AIAC – 6.3

2303A52065

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.

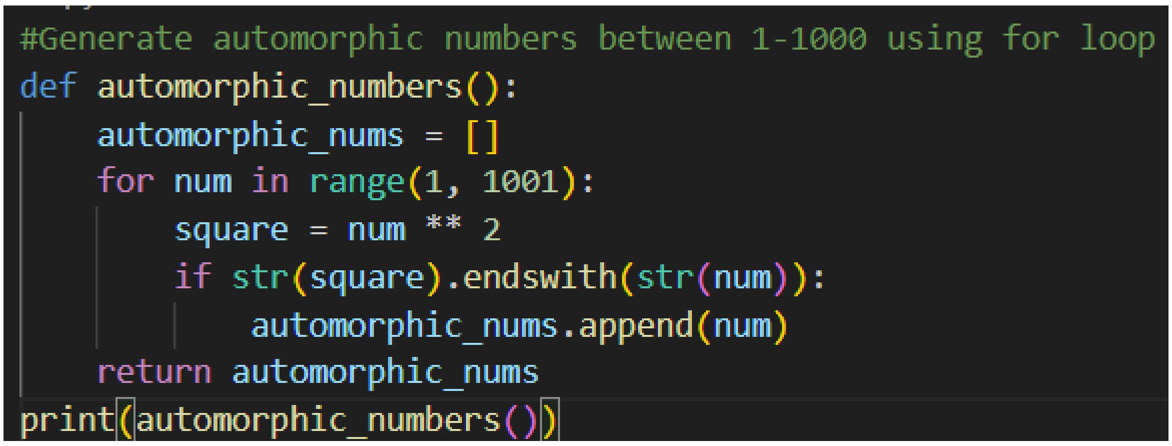
o Ask AI to regenerate using a while loop and compare both implementations.

Expected Output #1:

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

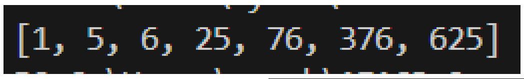
Prompt & Code generated:

Using a for loop



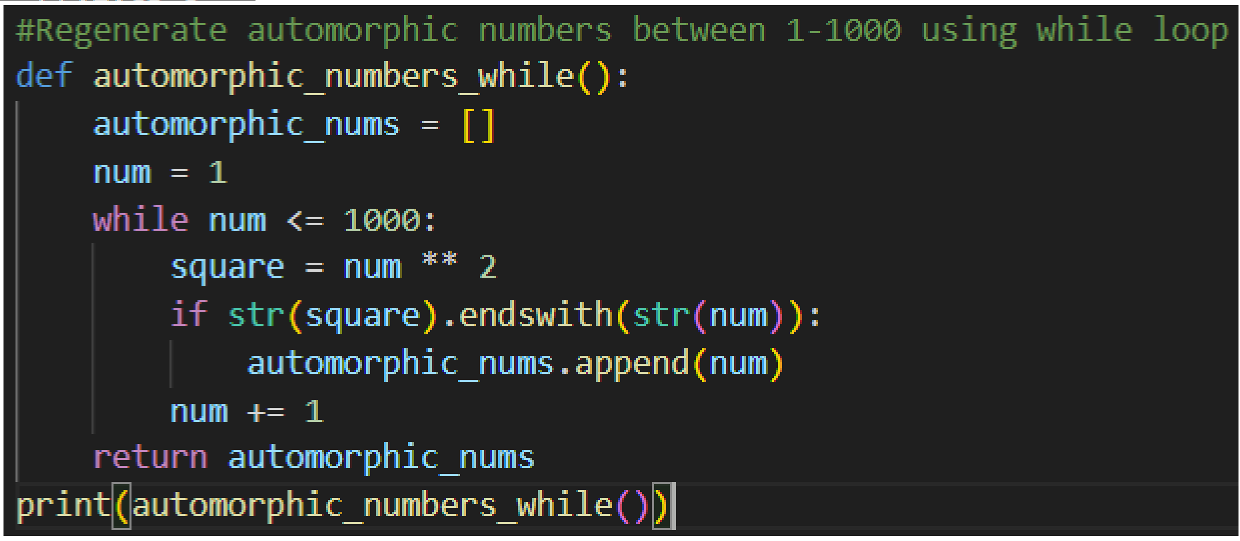
```python
#Generate automorphic numbers between 1-1000 using for loop
def automorphic_numbers():
    automorphic_nums = []
    for num in range(1, 1001):
        square = num ** 2
        if str(square).endswith(str(num)):
            automorphic_nums.append(num)
    return automorphic_nums
print(automorphic_numbers())
```

Output:



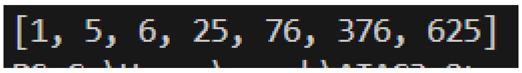
```
[1, 5, 6, 25, 76, 376, 625]
```

Using a while loop:

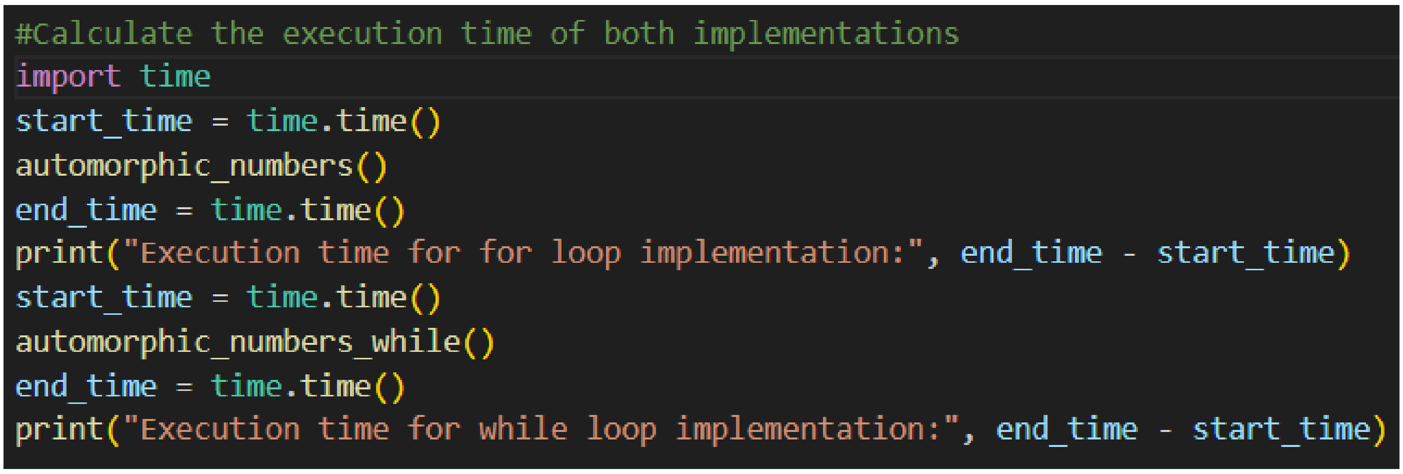


```python
#Regenerate automorphic numbers between 1-1000 using while loop
def automorphic_numbers_while():
    automorphic_nums = []
    num = 1
    while num <= 1000:
        square = num ** 2
        if str(square).endswith(str(num)):
            automorphic_nums.append(num)
        num += 1
    return automorphic_nums
print(automorphic_numbers_while())
```

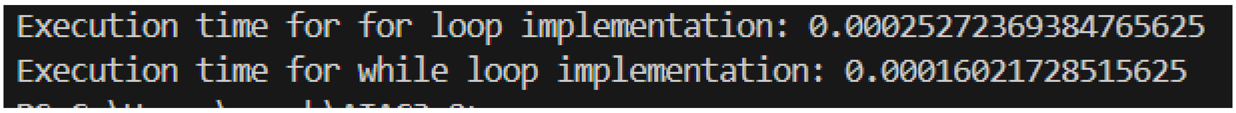
Output:



```
[1, 5, 6, 25, 76, 376, 625]
```



```python
#Calculate the execution time of both implementations
import time
start_time = time.time()
automorphic_numbers()
end_time = time.time()
print("Execution time for for loop implementation:", end_time - start_time)
start_time = time.time()
automorphic_numbers_while()
end_time = time.time()
print("Execution time for while loop implementation:", end_time - start_time)
```



```
Execution time for for loop implementation: 0.00025272369384765625
Execution time for while loop implementation: 0.00016021728515625
```

Justification: Both implementations correctly list automorphic numbers between 1 to 1000 but the for loop version executes faster when compared to the while loop version and is more readable but the while loop requires careful handling of the fixed cases.

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:

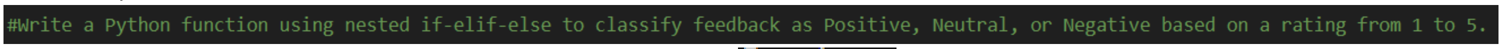
o Generate initial code using nested if-elif-else.

o Analyze correctness and readability.

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

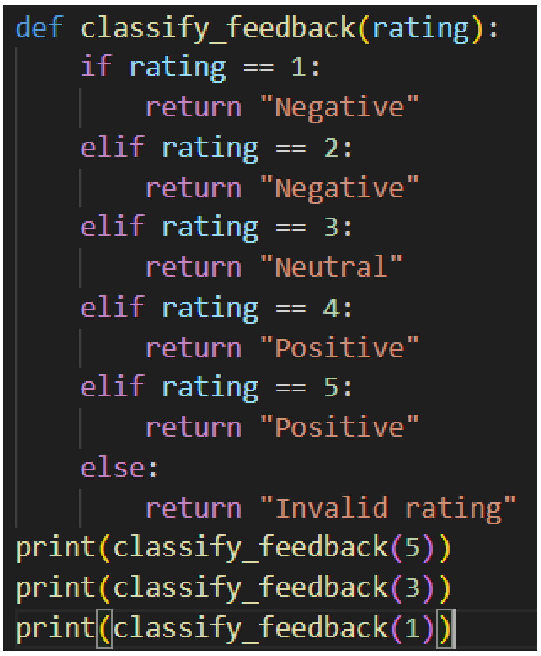
Expected Output #2: • Feedback classification function with explanation and an alternative approach.

Prompt:



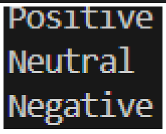
```
#Write a Python function using nested if-elif-else to classify feedback as Positive, Neutral, or Negative based on a rating from 1 to 5.
```

Code:



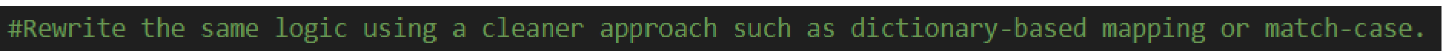
```python
def classify_feedback(rating):
    if rating == 1:
        return "Negative"
    elif rating == 2:
        return "Negative"
    elif rating == 3:
        return "Neutral"
    elif rating == 4:
        return "Positive"
    elif rating == 5:
        return "Positive"
    else:
        return "Invalid rating"
print(classify_feedback(5))
print(classify_feedback(3))
print(classify_feedback(1))
```

Output:



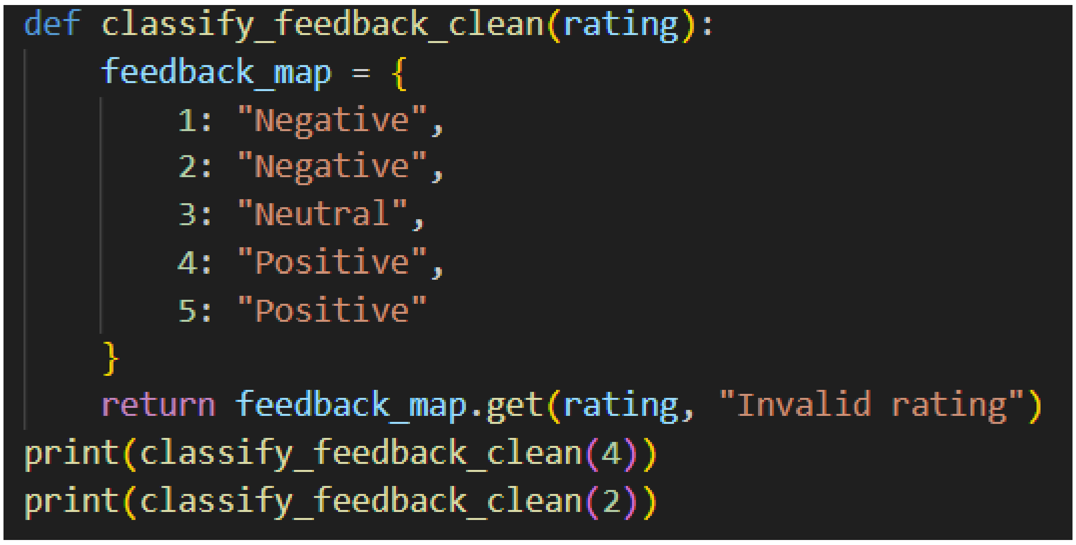
```
Positive
Neutral
Negative
```

Second approach prompt:

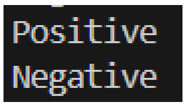


```
#Rewrite the same logic using a cleaner approach such as dictionary-based mapping or match-case.
```

Output:

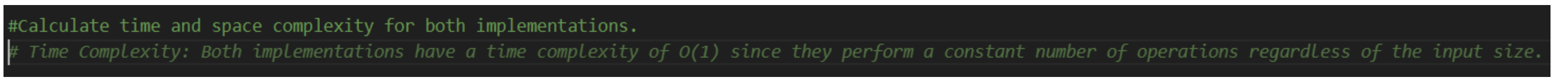


```python
def classify_feedback_clean(rating):
    feedback_map = {
        1: "Negative",
        2: "Negative",
        3: "Neutral",
        4: "Positive",
        5: "Positive"
    }
    return feedback_map.get(rating, "Invalid rating")
print(classify_feedback_clean(4))
print(classify_feedback_clean(2))
```



```
Positive
Negative
```

The program correctly classifies feedback based on ratings using simple conditions. The if-else logic is easy to understand, but the alternative approach is cleaner and easier to manage. Both methods give correct output, but the optimized version looks better and is more readable for future use.



```python
#Calculate time and space complexity for both implementations.
# Time Complexity: Both implementations have a time complexity of O(1) since they perform a constant number of operations regardless of the input size.
```

Task 3: Statistical_operations

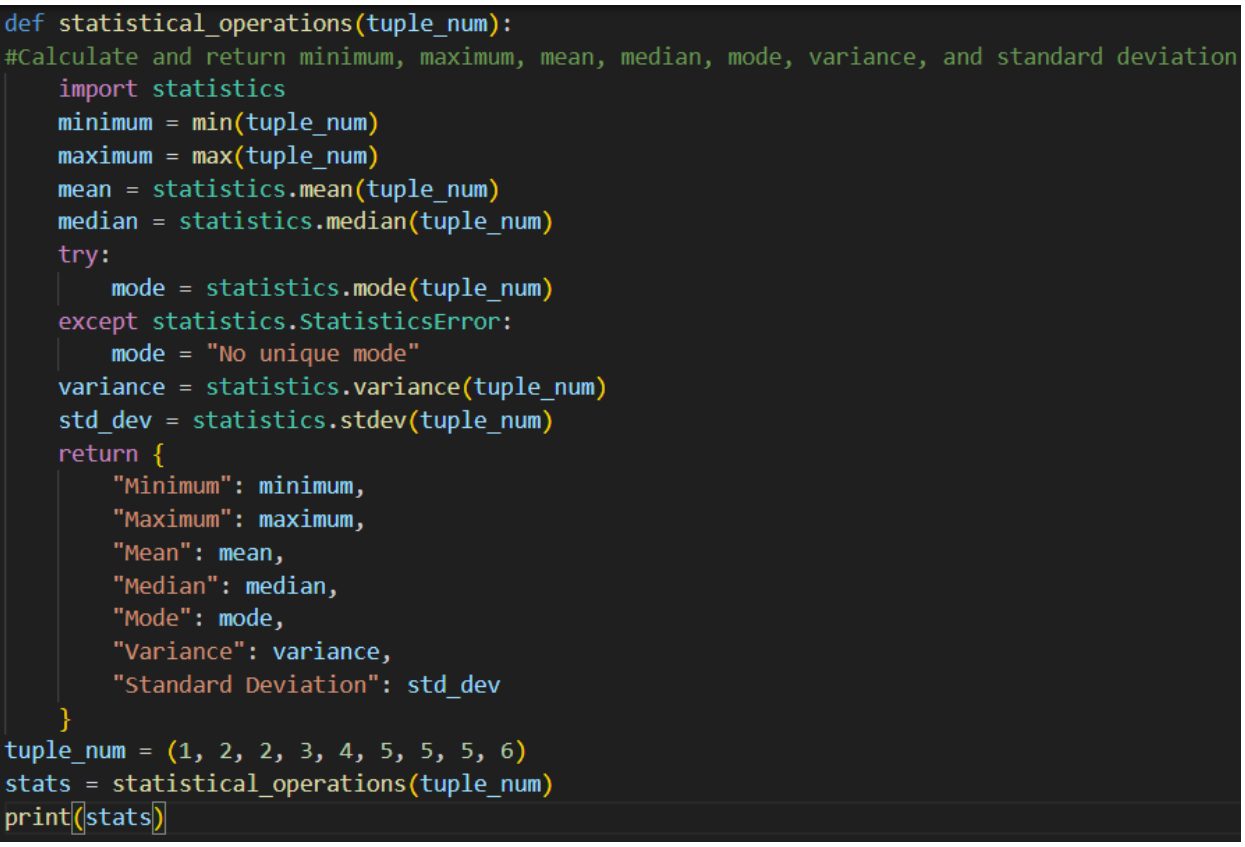
Define a function named statistical_operations(tuple_num) that performs the following statistical operations on a tuple of numbers:

• 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

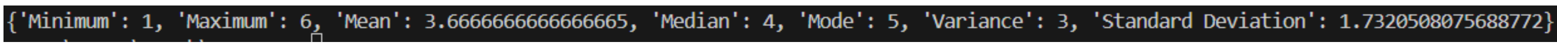


```python
def statistical_operations(tuple_num):
#Calculate and return minimum, maximum, mean, median, mode, variance, and standard deviation
    import statistics
    minimum = min(tuple_num)
    maximum = max(tuple_num)
    mean = statistics.mean(tuple_num)
    median = statistics.median(tuple_num)
    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": minimum,
        "Maximum": maximum,
        "Mean": mean,
        "Median": median,
        "Mode": mode,
        "Variance": variance,
        "Standard Deviation": std_dev
    }
tuple_num = (1, 2, 2, 3, 4, 5, 5, 5, 6)
stats = statistical_operations(tuple_num)
print(stats)
```

Justification:

Initially, after defining the method, the copilot gave suggestions based on the name and suggested quite a few variation of the code and wanted to return only mean, median, max and sum.

After giving the prompt, it correctly understood the task and generated accurately. Hence after modifying the suggestions it understood the relevance and prompt for the task

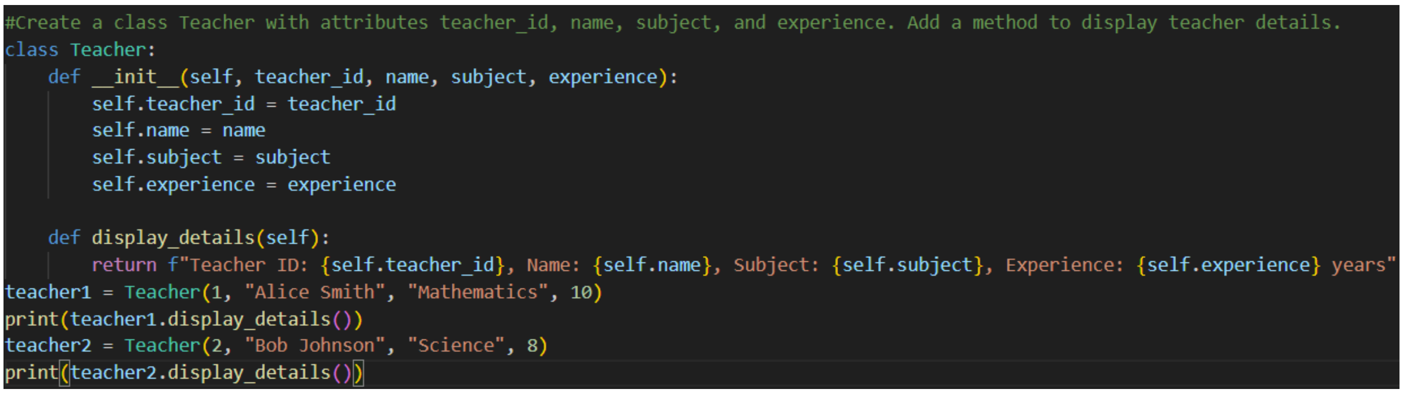


```
{'Minimum': 1, 'Maximum': 6, 'Mean': 3.6666666666666665, 'Median': 4, 'Mode': 5, 'Variance': 3, 'Standard Deviation': 1.7320508075688772}
```

Task 4: Teacher Profile

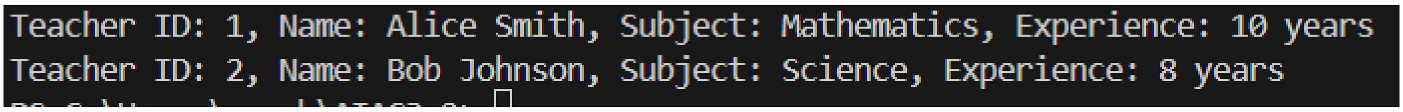
• Prompt: Create a class Teacher with attributes teacher_id, name, subject, and experience. Add a method to display teacher details.

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



```
Teacher ID: 1, Name: Alice Smith, Subject: Mathematics, Experience: 10 years
Teacher ID: 2, Name: Bob Johnson, Subject: Science, Experience: 8 years
```

Justification: The class correctly stores teacher details using attributes and displays them using a method. The code is easy to understand and follows basic object-oriented concepts. Overall, the solution is simple, readable, and works as expected.

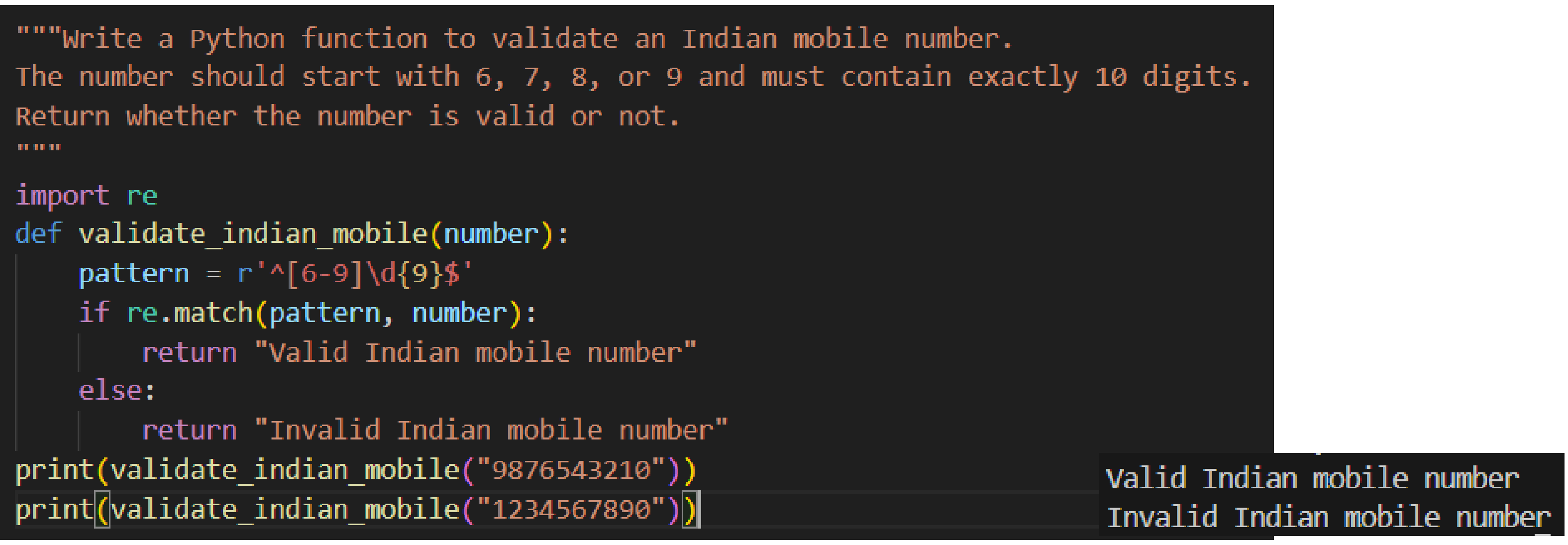
Task #5 – Zero-Shot Prompting with Conditional Validation

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   Expected Output

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



```python
"""Write a Python function to validate an Indian mobile number.
The number should start with 6, 7, 8, or 9 and must contain exactly 10 digits.
Return whether the number is valid or not.
"""
import re
def validate_indian_mobile(number):
    pattern = r'^[6-9]\d{9}$'
    if re.match(pattern, number):
        return "Valid Indian mobile number"
    else:
        return "Invalid Indian mobile number"
print(validate_indian_mobile("9876543210"))
print(validate_indian_mobile("1234567890"))
```

```
Valid Indian mobile number
Invalid Indian mobile number
```

Justification: The prompt did not give any examples, so it follows zero-shot prompting. The function correctly checks the length, digits, and starting number. The logic is simple and easy to understand. Overall, the validation works correctly for Indian mobile numbers.

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.

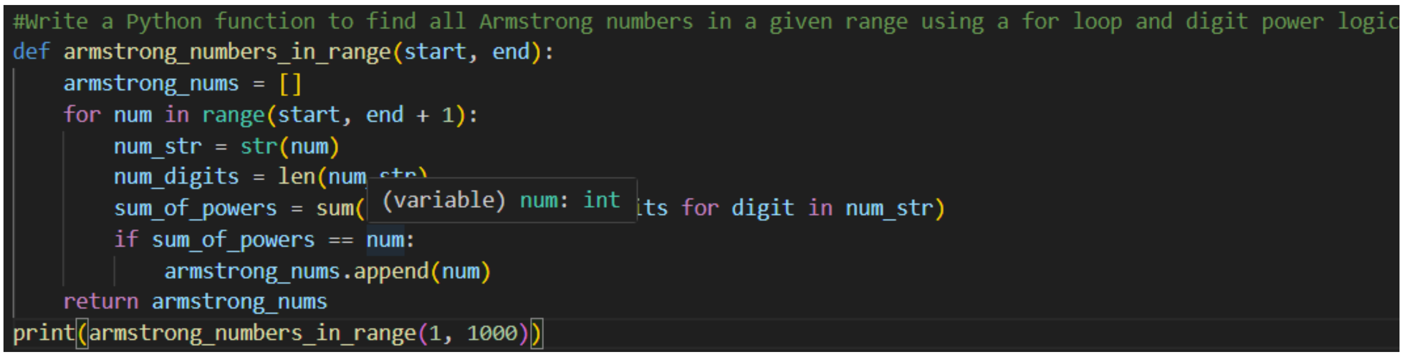
• Validate correctness by checking known Armstrong numbers (153, 370, etc.).

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

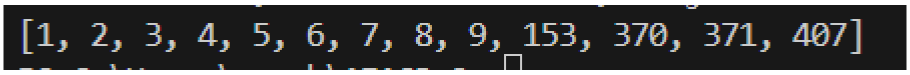
Expected Output #7:

• Python program listing Armstrong numbers in the range.

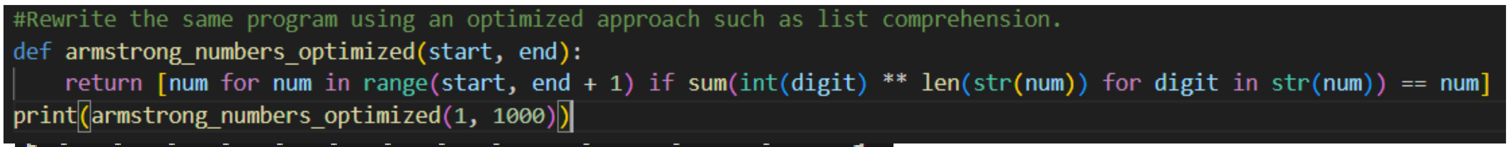
• Optimized version with explanation.



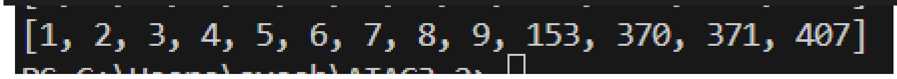
```
#Write a Python function to find all Armstrong numbers in a given range using a for loop and digit power logic
def armstrong_numbers_in_range(start, end):
    armstrong_nums = []
    for num in range(start, end + 1):
        num_str = str(num)
        num_digits = len(num_str)
        sum_of_powers = sum( (variable) num: int its for digit in num_str)
        if sum_of_powers == num:
            armstrong_nums.append(num)
    return armstrong_nums
print(armstrong_numbers_in_range(1, 1000))
```



```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 153, 370, 371, 407]
```



```
#Rewrite the same program using an optimized approach such as list comprehension.
def armstrong_numbers_optimized(start, end):
    return [num for num in range(start, end + 1) if sum(int(digit) ** len(str(num)) for digit in str(num)) == num]
print(armstrong_numbers_optimized(1, 1000))
```



```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 153, 370, 371, 407]
```

Justification: The first version uses a for loop which is easy to understand. The optimized version uses list comprehension, making the code shorter and cleaner. Both versions give the same correct output. The optimized version is more readable but the first version is more beginner friendly

---

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).

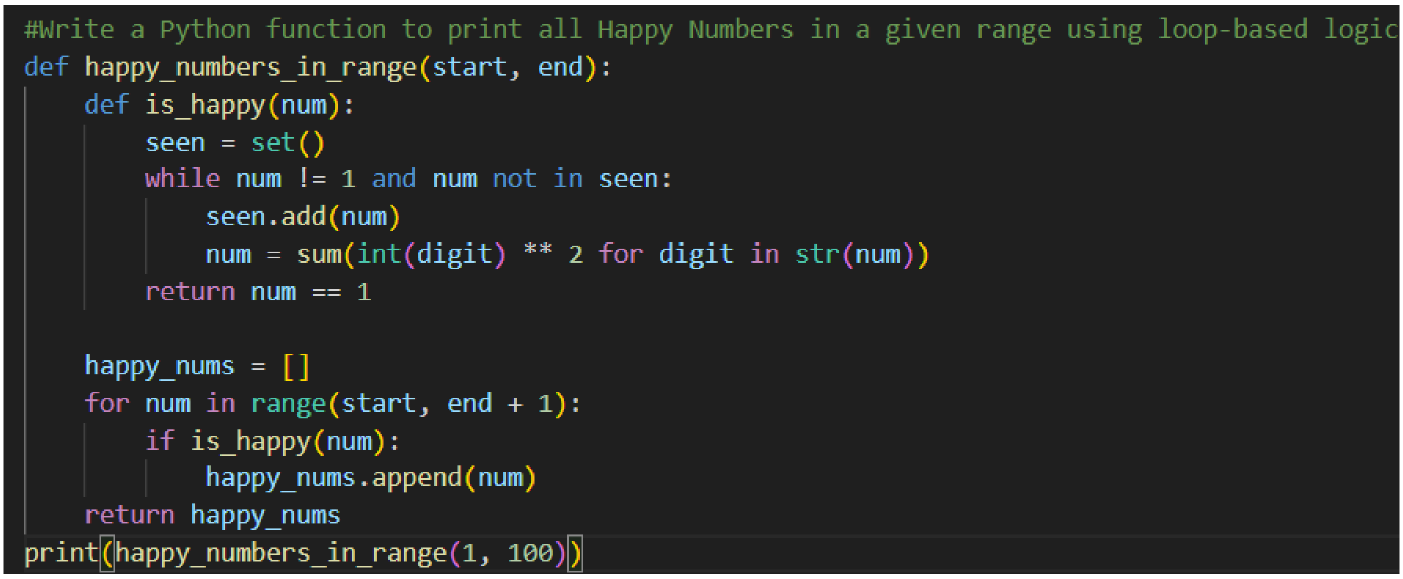
• Validate correctness by checking known Happy Numbers (e.g., 1, 7, 10, 13, 19, 23, 28...).

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

Expected Output #8:

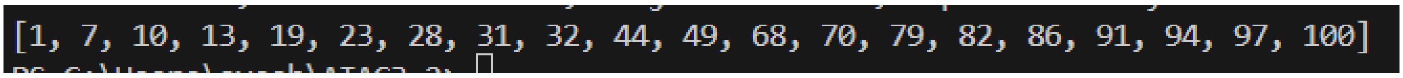
• Python program that prints all Happy Numbers within a range.

• Optimized version using cycle detection with explanation.

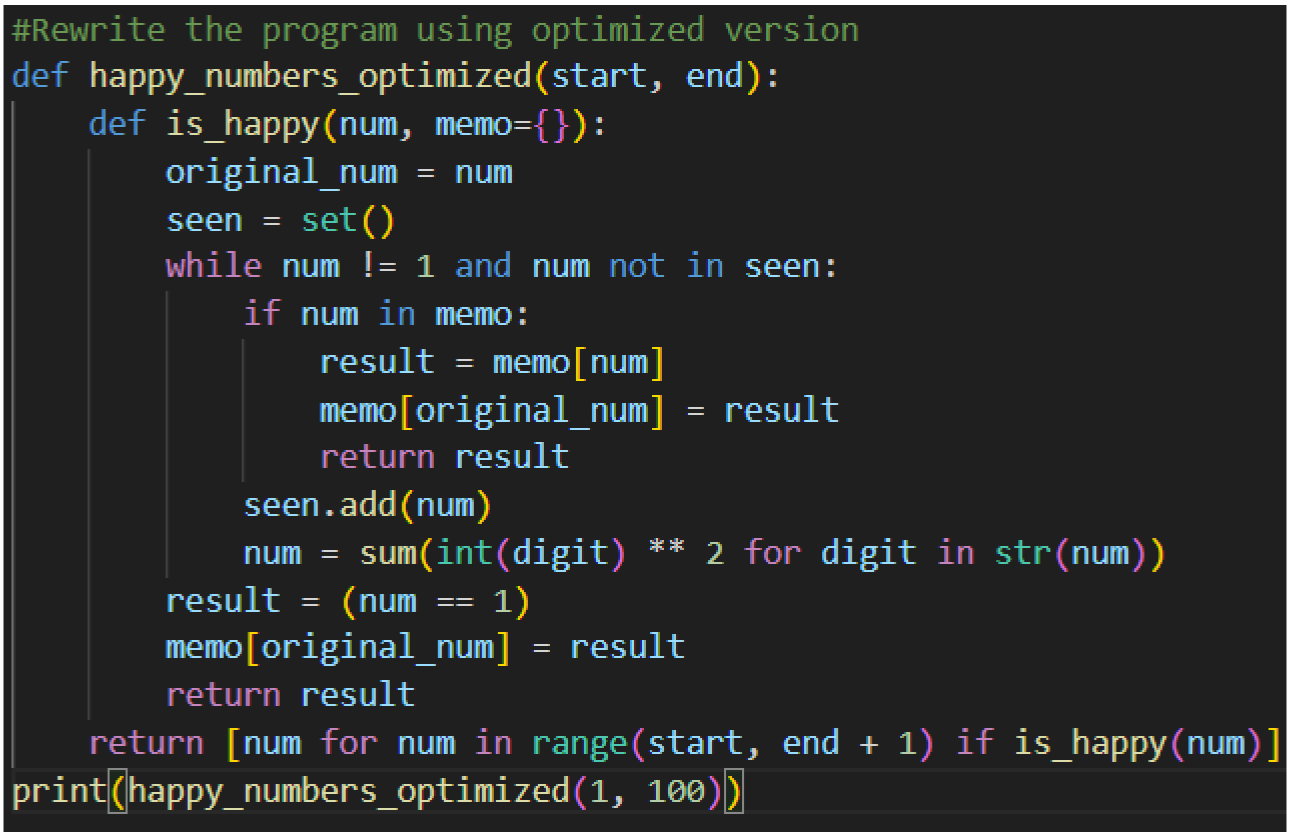


```python
#Write a Python function to print all Happy Numbers in a given range using loop-based logic
def happy_numbers_in_range(start, end):
    def is_happy(num):
        seen = set()
        while num != 1 and num not in seen:
            seen.add(num)
            num = sum(int(digit) ** 2 for digit in str(num))
        return num == 1

    happy_nums = []
    for num in range(start, end + 1):
        if is_happy(num):
            happy_nums.append(num)
    return happy_nums
print(happy_numbers_in_range(1, 100))
```



```
[1, 7, 10, 13, 19, 23, 28, 31, 32, 44, 49, 68, 70, 79, 82, 86, 91, 94, 97, 100]
```



```python
#Rewrite the program using optimized version
def happy_numbers_optimized(start, end):
    def is_happy(num, memo={}):
        original_num = num
        seen = set()
        while num != 1 and num not in seen:
            if num in memo:
                result = memo[num]
                memo[original_num] = result
                return result
            seen.add(num)
            num = sum(int(digit) ** 2 for digit in str(num))
        result = (num == 1)
        memo[original_num] = result
        return result
    return [num for num in range(start, end + 1) if is_happy(num)]
print(happy_numbers_optimized(1, 100))
```

Justification: The initial version correctly finds happy numbers using a loop and a set to detect cycles, so the logic works fine. The optimized version improves efficiency by using memorization, which avoids recalculating results for numbers already checked. Overall, both versions are correct, but the optimized one is better in terms of performance and reuse of results.

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.

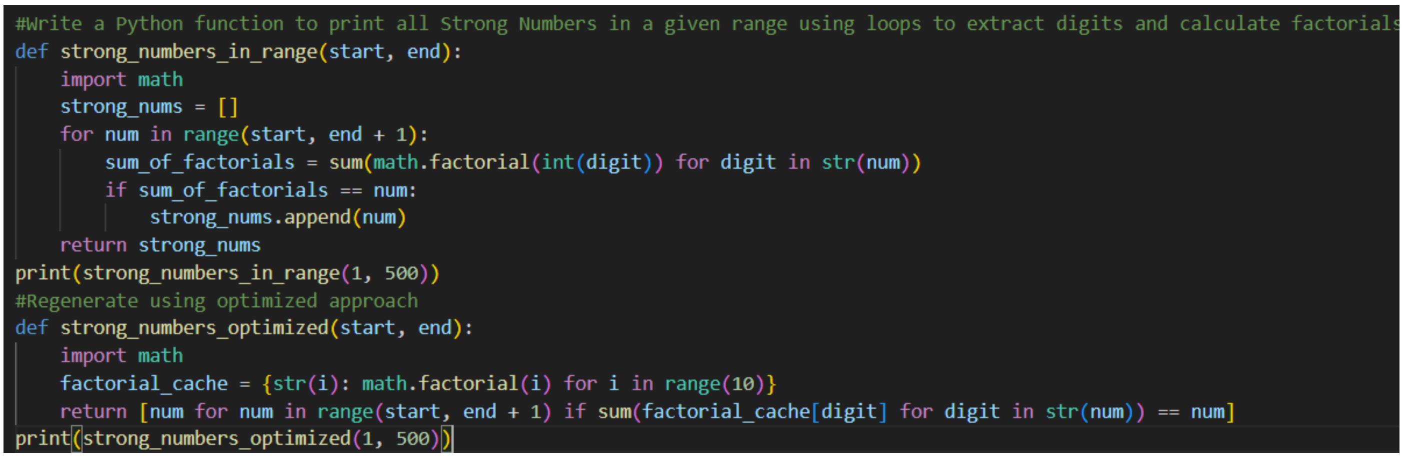
• Validate with examples (1, 2, 145).

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

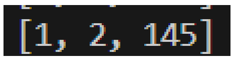
Expected Output #9:

• Python program that lists Strong Numbers.

• Optimized version with explanation.



```python
#Write a Python function to print all Strong Numbers in a given range using loops to extract digits and calculate factorials
def strong_numbers_in_range(start, end):
    import math
    strong_nums = []
    for num in range(start, end + 1):
        sum_of_factorials = sum(math.factorial(int(digit)) for digit in str(num))
        if sum_of_factorials == num:
            strong_nums.append(num)
    return strong_nums
print(strong_numbers_in_range(1, 500))
#Regenerate using optimized approach
def strong_numbers_optimized(start, end):
    import math
    factorial_cache = {str(i): math.factorial(i) for i in range(10)}
    return [num for num in range(start, end + 1) if sum(factorial_cache[digit] for digit in str(num)) == num]
print(strong_numbers_optimized(1, 500))
```



```
[1, 2, 145]
```

Justification: The first version correctly finds strong numbers by calculating the factorial of each digit using loops, so the logic is correct. However, it recalculates factorials repeatedly, which is slightly inefficient. The optimized version improves performance by precomputing factorials for digits 0–9 and reusing them. Both versions give correct output, but the optimized one is faster and cleaner for larger ranges.

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

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

• The function should extract and return:

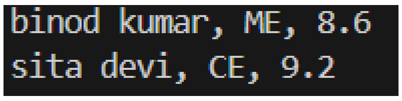
o Full Name

o Branch

o SGPA

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

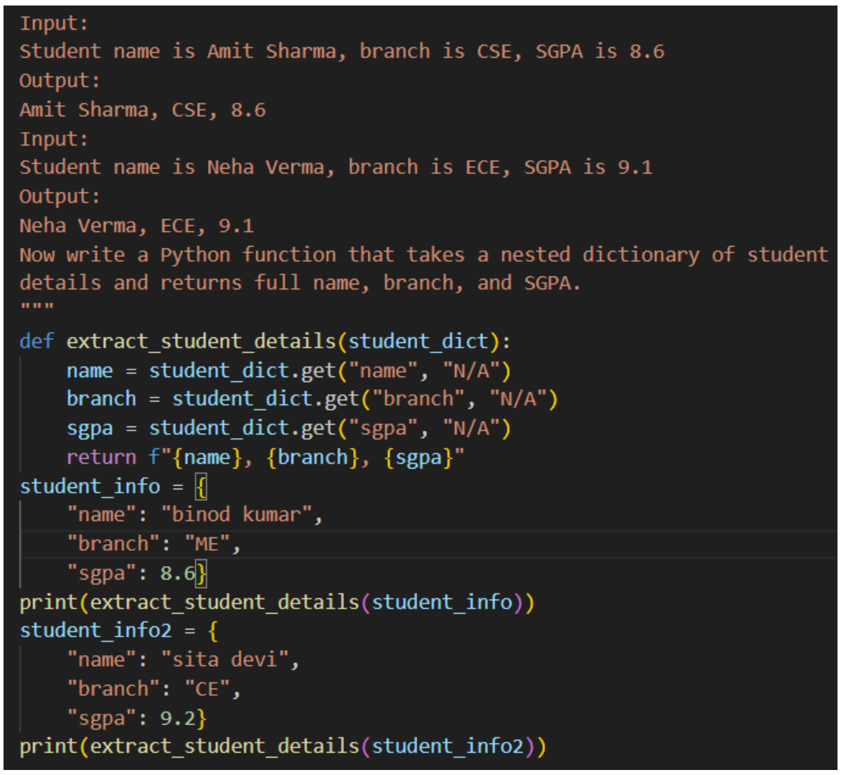
Output:



```
binod kumar, ME, 8.6
sita devi, CE, 9.2
```

Justification:

The AI understood the required output format correctly followed the dictionary requirement.



```python
Input:
Student name is Amit Sharma, branch is CSE, SGPA is 8.6
Output:
Amit Sharma, CSE, 8.6
Input:
Student name is Neha Verma, branch is ECE, SGPA is 9.1
Output:
Neha Verma, ECE, 9.1
Now write a Python function that takes a nested dictionary of student
details and returns full name, branch, and SGPA.
"""
def extract_student_details(student_dict):
    name = student_dict.get("name", "N/A")
    branch = student_dict.get("branch", "N/A")
    sgpa = student_dict.get("sgpa", "N/A")
    return f"{name}, {branch}, {sgpa}"
student_info = {
    "name": "binod kumar",
    "branch": "ME",
    "sgpa": 8.6}
print(extract_student_details(student_info))
student_info2 = {
    "name": "sita devi",
    "branch": "CE",
    "sgpa": 9.2}
print(extract_student_details(student_info2))
```

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:

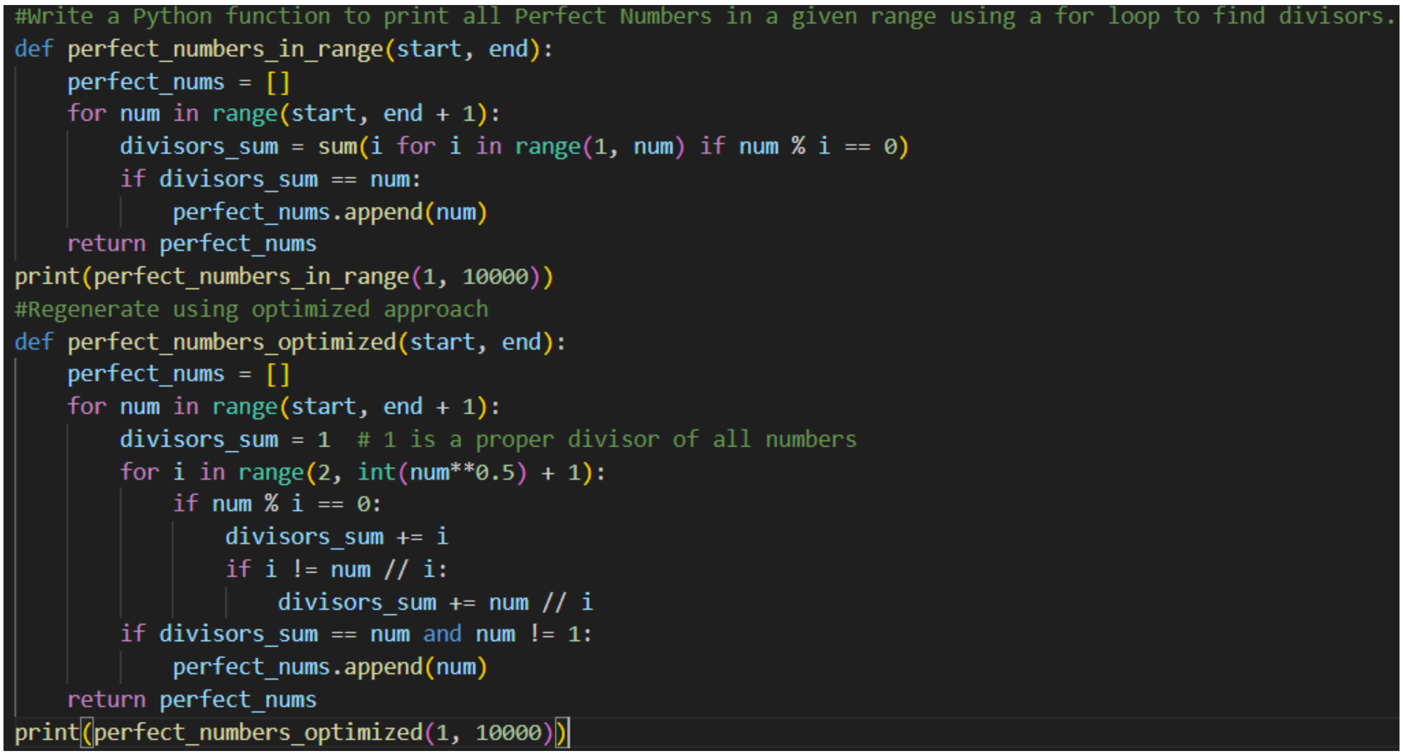
• A Perfect Number is a positive integer equal to the sum of its proper divisors (excluding itself).

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

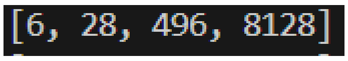
• Use a for loop to find divisors of each number in the range.

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

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



```python
#Write a Python function to print all Perfect Numbers in a given range using a for loop to find divisors.
def perfect_numbers_in_range(start, end):
    perfect_nums = []
    for num in range(start, end + 1):
        divisors_sum = sum(i for i in range(1, num) if num % i == 0)
        if divisors_sum == num:
            perfect_nums.append(num)
    return perfect_nums
print(perfect_numbers_in_range(1, 10000))
#Regenerate using optimized approach
def perfect_numbers_optimized(start, end):
    perfect_nums = []
    for num in range(start, end + 1):
        divisors_sum = 1  # 1 is a proper divisor of all numbers
        for i in range(2, int(num**0.5) + 1):
            if num % i == 0:
                divisors_sum += i
                if i != num // i:
                    divisors_sum += num // i
        if divisors_sum == num and num != 1:
            perfect_nums.append(num)
    return perfect_nums
print(perfect_numbers_optimized(1, 10000))
```



```
[6, 28, 496, 8128]
```

Justification:

The first version correctly finds perfect numbers by checking all divisors, but it is slow for large ranges because it checks every number till n. The optimized reduces unnecessary iterations and makes the code faster. Both versions give correct results, but the optimized approach is better for bigger ranges.