AI ASSISTED CODING

Assignment- 6.3

Name: M. Hasini

HT. No: 2303A51109

Batch:02

#1 Task Description (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:

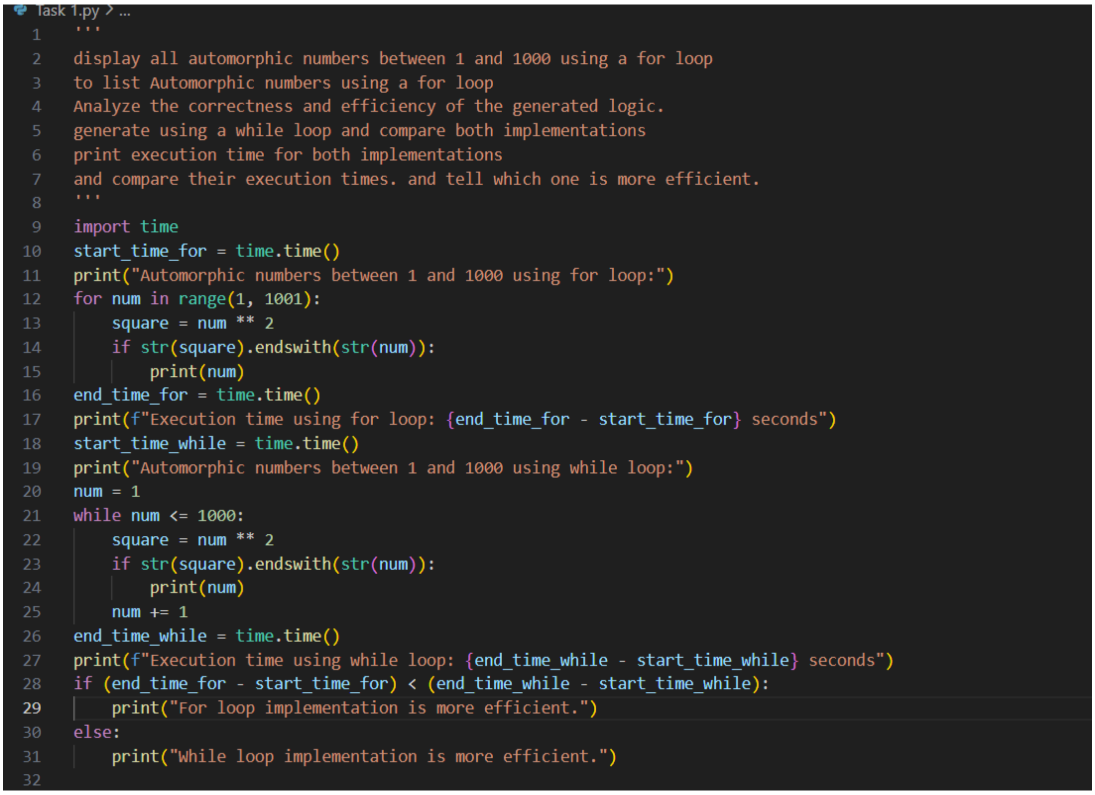
o Get AI-generated code to list Automorphic numbers using a for loop.

o Analyze the correctness and efficiency of the generated logic.

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

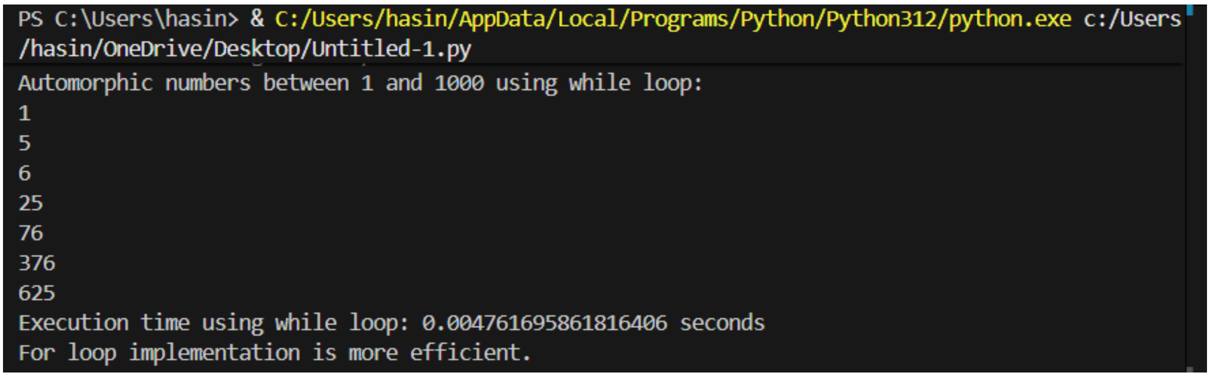
Expected Output #1:

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



```python
'''
display all automorphic numbers between 1 and 1000 using a for loop
to list Automorphic numbers using a for loop
Analyze the correctness and efficiency of the generated logic.
generate using a while loop and compare both implementations
print execution time for both implementations
and compare their execution times. and tell which one is more efficient.
'''
import time
start_time_for = time.time()
print("Automorphic numbers between 1 and 1000 using for loop:")
for num in range(1, 1001):
    square = num ** 2
    if str(square).endswith(str(num)):
        print(num)
end_time_for = time.time()
print(f"Execution time using for loop: {end_time_for - start_time_for} seconds")
start_time_while = time.time()
print("Automorphic numbers between 1 and 1000 using while loop:")
num = 1
while num <= 1000:
    square = num ** 2
    if str(square).endswith(str(num)):
        print(num)
    num += 1
end_time_while = time.time()
print(f"Execution time using while loop: {end_time_while - start_time_while} seconds")
if (end_time_for - start_time_for) < (end_time_while - start_time_while):
    print("For loop implementation is more efficient.")
else:
    print("While loop implementation is more efficient.")
```

Output:



```
PS C:\Users\hasin> & C:/Users/hasin/AppData/Local/Programs/Python/Python312/python.exe c:/Users
/hasin/OneDrive/Desktop/Untitled-1.py
Automorphic numbers between 1 and 1000 using while loop:
1
5
6
25
76
376
625
Execution time using while loop: 0.004761695861816406 seconds
For loop implementation is more efficient.
```

#2 Task Description (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:

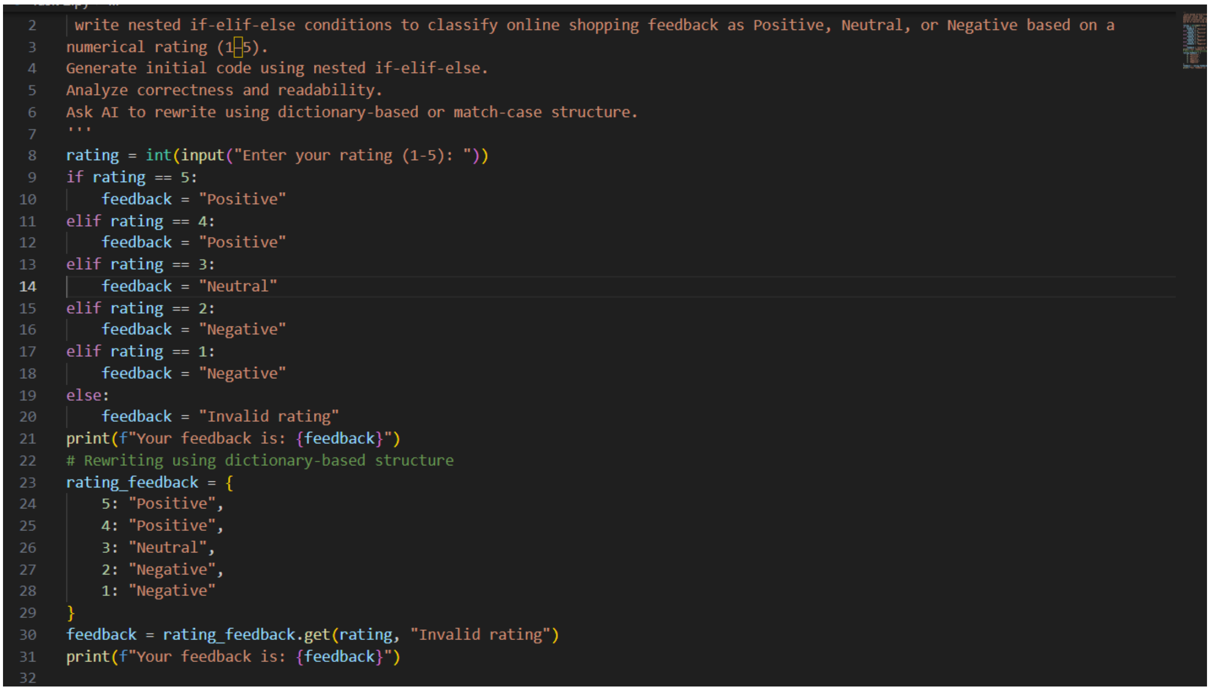
o Generate initial code using nested if-elif-else.

o Analyze correctness and readability.

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

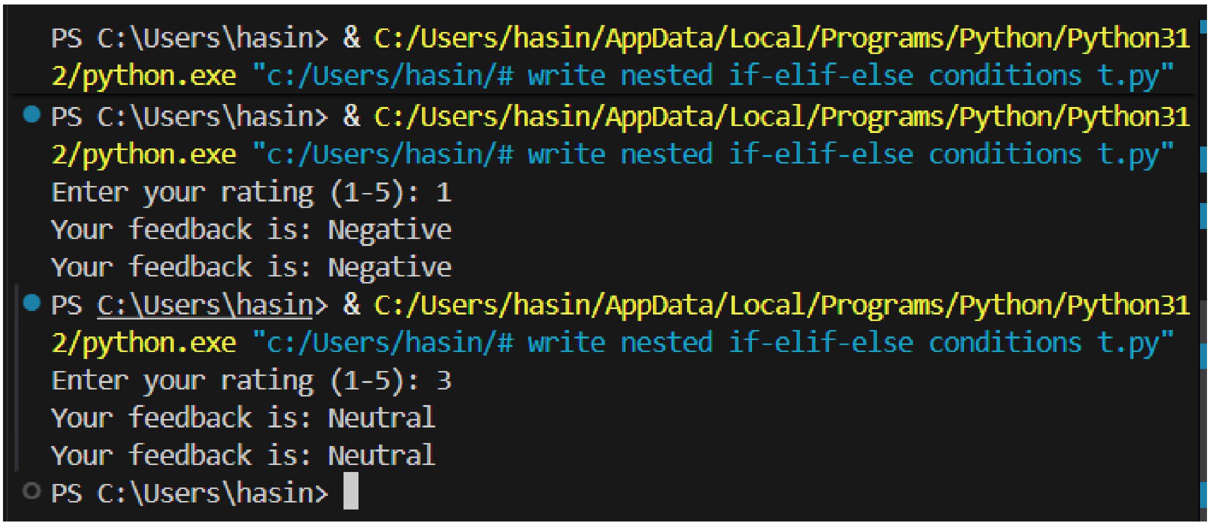
Expected Output #2:

• Feedback classification function with explanation and an alternative approach.



```
write nested if-elif-else conditions to classify online shopping feedback as Positive, Neutral, or Negative based on a
numerical rating (1□5).
Generate initial code using nested if-elif-else.
Analyze correctness and readability.
Ask AI to rewrite using dictionary-based or match-case structure.
'''
rating = int(input("Enter your rating (1-5): "))
if rating == 5:
    feedback = "Positive"
elif rating == 4:
    feedback = "Positive"
elif rating == 3:
    feedback = "Neutral"
elif rating == 2:
    feedback = "Negative"
elif rating == 1:
    feedback = "Negative"
else:
    feedback = "Invalid rating"
print(f"Your feedback is: {feedback}")
# Rewriting using dictionary-based structure
rating_feedback = {
    5: "Positive",
    4: "Positive",
    3: "Neutral",
    2: "Negative",
    1: "Negative"
}
feedback = rating_feedback.get(rating, "Invalid rating")
print(f"Your feedback is: {feedback}")
```

OUTPUT:



```
PS C:\Users\hasin> & C:/Users/hasin/AppData/Local/Programs/Python/Python31
2/python.exe "c:/Users/hasin/# write nested if-elif-else conditions t.py"
PS C:\Users\hasin> & C:/Users/hasin/AppData/Local/Programs/Python/Python31
2/python.exe "c:/Users/hasin/# write nested if-elif-else conditions t.py"
Enter your rating (1-5): 1
Your feedback is: Negative
Your feedback is: Negative
PS C:\Users\hasin> & C:/Users/hasin/AppData/Local/Programs/Python/Python31
2/python.exe "c:/Users/hasin/# write nested if-elif-else conditions t.py"
Enter your rating (1-5): 3
Your feedback is: Neutral
Your feedback is: Neutral
PS C:\Users\hasin>
```

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

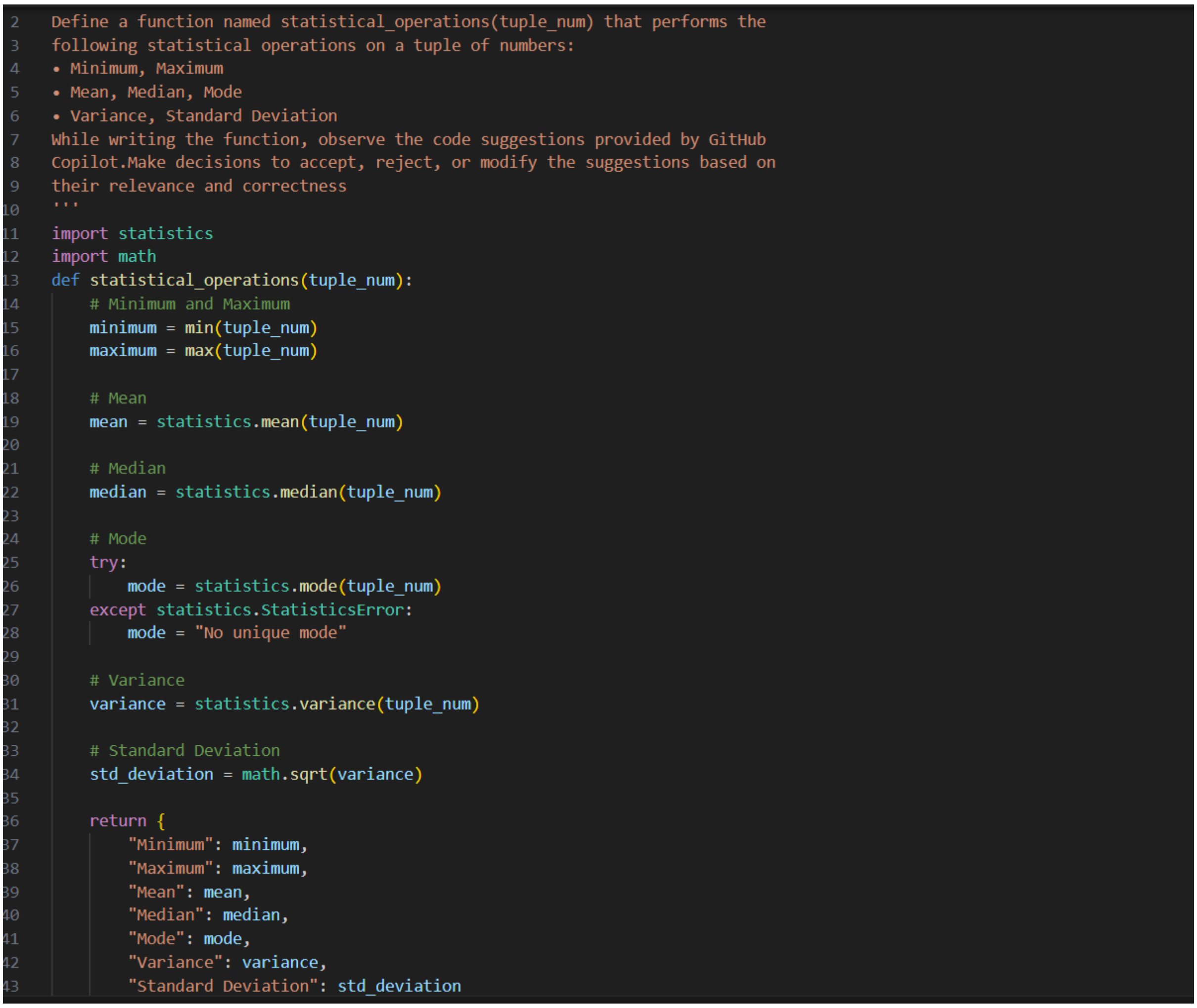
• Mean, Median, Mode

• Variance, Standard Deviation

While writing the function, observe the code suggestions provided by GitHub

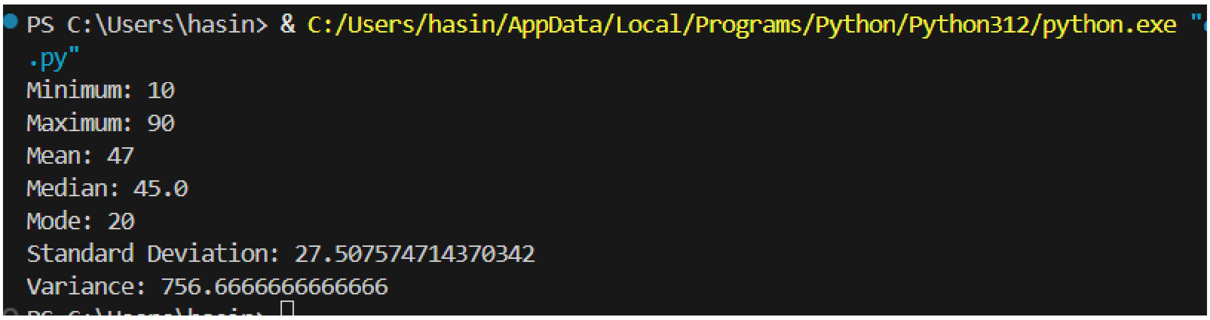
Copilot. Make decisions to accept, reject, or modify the suggestions based on

their relevance and correctness.



```
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
'''
import statistics
import math
def statistical_operations(tuple_num):
    # Minimum and Maximum
    minimum = min(tuple_num)
    maximum = max(tuple_num)

    # Mean
    mean = statistics.mean(tuple_num)

    # Median
    median = statistics.median(tuple_num)

    # Mode
    try:
        mode = statistics.mode(tuple_num)
    except statistics.StatisticsError:
        mode = "No unique mode"

    # Variance
    variance = statistics.variance(tuple_num)

    # Standard Deviation
    std_deviation = math.sqrt(variance)

    return {
        "Minimum": minimum,
        "Maximum": maximum,
        "Mean": mean,
        "Median": median,
        "Mode": mode,
        "Variance": variance,
        "Standard Deviation": std_deviation
```

Output:

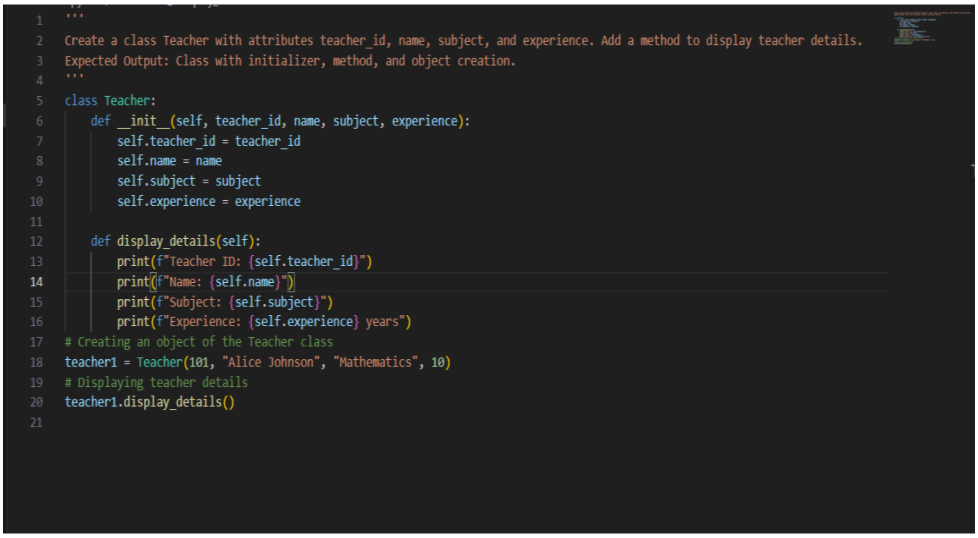


```
PS C:\Users\hasin> & C:/Users/hasin/AppData/Local/Programs/Python/Python312/python.exe "
.py"
Minimum: 10
Maximum: 90
Mean: 47
Median: 45.0
Mode: 20
Standard Deviation: 27.507574714370342
Variance: 756.6666666666666
PS C:\Users\hasin>
```

Task 4: Teacher Profile

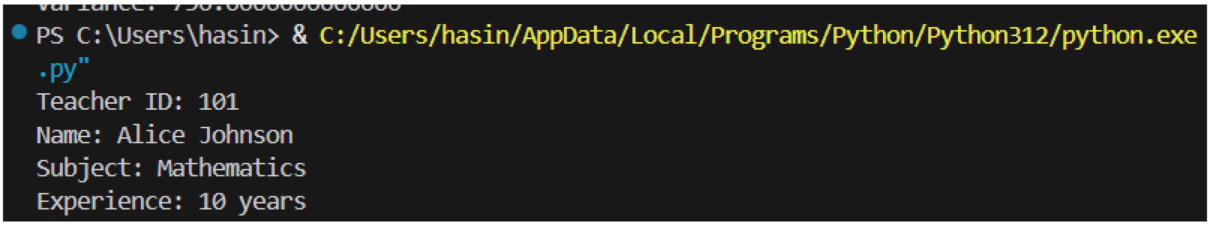
• Prompt: Create a class Teacher with attributes teacher_id, name, subject, and experience. Add a method to display teacher details.

• 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.
Expected Output: Class with initializer, method, and object creation.
'''
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")
# Creating an object of the Teacher class
teacher1 = Teacher(101, "Alice Johnson", "Mathematics", 10)
# Displaying teacher details
teacher1.display_details()
```

Output:



```
Variance: 756.6666666666666
PS C:\Users\hasin> & C:/Users/hasin/AppData/Local/Programs/Python/Python312/python.exe
.py"
Teacher ID: 101
Name: Alice Johnson
Subject: Mathematics
Experience: 10 years
```

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:

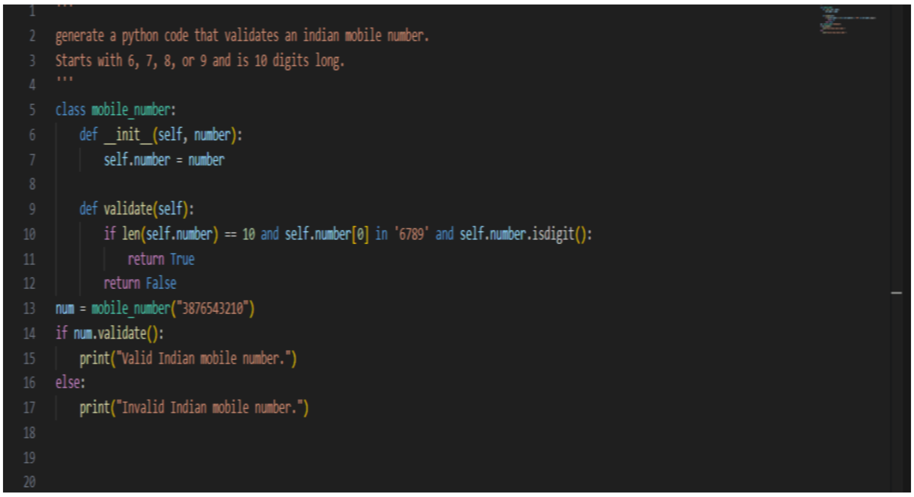
o Starts with 6, 7, 8, or 9

o Contains exactly 10 digits

Expected Output

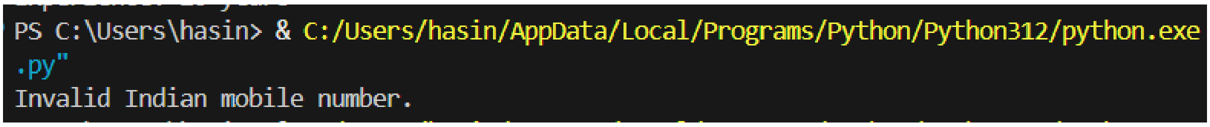
- A valid Python function that performs all required validations

without using any input-output examples in the prompt.



```
'''
generate a python code that validates an indian mobile number.
Starts with 6, 7, 8, or 9 and is 10 digits long.
'''
class mobile_number:
    def _init_(self, number):
        self.number = number

    def validate(self):
        if len(self.number) == 10 and self.number[0] in '6789' and self.number.isdigit():
            return True
        return False
num = mobile_number("3876543210")
if num.validate():
    print("Valid Indian mobile number.")
else:
    print("Invalid Indian mobile number.")
```

Output:



```
PS C:\Users\hasin> & C:/Users/hasin/AppData/Local/Programs/Python/Python312/python.exe
.py"
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:

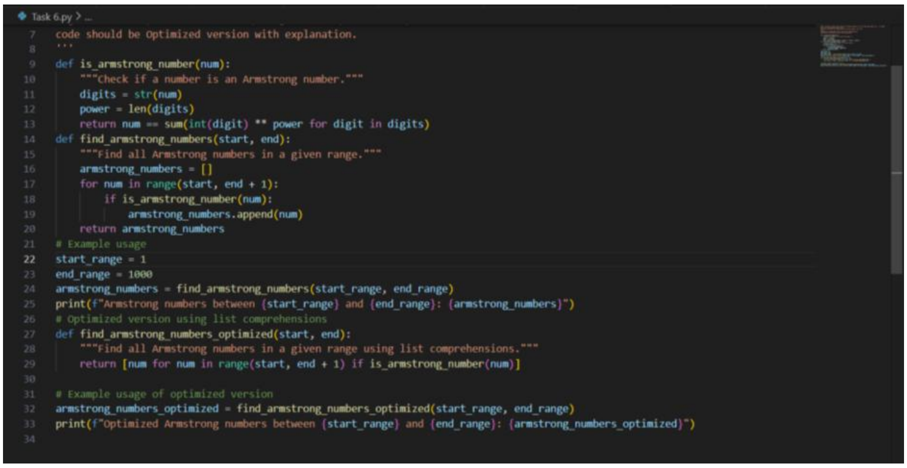
- Use a for loop and digit power logic.
- Validate correctness by checking known Armstrong numbers (153, 370,

etc.).

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

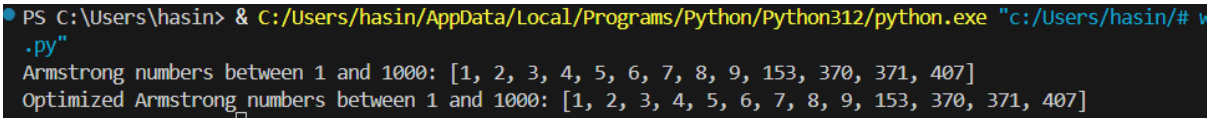
Expected Output #7:

- Python program listing Armstrong numbers in the range.
- Optimized version with explanation.



```python
code should be Optimized version with explanation.
'''
def is_armstrong_number(num):
    """Check if a number is an Armstrong number."""
    digits = str(num)
    power = len(digits)
    return num == sum(int(digit) ** power for digit in digits)
def find_armstrong_numbers(start, end):
    """Find all Armstrong numbers in a given range."""
    armstrong_numbers = []
    for num in range(start, end + 1):
        if is_armstrong_number(num):
            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}")
# Optimized version using list comprehensions
def find_armstrong_numbers_optimized(start, end):
    """Find all Armstrong numbers in a given range using list comprehensions."""
    return [num for num in range(start, end + 1) if is_armstrong_number(num)]

# Example usage of optimized version
armstrong_numbers_optimized = find_armstrong_numbers_optimized(start_range, end_range)
print(f"Optimized Armstrong numbers between {start_range} and {end_range}: {armstrong_numbers_optimized}")

```

Output:



```
PS C:\Users\hasin> & C:/Users/hasin/AppData/Local/Programs/Python/Python312/python.exe "c:/Users/hasin/# w
.py"
Armstrong numbers between 1 and 1000: [1, 2, 3, 4, 5, 6, 7, 8, 9, 153, 370, 371, 407]
Optimized Armstrong numbers between 1 and 1000: [1, 2, 3, 4, 5, 6, 7, 8, 9, 153, 370, 371, 407]
```

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…).

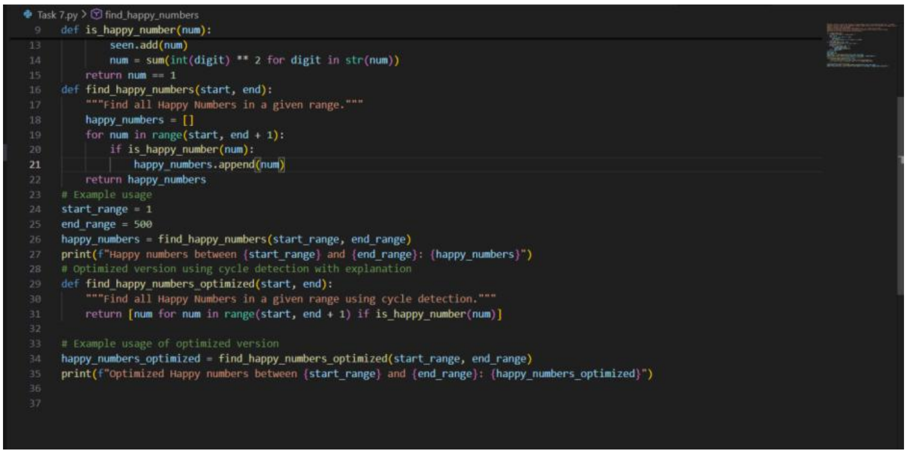
- Ask AI to regenerate an optimized version (e.g., by using a set to detect

cycles instead of infinite loops).

Expected Output #8:

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

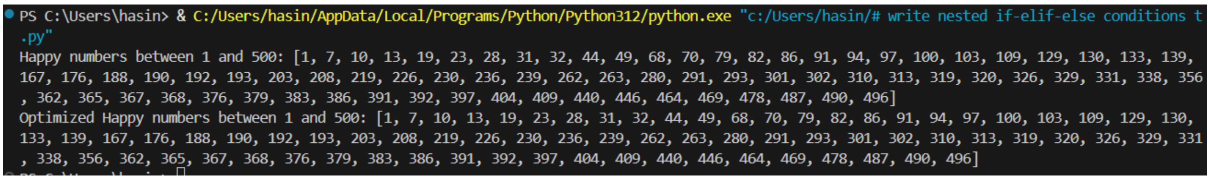
- Optimized version using cycle detection with explanation



Task 7.py > find_happy_numbers

```python
def is_happy_number(num):
            seen.add(num)
            num = sum(int(digit) ** 2 for digit in str(num))
        return num == 1
def find_happy_numbers(start, end):
    """Find all Happy Numbers in a given range."""
    happy_numbers = []
    for num in range(start, end + 1):
        if is_happy_number(num):
            happy_numbers.append(num)
    return happy_numbers
# Example usage
start_range = 1
end_range = 500
happy_numbers = find_happy_numbers(start_range, end_range)
print(f"Happy numbers between {start_range} and {end_range}: {happy_numbers}")
# Optimized version using cycle detection with explanation
def find_happy_numbers_optimized(start, end):
    """Find all Happy Numbers in a given range using cycle detection."""
    return [num for num in range(start, end + 1) if is_happy_number(num)]

# Example usage of optimized version
happy_numbers_optimized = find_happy_numbers_optimized(start_range, end_range)
print(f"Optimized Happy numbers between {start_range} and {end_range}: {happy_numbers_optimized}")
```

Output:



```
PS C:\Users\hasin> & C:/Users/hasin/AppData/Local/Programs/Python/Python312/python.exe "c:/Users/hasin/# write nested if-elif-else conditions t
.py"
Happy numbers between 1 and 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]
Optimized Happy numbers between 1 and 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]
```

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:

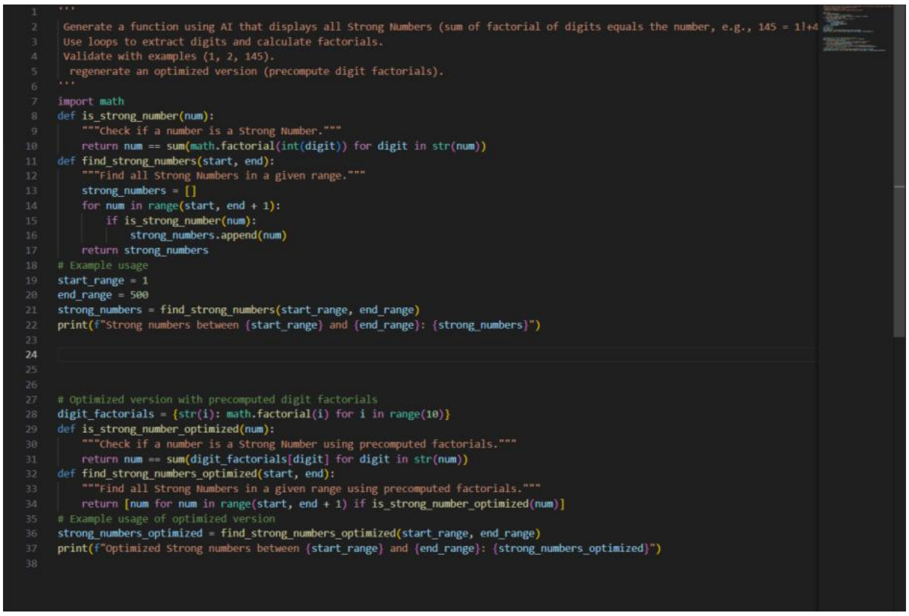
- Use loops to extract digits and calculate factorials.

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

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

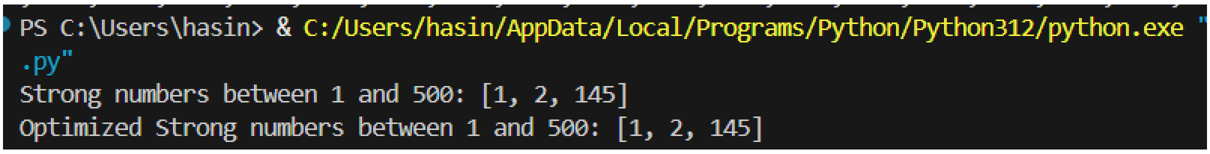
Expected Output #9:

- Python program that lists Strong Numbers.
- Optimized version with explanation.



```python
'''
 Generate a function using AI that displays all Strong Numbers (sum of factorial of digits equals the number, e.g., 145 = 1!+4
 Use loops to extract digits and calculate factorials.
 Validate with examples (1, 2, 145).
  regenerate an optimized version (precompute digit factorials).
'''
import math
def is_strong_number(num):
    """Check if a number is a Strong Number."""
    return num == sum(math.factorial(int(digit)) for digit in str(num))
def find_strong_numbers(start, end):
    """Find all Strong Numbers in a given range."""
    strong_numbers = []
    for num in range(start, end + 1):
        if is_strong_number(num):
            strong_numbers.append(num)
    return strong_numbers
# Example usage
start_range = 1
end_range = 500
strong_numbers = find_strong_numbers(start_range, end_range)
print(f"Strong numbers between {start_range} and {end_range}: {strong_numbers}")



# Optimized version with precomputed digit factorials
digit_factorials = {str(i): math.factorial(i) for i in range(10)}
def is_strong_number_optimized(num):
    """Check if a number is a Strong Number using precomputed factorials."""
    return num == sum(digit_factorials[digit] for digit in str(num))
def find_strong_numbers_optimized(start, end):
    """Find all Strong Numbers in a given range using precomputed factorials."""
    return [num for num in range(start, end + 1) if is_strong_number_optimized(num)]
# Example usage of optimized version
strong_numbers_optimized = find_strong_numbers_optimized(start_range, end_range)
print(f"Optimized Strong numbers between {start_range} and {end_range}: {strong_numbers_optimized}")
```

Output:



```
PS C:\Users\hasin> & C:/Users/hasin/AppData/Local/Programs/Python/Python312/python.exe "
.py"
Strong numbers between 1 and 500: [1, 2, 145]
Optimized Strong numbers between 1 and 500: [1, 2, 145]
```

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

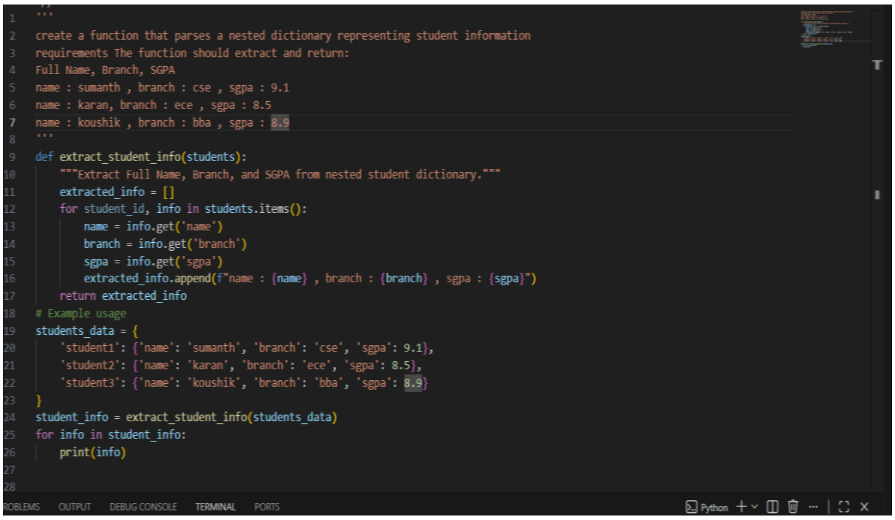
o Branch

o SGPA

Expected Output

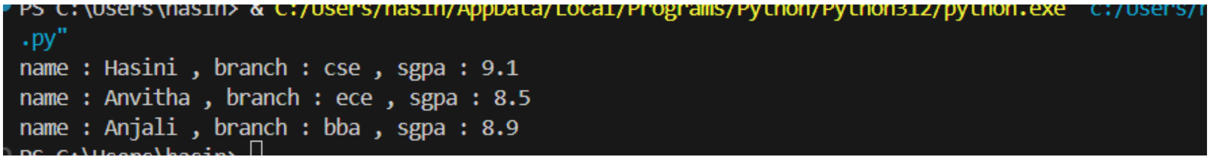
A reusable Python function that correctly navigates and extracts values

from nested dictionaries based on the provided examples



```python
'''
create a function that parses a nested dictionary representing student information
requirements The function should extract and return:
Full Name, Branch, SGPA
name : sumanth , branch : cse , sgpa : 9.1
name : karan, branch : ece , sgpa : 8.5
name : koushik , branch : bba , sgpa : 8.9
'''
def extract_student_info(students):
    """Extract Full Name, Branch, and SGPA from nested student dictionary."""
    extracted_info = []
    for student_id, info in students.items():
        name = info.get('name')
        branch = info.get('branch')
        sgpa = info.get('sgpa')
        extracted_info.append(f"name : {name} , branch : {branch} , sgpa : {sgpa}")
    return extracted_info
# Example usage
students_data = {
    'student1': {'name': 'sumanth', 'branch': 'cse', 'sgpa': 9.1},
    'student2': {'name': 'karan', 'branch': 'ece', 'sgpa': 8.5},
    'student3': {'name': 'koushik', 'branch': 'bba', 'sgpa': 8.9}
}
student_info = extract_student_info(students_data)
for info in student_info:
    print(info)
```

Output:



```
PS C:\Users\hasin> & C:/Users/hasin/AppData/Local/Programs/Python/Python312/python.exe  c:/Users/h
.py"
name : Hasini , branch : cse , sgpa : 9.1
name : Anvitha , branch : ece , sgpa : 8.5
name : Anjali , branch : bba , sgpa : 8.9
PS C:\Users\hasin> 
```

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).

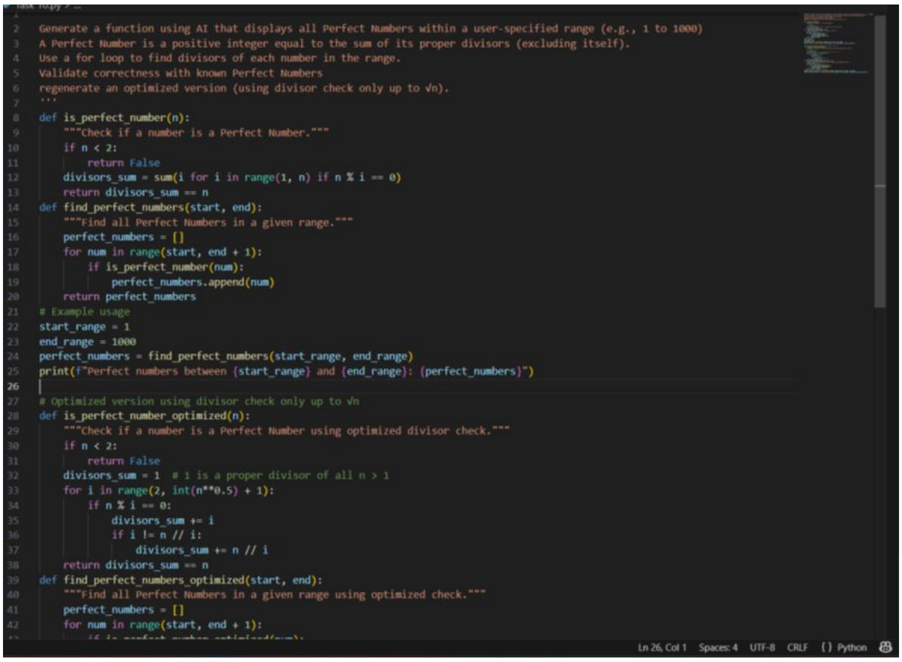
o Example: 6 = 1 + 2 + 3, 28 = 1 + 2 + 4 + 7 + 14.

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

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

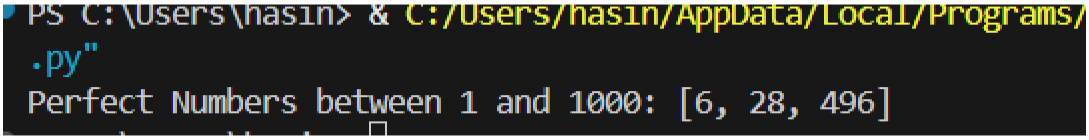
• Ask AI to regenerate an optimized version (using divisor check only up to

√n).



```python
Generate a function using AI that displays all Perfect Numbers within a user-specified range (e.g., 1 to 1000)
A Perfect Number is a positive integer equal to the sum of its proper divisors (excluding itself).
Use a for loop to find divisors of each number in the range.
Validate correctness with known Perfect Numbers
regenerate an optimized version (using divisor check only up to √n).
'''
def is_perfect_number(n):
    """Check if a number is a Perfect Number."""
    if n < 2:
        return False
    divisors_sum = sum(i for i in range(1, n) if n % i == 0)
    return divisors_sum == n
def find_perfect_numbers(start, end):
    """Find all Perfect Numbers in a given range."""
    perfect_numbers = []
    for num in range(start, end + 1):
        if is_perfect_number(num):
            perfect_numbers.append(num)
    return perfect_numbers
# Example usage
start_range = 1
end_range = 1000
perfect_numbers = find_perfect_numbers(start_range, end_range)
print(f"Perfect numbers between {start_range} and {end_range}: {perfect_numbers}")

# Optimized version using divisor check only up to √n
def is_perfect_number_optimized(n):
    """Check if a number is a Perfect Number using optimized divisor check."""
    if n < 2:
        return False
    divisors_sum = 1  # 1 is a proper divisor of all n > 1
    for i in range(2, int(n**0.5) + 1):
        if n % i == 0:
            divisors_sum += i
            if i != n // i:
                divisors_sum += n // i
    return divisors_sum == n
def find_perfect_numbers_optimized(start, end):
    """Find all Perfect Numbers in a given range using optimized check."""
    perfect_numbers = []
    for num in range(start, end + 1):
```

Output:



```
PS C:\Users\hasin> & C:/Users/hasin/AppData/Local/Programs/
.py"
Perfect Numbers between 1 and 1000: [6, 28, 496]
```