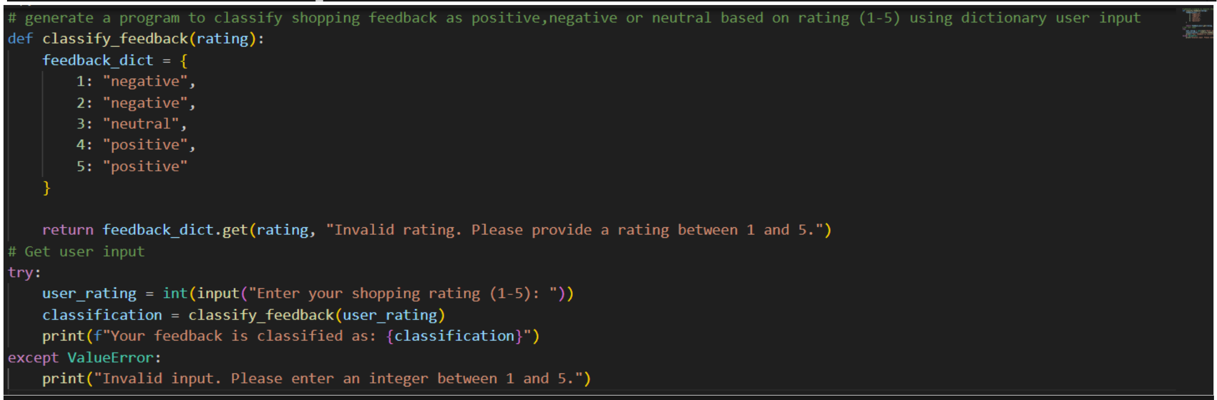
• Feedback classification function with explanation and an alternative approach.



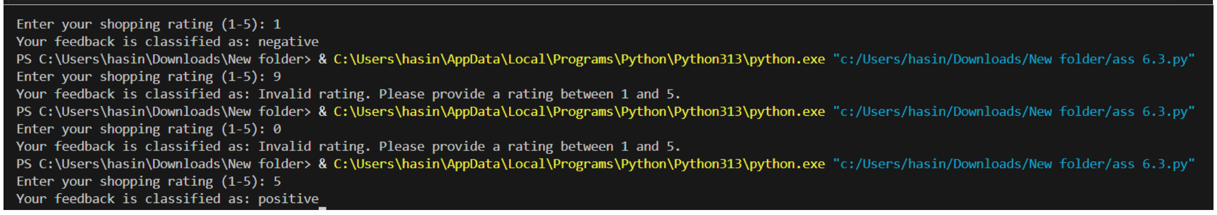
```python
# generate a nested if elif else to classify shopping feedback as positive,negative or neutral based on rating (1-5) user input
def classify_feedback(rating):
    if rating == 5:
        return "Positive"
    elif rating == 4:
        return "Positive"
    elif rating == 3:
        return "Neutral"
    elif rating == 2:
        return "Negative"
    elif rating == 1:
        return "Negative"
    else:
        return "Invalid rating. Please enter a rating between 1 and 5."
# Example usage
user_rating = int(input("Enter your shopping rating (1-5): "))
feedback = classify_feedback(user_rating)
print("Your feedback is classified as:", feedback)
```



```
Enter your shopping rating (1-5): 1
Your feedback is classified as: Negative
PS C:\Users\hasin\Downloads\New folder> & C:\Users\hasin\AppData\Local\Programs\Python\Python313\python.exe "c:/Users/hasin/Downloads/New folder/ass 6.3.py"
Enter your shopping rating (1-5): 3
Your feedback is classified as: Neutral
PS C:\Users\hasin\Downloads\New folder> & C:\Users\hasin\AppData\Local\Programs\Python\Python313\python.exe "c:/Users/hasin/Downloads/New folder/ass 6.3.py"
Enter your shopping rating (1-5): 5
Your feedback is classified as: Positive
PS C:\Users\hasin\Downloads\New folder> & C:\Users\hasin\AppData\Local\Programs\Python\Python313\python.exe "c:/Users/hasin/Downloads/New folder/ass 6.3.py"
Enter your shopping rating (1-5): -1
Your feedback is classified as: Invalid rating. Please enter a rating between 1 and 5.
```



```python
# generate a program to classify shopping feedback as positive,negative or neutral based on rating (1-5) using dictionary user input
def classify_feedback(rating):
    feedback_dict = {
        1: "negative",
        2: "negative",
        3: "neutral",
        4: "positive",
        5: "positive"
    }

    return feedback_dict.get(rating, "Invalid rating. Please provide a rating between 1 and 5.")
# Get user input
try:
    user_rating = int(input("Enter your shopping rating (1-5): "))
    classification = classify_feedback(user_rating)
    print(f"Your feedback is classified as: {classification}")
except ValueError:
    print("Invalid input. Please enter an integer between 1 and 5.")
```



```
Enter your shopping rating (1-5): 1
Your feedback is classified as: negative
PS C:\Users\hasin\Downloads\New folder> & C:\Users\hasin\AppData\Local\Programs\Python\Python313\python.exe "c:/Users/hasin/Downloads/New folder/ass 6.3.py"
Enter your shopping rating (1-5): 9
Your feedback is classified as: Invalid rating. Please provide a rating between 1 and 5.
PS C:\Users\hasin\Downloads\New folder> & C:\Users\hasin\AppData\Local\Programs\Python\Python313\python.exe "c:/Users/hasin/Downloads/New folder/ass 6.3.py"
Enter your shopping rating (1-5): 0
Your feedback is classified as: Invalid rating. Please provide a rating between 1 and 5.
PS C:\Users\hasin\Downloads\New folder> & C:\Users\hasin\AppData\Local\Programs\Python\Python313\python.exe "c:/Users/hasin/Downloads/New folder/ass 6.3.py"
Enter your shopping rating (1-5): 5
Your feedback is classified as: positive
```

1. A dictionary makes things clear because all rating-to-feedback pairs are visible in one place.
2. It is easy to maintain since you can add or change ratings without changing the main logic.
3. It handles invalid ratings more safely by letting you check or give a default message.
4. It follows Python's standard and clean way of mapping values, so the code looks natural.
5. It stays clean and readable even if the number of rating categories increases.

Task 3: Statistical_operations

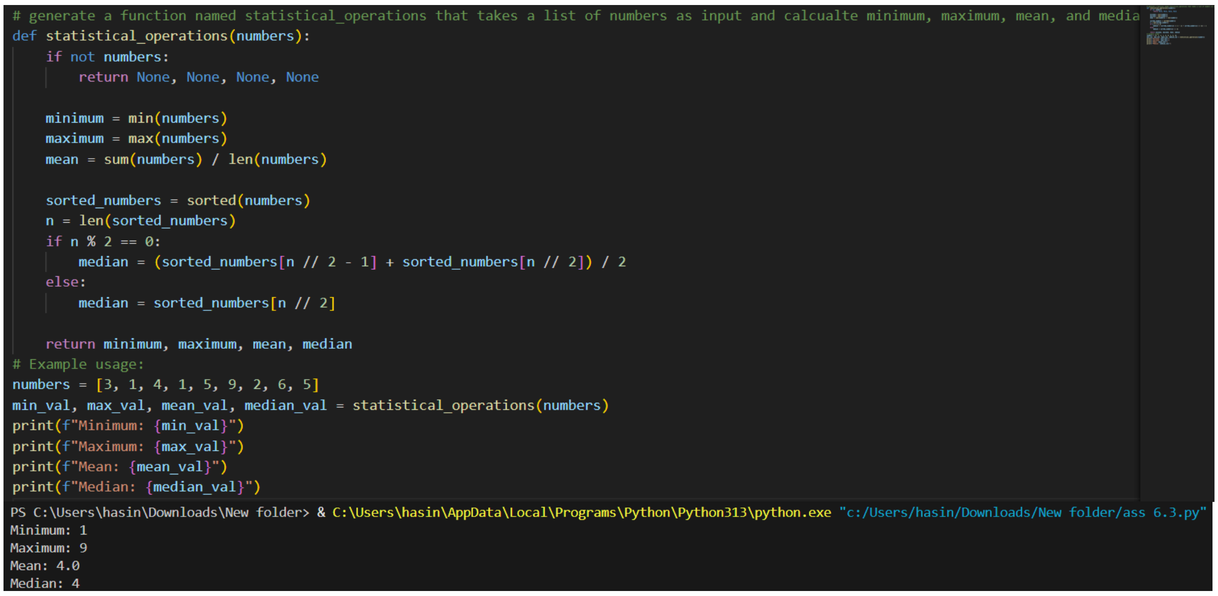
Define a function named statistical_operations(tuple_num) that performs the following statistical operations on a tuple of numbers:

• 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



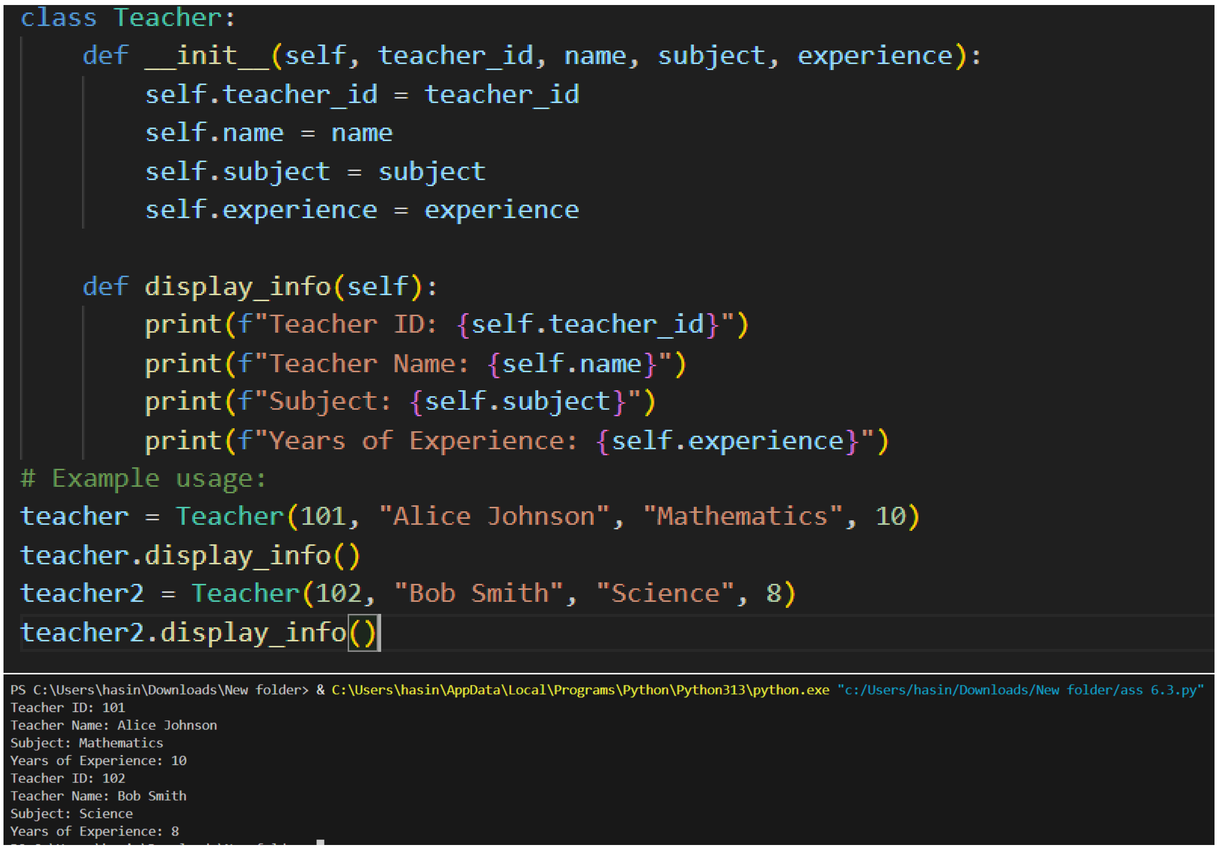
```
# generate a function named statistical_operations that takes a list of numbers as input and calcualte minimum, maximum, mean, and media
def statistical_operations(numbers):
    if not numbers:
        return None, None, None, None

    minimum = min(numbers)
    maximum = max(numbers)
    mean = sum(numbers) / len(numbers)

    sorted_numbers = sorted(numbers)
    n = len(sorted_numbers)
    if n % 2 == 0:
        median = (sorted_numbers[n // 2 - 1] + sorted_numbers[n // 2]) / 2
    else:
        median = sorted_numbers[n // 2]

    return minimum, maximum, mean, median
# Example usage:
numbers = [3, 1, 4, 1, 5, 9, 2, 6, 5]
min_val, max_val, mean_val, median_val = statistical_operations(numbers)
print(f"Minimum: {min_val}")
print(f"Maximum: {max_val}")
print(f"Mean: {mean_val}")
print(f"Median: {median_val}")
```

```
PS C:\Users\hasin\Downloads\New folder> & C:\Users\hasin\AppData\Local\Programs\Python\Python313\python.exe "c:/Users/hasin/Downloads/New folder/ass 6.3.py"
Minimum: 1
Maximum: 9
Mean: 4.0
Median: 4
```

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
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_info(self):
        print(f"Teacher ID: {self.teacher_id}")
        print(f"Teacher Name: {self.name}")
        print(f"Subject: {self.subject}")
        print(f"Years of Experience: {self.experience}")
# Example usage:
teacher = Teacher(101, "Alice Johnson", "Mathematics", 10)
teacher.display_info()
teacher2 = Teacher(102, "Bob Smith", "Science", 8)
teacher2.display_info()
```

```
PS C:\Users\hasin\Downloads\New folder> & C:\Users\hasin\AppData\Local\Programs\Python\Python313\python.exe "c:/Users/hasin/Downloads/New folder/ass 6.3.py"
Teacher ID: 101
Teacher Name: Alice Johnson
Subject: Mathematics
Years of Experience: 10
Teacher ID: 102
Teacher Name: Bob Smith
Subject: Science
Years of Experience: 8
```

Task #5 – Zero-Shot Prompting with Conditional Validation

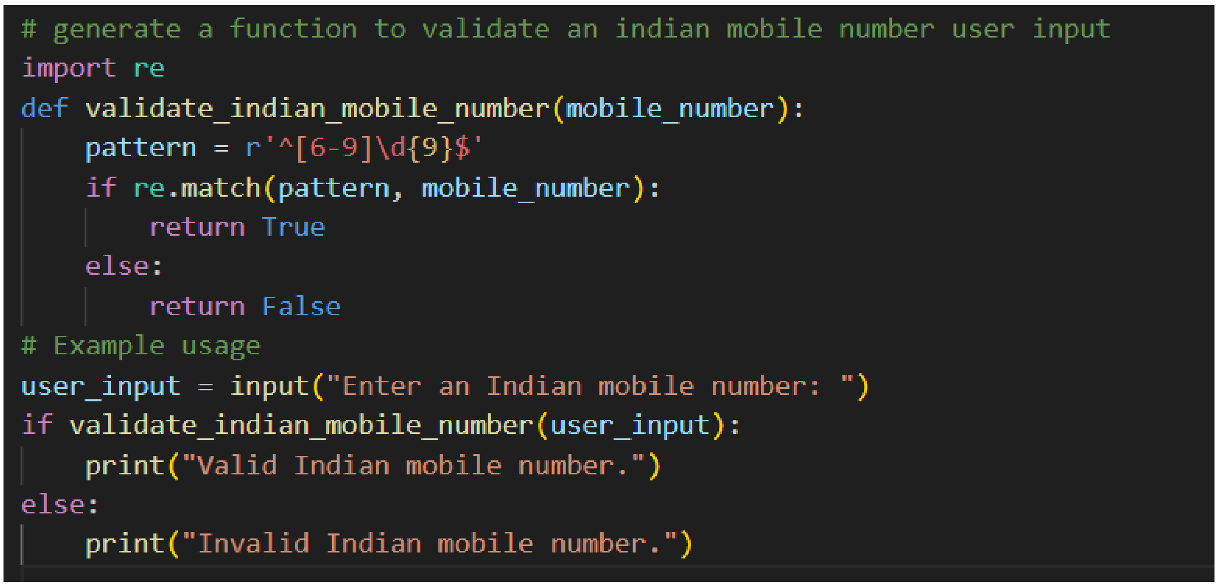
Use zero-shot prompting to instruct an AI tool to generate a function that validates an Indian mobile number.

Requirements

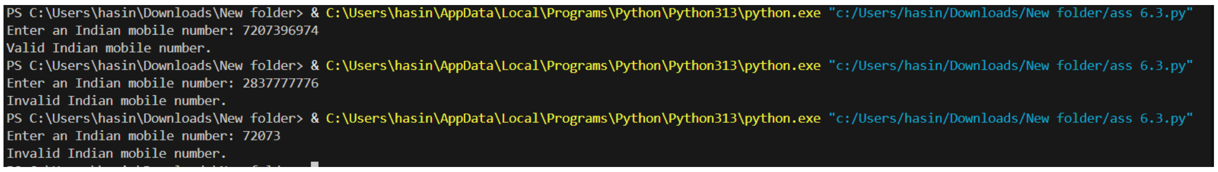
• The function must ensure the mobile number:

o Starts with 6, 7, 8, or 9

o Contains exactly 10 digits



```python
# generate a function to validate an indian mobile number user input
import re
def validate_indian_mobile_number(mobile_number):
    pattern = r'^[6-9]\d{9}$'
    if re.match(pattern, mobile_number):
        return True
    else:
        return False
# Example usage
user_input = input("Enter an Indian mobile number: ")
if validate_indian_mobile_number(user_input):
    print("Valid Indian mobile number.")
else:
    print("Invalid Indian mobile number.")
```


```
PS C:\Users\hasin\Downloads\New folder> & C:\Users\hasin\AppData\Local\Programs\Python\Python313\python.exe "c:/Users/hasin/Downloads/New folder/ass 6.3.py"
Enter an Indian mobile number: 7207396974
Valid Indian mobile number.
PS C:\Users\hasin\Downloads\New folder> & C:\Users\hasin\AppData\Local\Programs\Python\Python313\python.exe "c:/Users/hasin/Downloads/New folder/ass 6.3.py"
Enter an Indian mobile number: 2837777776
Invalid Indian mobile number.
PS C:\Users\hasin\Downloads\New folder> & C:\Users\hasin\AppData\Local\Programs\Python\Python313\python.exe "c:/Users/hasin/Downloads/New folder/ass 6.3.py"
Enter an Indian mobile number: 72073
Invalid Indian mobile number.
```

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.

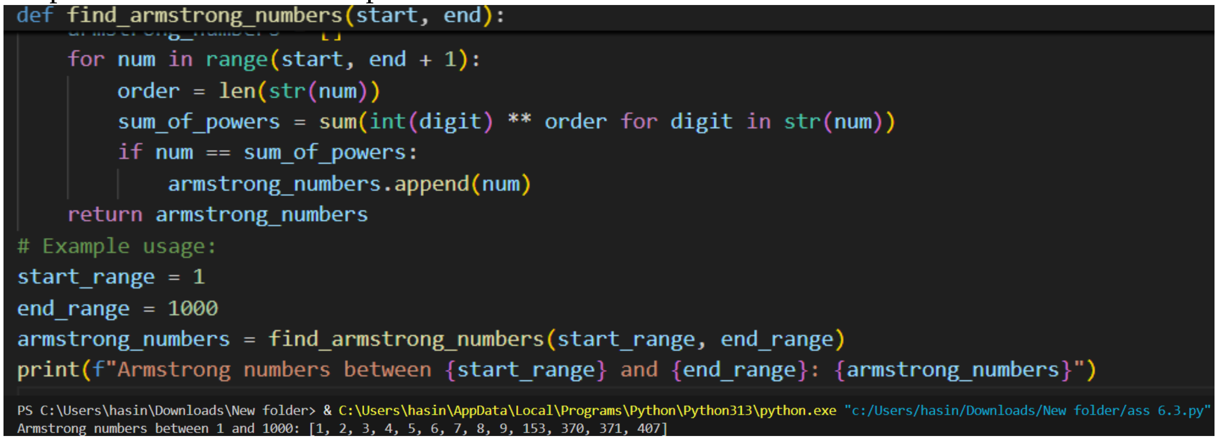
• Validate correctness by checking known Armstrong numbers (153, 370, etc.).

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

Expected Output #7:

• Python program listing Armstrong numbers in the range.

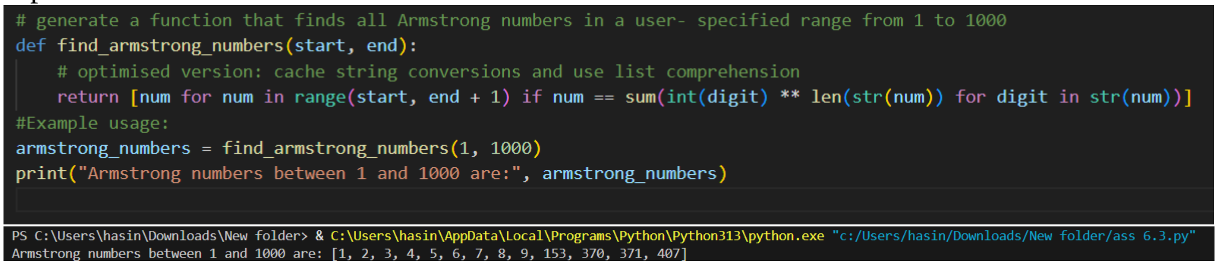
• Optimized version with explanation.



```
def find_armstrong_numbers(start, end):
    for num in range(start, end + 1):
        order = len(str(num))
        sum_of_powers = sum(int(digit) ** order for digit in str(num))
        if num == sum_of_powers:
            armstrong_numbers.append(num)
    return armstrong_numbers
# Example usage:
start_range = 1
end_range = 1000
armstrong_numbers = find_armstrong_numbers(start_range, end_range)
print(f"Armstrong numbers between {start_range} and {end_range}: {armstrong_numbers}")
```

```
PS C:\Users\hasin\Downloads\New folder> & C:\Users\hasin\AppData\Local\Programs\Python\Python313\python.exe "c:/Users/hasin/Downloads/New folder/ass 6.3.py"
Armstrong numbers between 1 and 1000: [1, 2, 3, 4, 5, 6, 7, 8, 9, 153, 370, 371, 407]
```

Optimized code:



```
# generate a function that finds all Armstrong numbers in a user- specified range from 1 to 1000
def find_armstrong_numbers(start, end):
    # optimised version: cache string conversions and use list comprehension
    return [num for num in range(start, end + 1) if num == sum(int(digit) ** len(str(num)) for digit in str(num))]
#Example usage:
armstrong_numbers = find_armstrong_numbers(1, 1000)
print("Armstrong numbers between 1 and 1000 are:", armstrong_numbers)
```

```
PS C:\Users\hasin\Downloads\New folder> & C:\Users\hasin\AppData\Local\Programs\Python\Python313\python.exe "c:/Users/hasin/Downloads/New folder/ass 6.3.py"
Armstrong numbers between 1 and 1000 are: [1, 2, 3, 4, 5, 6, 7, 8, 9, 153, 370, 371, 407]
```

1. It does the same work in fewer lines, so there is less overhead and faster execution.
2. It calculates the digit powers directly instead of using extra variables, reducing unnecessary steps.

3. List comprehension is faster than manually appending values in a loop.
4. It avoids storing extra temporary values, so memory usage stays low.
5. The code is cleaner and easier to read while giving the same correct result.

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).

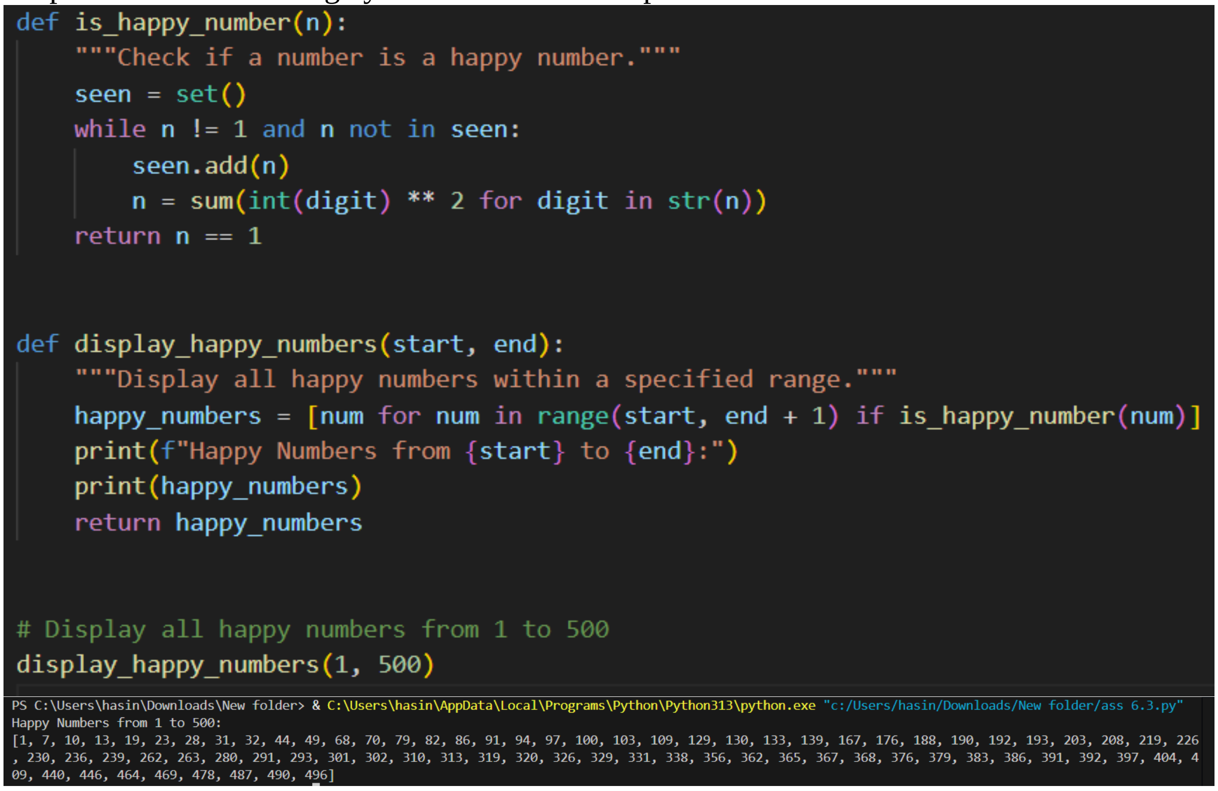
• Validate correctness by checking known Happy Numbers (e.g., 1, 7, 10,13, 19, 23, 28…).

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

Expected Output #8:

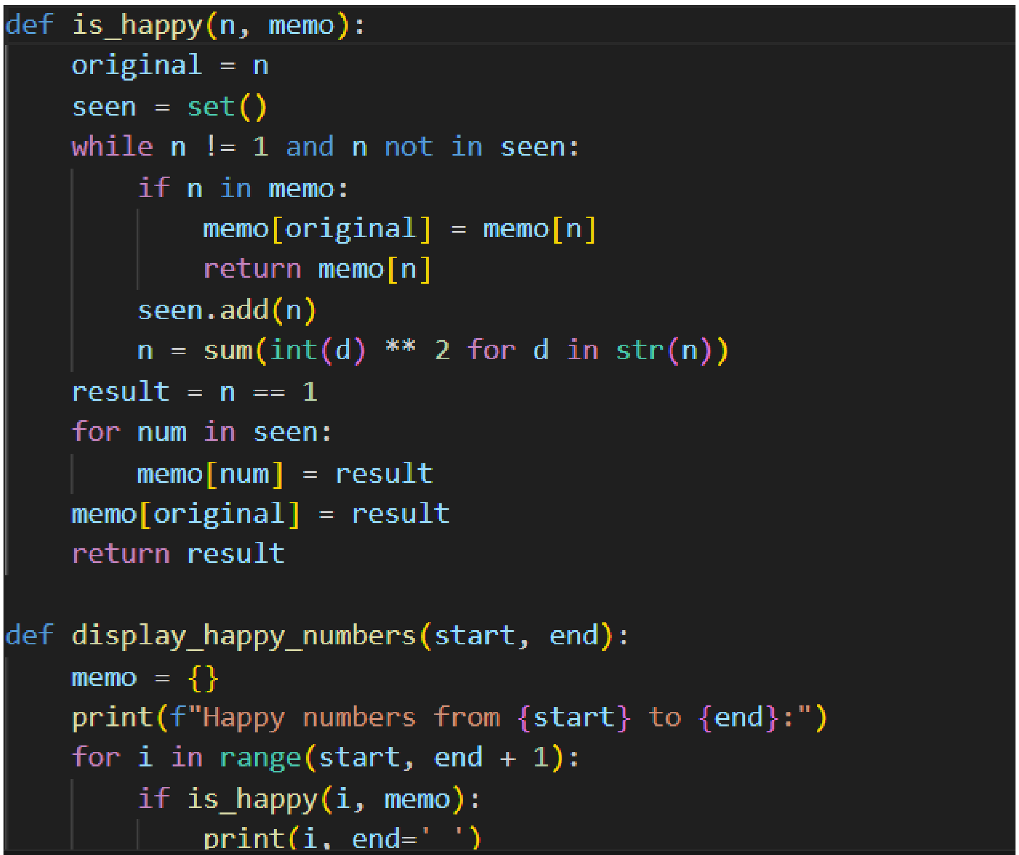
• Python program that prints all Happy Numbers within a range.

• Optimized version using cycle detection with explanation.

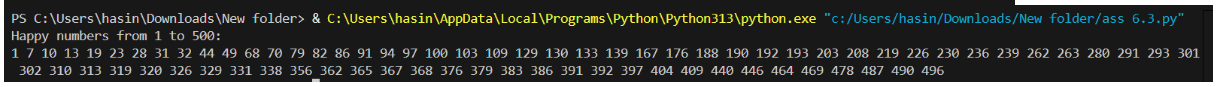


```python
def is_happy_number(n):
    """Check if a number is a happy number."""
    seen = set()
    while n != 1 and n not in seen:
        seen.add(n)
        n = sum(int(digit) ** 2 for digit in str(n))
    return n == 1


def display_happy_numbers(start, end):
    """Display all happy numbers within a specified range."""
    happy_numbers = [num for num in range(start, end + 1) if is_happy_number(num)]
    print(f"Happy Numbers from {start} to {end}:")
    print(happy_numbers)
    return happy_numbers


# Display all happy numbers from 1 to 500
display_happy_numbers(1, 500)
```

```
PS C:\Users\hasin\Downloads\New folder> & C:\Users\hasin\AppData\Local\Programs\Python\Python313\python.exe "c:/Users/hasin/Downloads/New folder/ass 6.3.py"
Happy Numbers from 1 to 500:
[1, 7, 10, 13, 19, 23, 28, 31, 32, 44, 49, 68, 70, 79, 82, 86, 91, 94, 97, 100, 103, 109, 129, 130, 133, 139, 167, 176, 188, 190, 192, 193, 203, 208, 219, 226
, 230, 236, 239, 262, 263, 280, 291, 293, 301, 302, 310, 313, 319, 320, 326, 329, 331, 338, 356, 362, 365, 367, 368, 376, 379, 383, 386, 391, 392, 397, 404, 4
09, 440, 446, 464, 469, 478, 487, 490, 496]
```



```python
def is_happy(n, memo):
    original = n
    seen = set()
    while n != 1 and n not in seen:
        if n in memo:
            memo[original] = memo[n]
            return memo[n]
        seen.add(n)
        n = sum(int(d) ** 2 for d in str(n))
    result = n == 1
    for num in seen:
        memo[num] = result
    memo[original] = result
    return result

def display_happy_numbers(start, end):
    memo = {}
    print(f"Happy numbers from {start} to {end}:")
    for i in range(start, end + 1):
        if is_happy(i, memo):
            print(i, end=' ')
```



```
PS C:\Users\hasin\Downloads\New folder> & C:\Users\hasin\AppData\Local\Programs\Python\Python313\python.exe "c:/Users/hasin/Downloads/New folder/ass 6.3.py"
Happy numbers from 1 to 500:
1 7 10 13 19 23 28 31 32 44 49 68 70 79 82 86 91 94 97 100 103 109 129 130 133 139 167 176 188 190 192 193 203 208 219 226 230 236 239 262 263 280 291 293 301
 302 310 313 319 320 326 329 331 338 356 362 365 367 368 376 379 383 386 391 392 397 404 409 440 446 464 469 478 487 490 496
```

1. It saves results of numbers already checked, so the same calculations are not done again and again.
2. This avoids repeated loops for the same digit sequences, which makes it much faster for large ranges.
3. Using a set still prevents infinite loops, but caching adds an extra speed boost.
4. The work per number is reduced because many numbers reuse previously computed results.
5. Overall, it runs faster with only a small amount of extra memory, making it more efficient and scalable.

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.

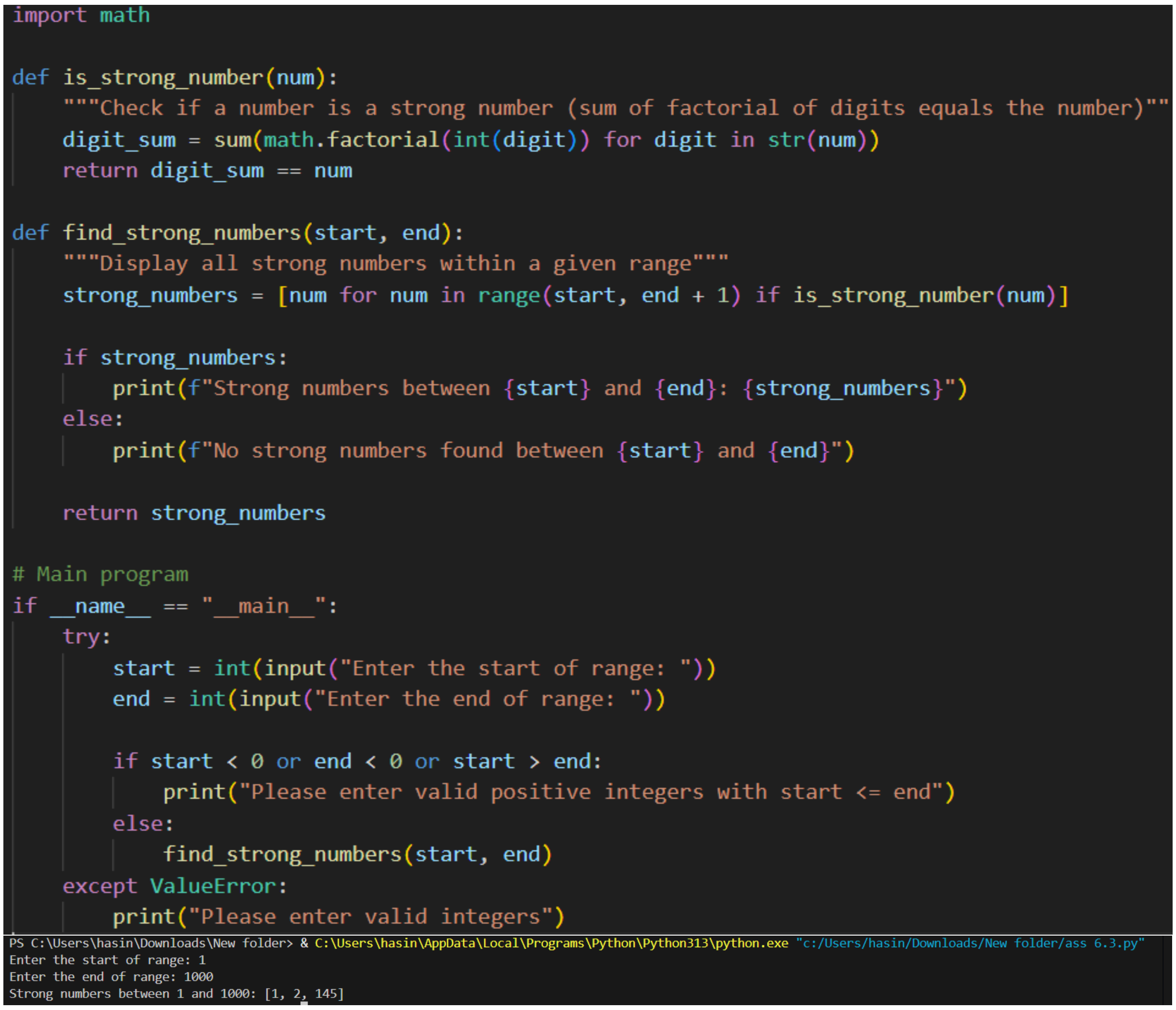
• Validate with examples (1, 2, 145).

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

Expected Output #9:

• Python program that lists Strong Numbers.

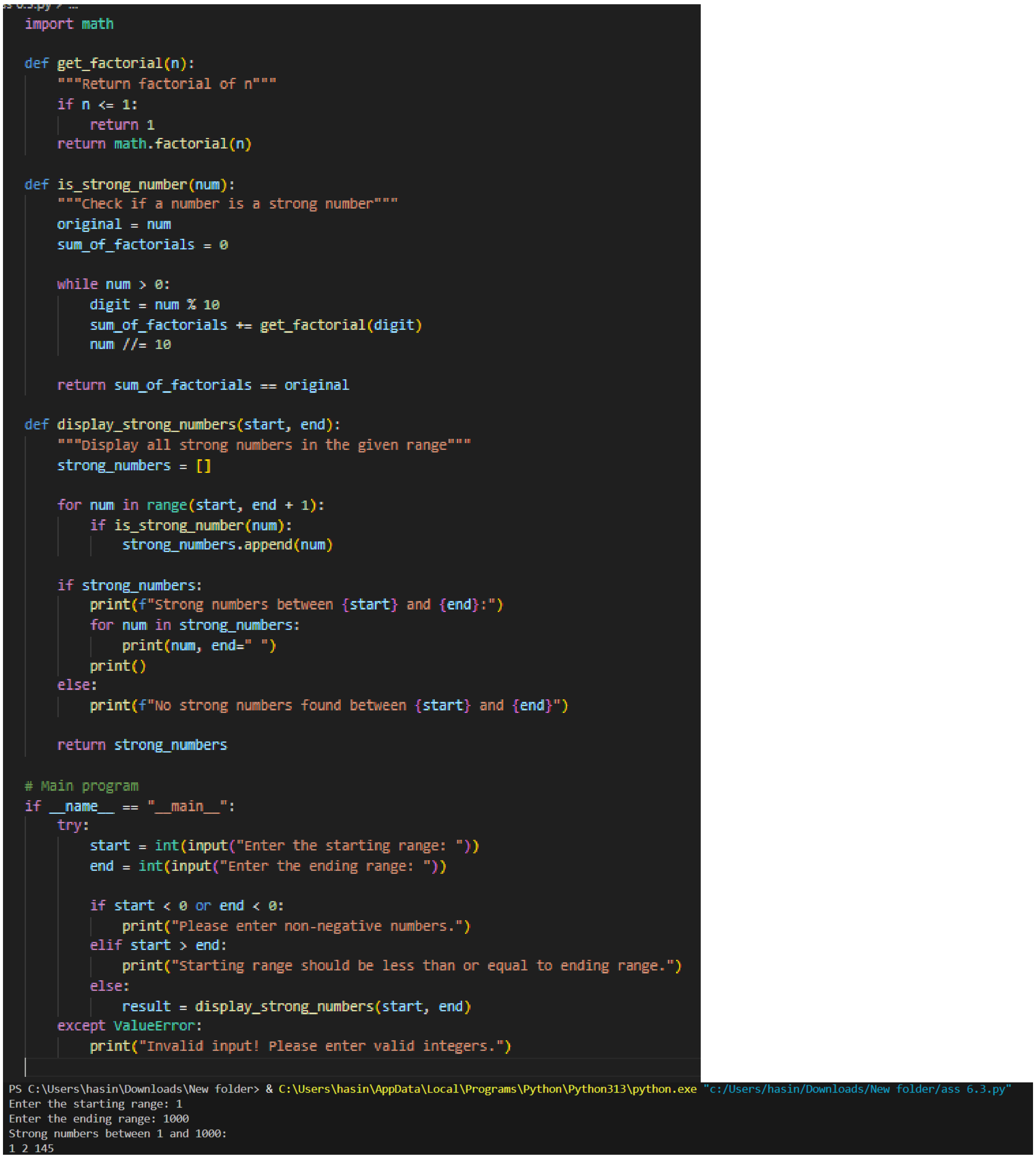
- Optimized version with explanation.



```python
import math

def is_strong_number(num):
    """Check if a number is a strong number (sum of factorial of digits equals the number)""
    digit_sum = sum(math.factorial(int(digit)) for digit in str(num))
    return digit_sum == num

def find_strong_numbers(start, end):
    """Display all strong numbers within a given range"""
    strong_numbers = [num for num in range(start, end + 1) if is_strong_number(num)]

    if strong_numbers:
        print(f"Strong numbers between {start} and {end}: {strong_numbers}")
    else:
        print(f"No strong numbers found between {start} and {end}")

    return strong_numbers

# Main program
if __name__ == "__main__":
    try:
        start = int(input("Enter the start of range: "))
        end = int(input("Enter the end of range: "))

        if start < 0 or end < 0 or start > end:
            print("Please enter valid positive integers with start <= end")
        else:
            find_strong_numbers(start, end)
    except ValueError:
        print("Please enter valid integers")
```

```
PS C:\Users\hasin\Downloads\New folder> & C:\Users\hasin\AppData\Local\Programs\Python\Python313\python.exe "c:/Users/hasin/Downloads/New folder/ass 6.3.py"
Enter the start of range: 1
Enter the end of range: 1000
Strong numbers between 1 and 1000: [1, 2, 145]
```

Optimized:



```python
import math

def get_factorial(n):
    """Return factorial of n"""
    if n <= 1:
        return 1
    return math.factorial(n)

def is_strong_number(num):
    """Check if a number is a strong number"""
    original = num
    sum_of_factorials = 0

    while num > 0:
        digit = num % 10
        sum_of_factorials += get_factorial(digit)
        num //= 10

    return sum_of_factorials == original

def display_strong_numbers(start, end):
    """Display all strong numbers in the given range"""
    strong_numbers = []

    for num in range(start, end + 1):
        if is_strong_number(num):
            strong_numbers.append(num)

    if strong_numbers:
        print(f"Strong numbers between {start} and {end}:")
        for num in strong_numbers:
            print(num, end=" ")
        print()
    else:
        print(f"No strong numbers found between {start} and {end}")

    return strong_numbers

# Main program
if __name__ == "__main__":
    try:
        start = int(input("Enter the starting range: "))
        end = int(input("Enter the ending range: "))

        if start < 0 or end < 0:
            print("Please enter non-negative numbers.")
        elif start > end:
            print("Starting range should be less than or equal to ending range.")
        else:
            result = display_strong_numbers(start, end)
    except ValueError:
        print("Invalid input! Please enter valid integers.")
```

```
PS C:\Users\hasin\Downloads\New folder> & C:\Users\hasin\AppData\Local\Programs\Python\Python313\python.exe "c:/Users/hasin/Downloads/New folder/ass 6.3.py"
Enter the starting range: 1
Enter the ending range: 1000
Strong numbers between 1 and 1000:
1 2 145
```

Time Complexity Comparison:

- **Original**: O(n × d × f) where n = range size, d = digits per number, f = factorial computation cost
- **Optimized**: O(10 × f) + O(n × d) = effectively O(n × d) since the precomputation happens only once

For larger ranges, this makes a significant performance difference since you're trading a tiny bit of extra memory (10 dictionary entries) for substantial computation savings across the entire loop.

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:

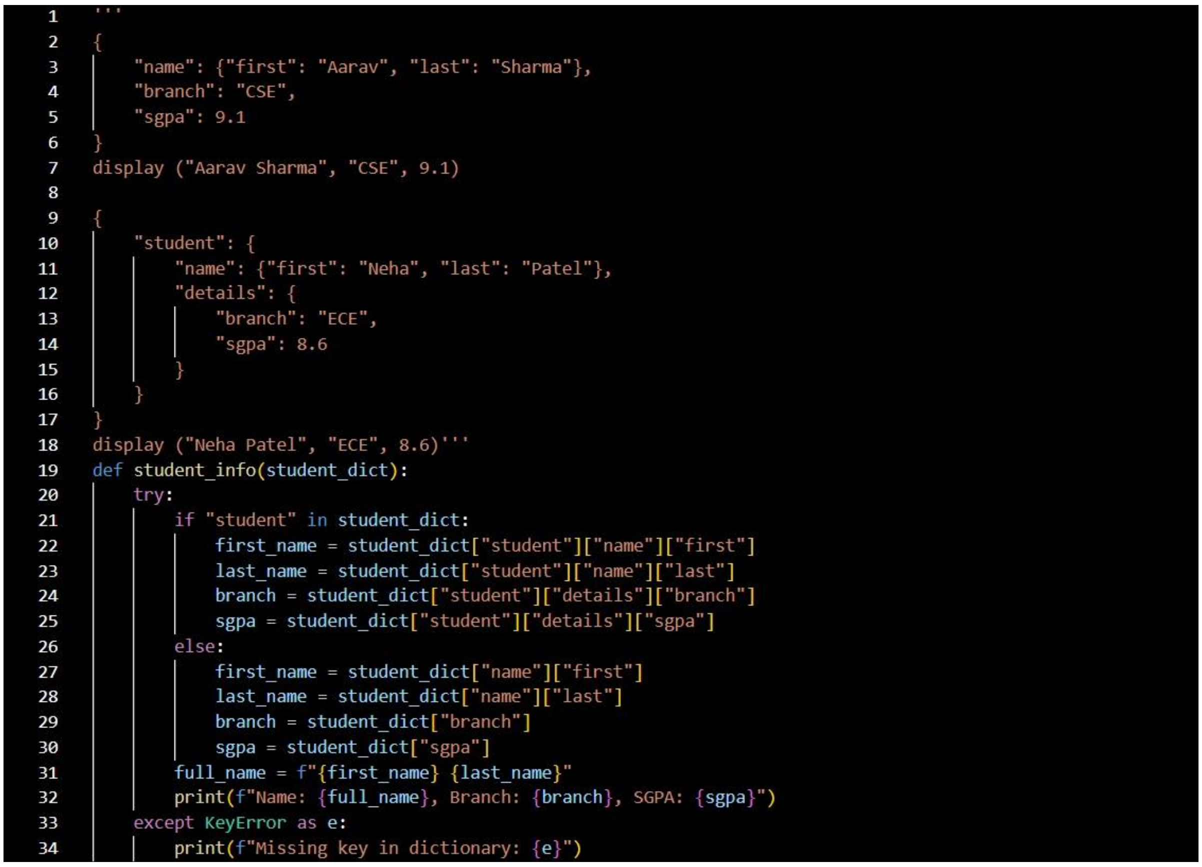
o Full Name

o Branch

o SGPA

Expected Output

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



```python
'''
{
    "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}")
```

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).

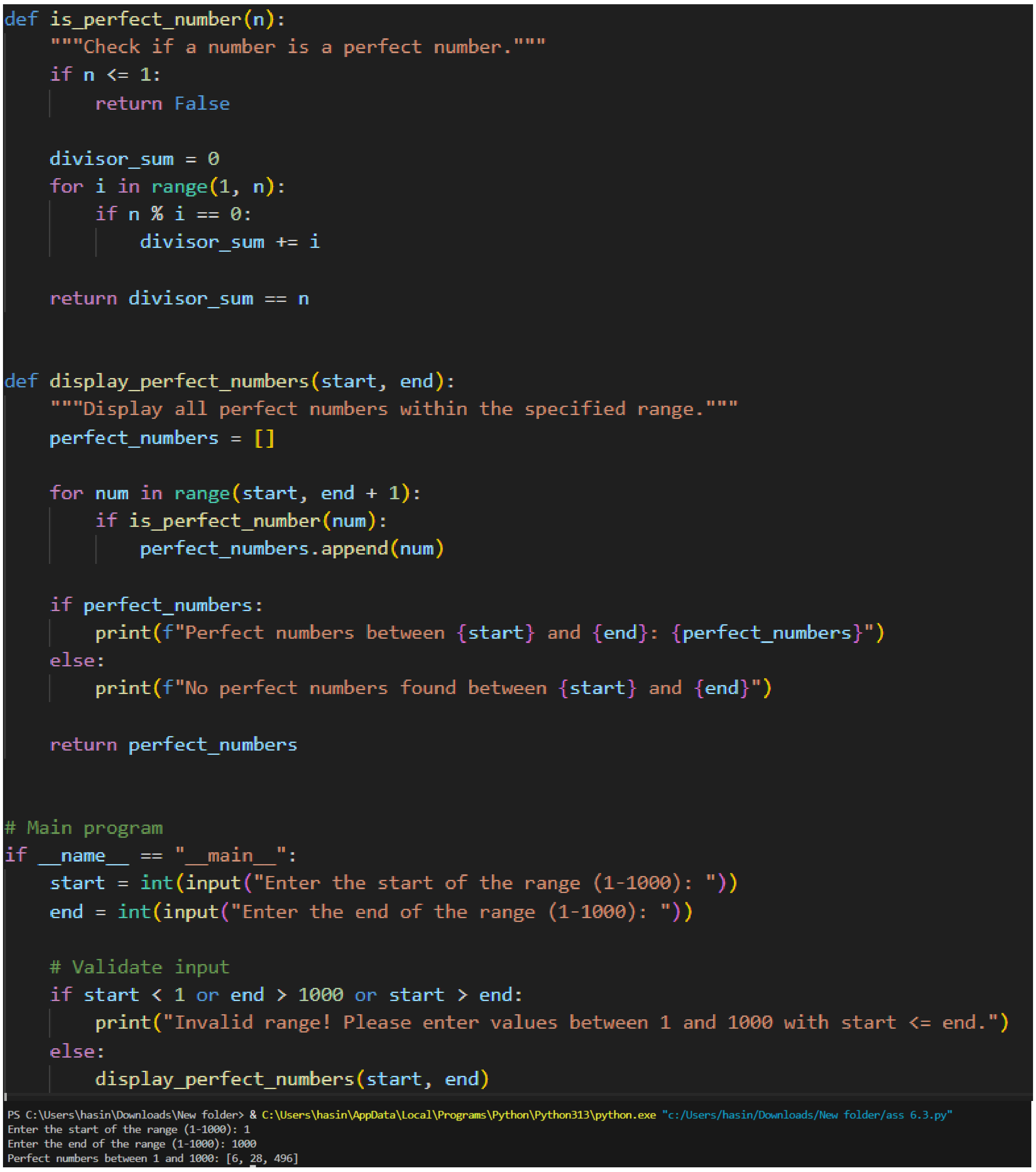
o Example: 6 = 1 + 2 + 3, 28 = 1 + 2 + 4 + 7 + 14.

• 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 root n

Optimized:



```python
def is_perfect_number(n):
    """Check if a number is a perfect number."""
    if n <= 1:
        return False

    divisor_sum = 0
    for i in range(1, n):
        if n % i == 0:
            divisor_sum += i

    return divisor_sum == n


def display_perfect_numbers(start, end):
    """Display all perfect numbers within the specified range."""
    perfect_numbers = []

    for num in range(start, end + 1):
        if is_perfect_number(num):
            perfect_numbers.append(num)

    if perfect_numbers:
        print(f"Perfect numbers between {start} and {end}: {perfect_numbers}")
    else:
        print(f"No perfect numbers found between {start} and {end}")

    return perfect_numbers


# Main program
if __name__ == "__main__":
    start = int(input("Enter the start of the range (1-1000): "))
    end = int(input("Enter the end of the range (1-1000): "))

    # Validate input
    if start < 1 or end > 1000 or start > end:
        print("Invalid range! Please enter values between 1 and 1000 with start <= end.")
    else:
        display_perfect_numbers(start, end)
```

```
PS C:\Users\hasin\Downloads\New folder> & C:\Users\hasin\AppData\Local\Programs\Python\Python313\python.exe "c:/Users/hasin/Downloads/New folder/ass 6.3.py"
Enter the start of the range (1-1000): 1
Enter the end of the range (1-1000): 1000
Perfect numbers between 1 and 1000: [6, 28, 496]
```

1. The optimized code checks only up to √n instead of all numbers from 1 to n, which greatly reduces the amount of work.
2. It uses the fact that divisors come in pairs, so when it finds one divisor, it automatically gets the other (n ÷ i) in the same step.
3. It avoids counting the same divisor twice by checking that i and n // i are different, which is important for perfect squares like 36.
4. The original method is very slow because it works in O(n × m), while the optimized one works in O(n × √m).
5. For numbers up to 10,000, the optimized version checks about 100 values instead of 10,000, making it roughly 100 times faster.