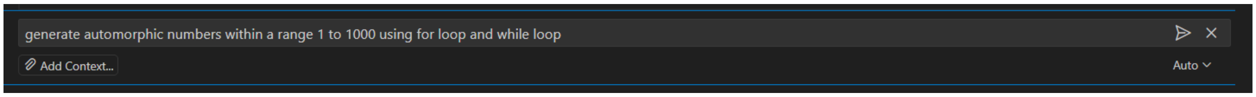
AI ASSISTED CODING
ASSIGNMENT - 6.3

Name: B.Siri Chandana
HTNo: 2303A52004
Batch:31

Question:

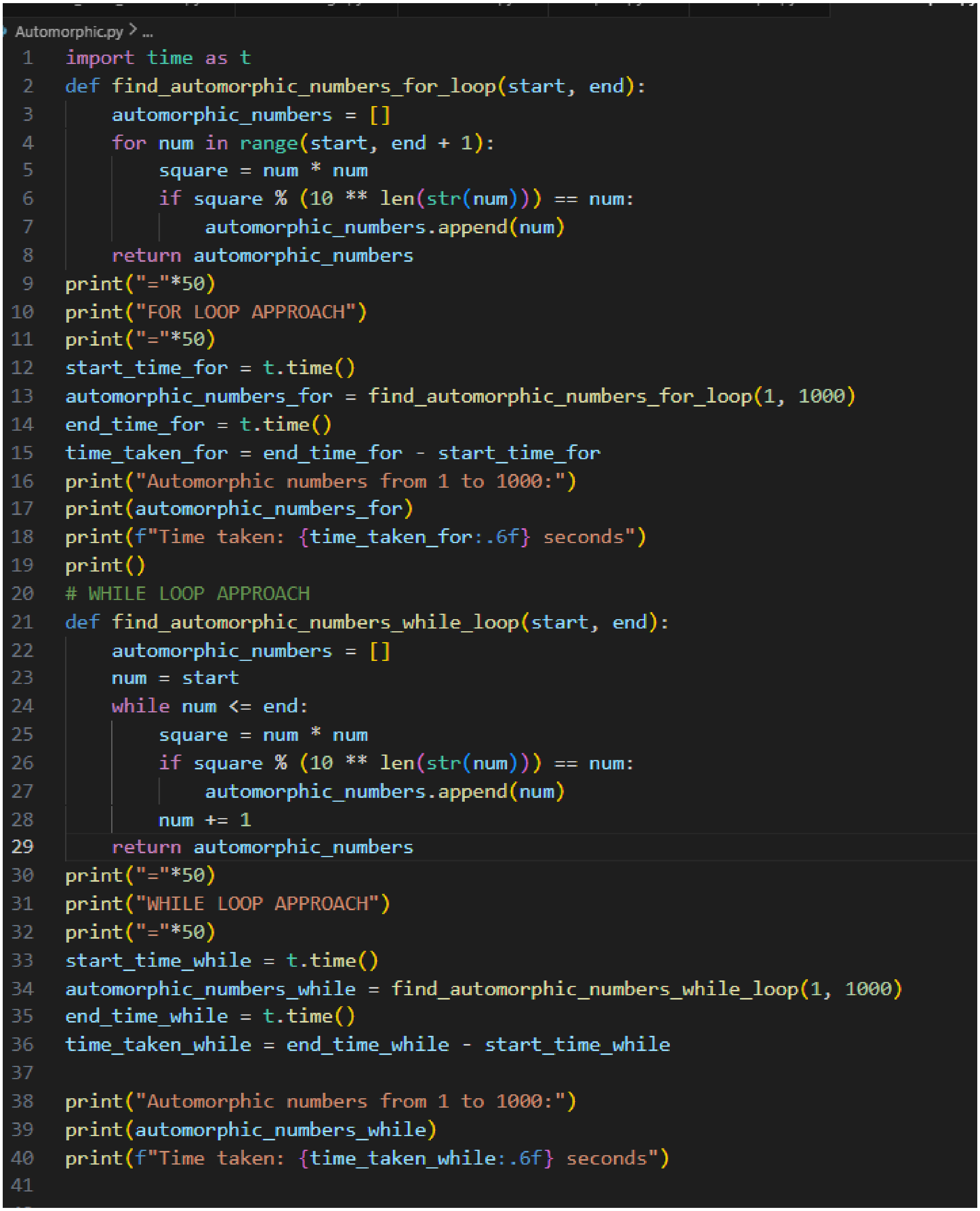
Task Description #1 (Loops – Automorphic Numbers in a Range)
• Task: Prompt AI to generate a function that displays all Automorphic numbers between 1 and 1000 using a for loop.
• Instructions:
o Get AI-generated code to list Automorphic numbers using a for loop.
o Analyze the correctness and efficiency 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:



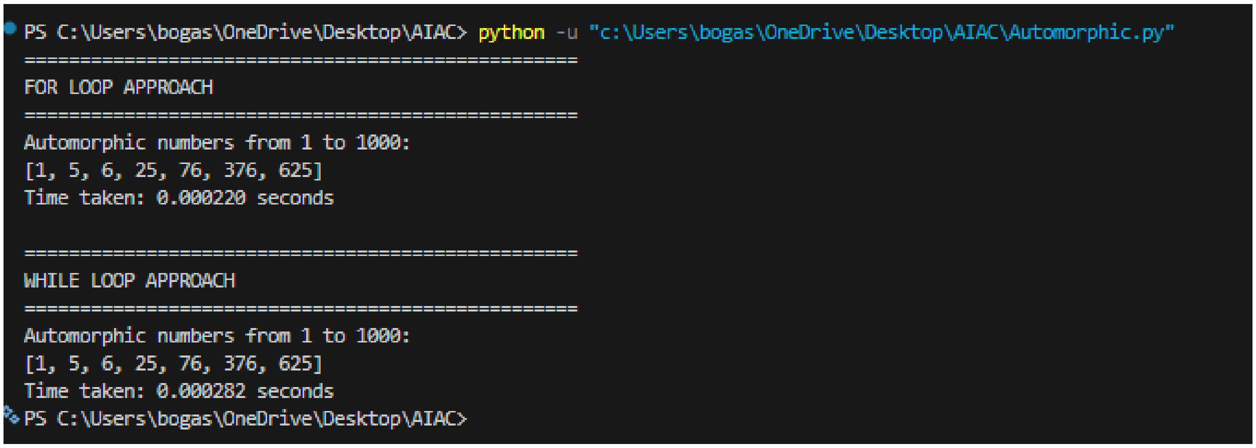
generate automorphic numbers within a range 1 to 1000 using for loop and while loop

Code:



```python
import time as t
def find_automorphic_numbers_for_loop(start, end):
    automorphic_numbers = []
    for num in range(start, end + 1):
        square = num * num
        if square % (10 ** len(str(num))) == num:
            automorphic_numbers.append(num)
    return automorphic_numbers
print("="*50)
print("FOR LOOP APPROACH")
print("="*50)
start_time_for = t.time()
automorphic_numbers_for = find_automorphic_numbers_for_loop(1, 1000)
end_time_for = t.time()
time_taken_for = end_time_for - start_time_for
print("Automorphic numbers from 1 to 1000:")
print(automorphic_numbers_for)
print(f"Time taken: {time_taken_for:.6f} seconds")
print()
# WHILE LOOP APPROACH
def find_automorphic_numbers_while_loop(start, end):
    automorphic_numbers = []
    num = start
    while num <= end:
        square = num * num
        if square % (10 ** len(str(num))) == num:
            automorphic_numbers.append(num)
        num += 1
    return automorphic_numbers
print("="*50)
print("WHILE LOOP APPROACH")
print("="*50)
start_time_while = t.time()
automorphic_numbers_while = find_automorphic_numbers_while_loop(1, 1000)
end_time_while = t.time()
time_taken_while = end_time_while - start_time_while

print("Automorphic numbers from 1 to 1000:")
print(automorphic_numbers_while)
print(f"Time taken: {time_taken_while:.6f} seconds")

```

Output:



```
PS C:\Users\bogas\OneDrive\Desktop\AIAC> python -u "c:\Users\bogas\OneDrive\Desktop\AIAC\Automorphic.py"
==================================================
FOR LOOP APPROACH
==================================================
Automorphic numbers from 1 to 1000:
[1, 5, 6, 25, 76, 376, 625]
Time taken: 0.000220 seconds

==================================================
WHILE LOOP APPROACH
==================================================
Automorphic numbers from 1 to 1000:
[1, 5, 6, 25, 76, 376, 625]
Time taken: 0.000282 seconds
PS C:\Users\bogas\OneDrive\Desktop\AIAC>
```

Justification:

The time complexity to find automorphic numbers within range 1 to 1000 using for loop is less than the time complexity using while loop

Question:

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

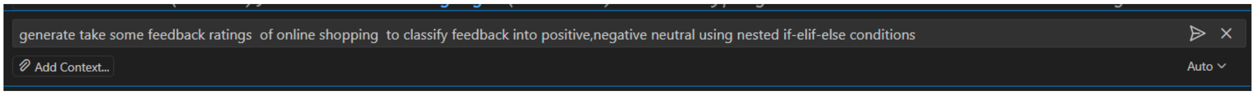
- Task: Ask AI to write nested if-elif-else conditions to classify online shopping feedback as Positive, Neutral, or Negative based on a numerical rating (1–5).
- Instructions:

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:

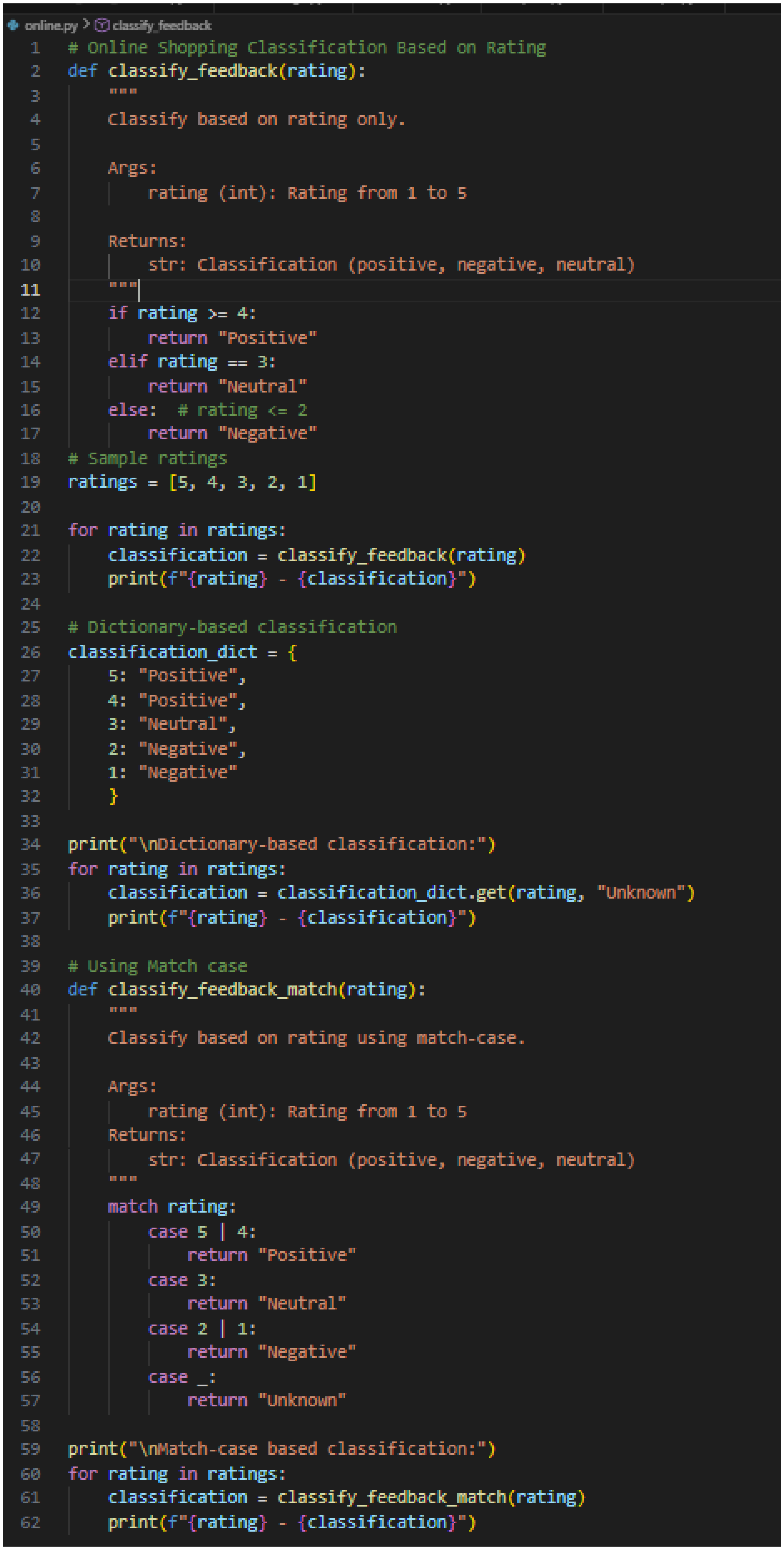
- Feedback classification function with explanation and an alternative Approach.

Prompt:



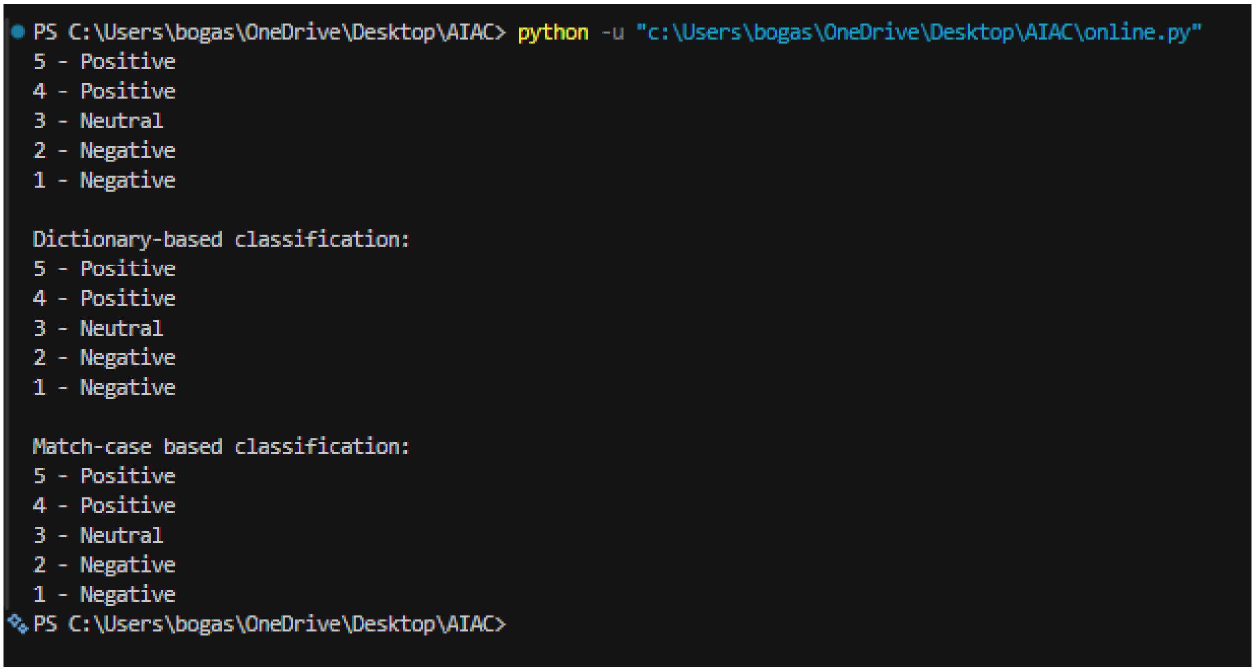
generate take some feedback ratings of online shopping to classify feedback into positive,negative neutral using nested if-elif-else conditions

Code:



```python
# Online Shopping Classification Based on Rating
def classify_feedback(rating):
    """
    Classify based on rating only.

    Args:
        rating (int): Rating from 1 to 5

    Returns:
        str: Classification (positive, negative, neutral)
    """
    if rating >= 4:
        return "Positive"
    elif rating == 3:
        return "Neutral"
    else:  # rating <= 2
        return "Negative"
# Sample ratings
ratings = [5, 4, 3, 2, 1]

for rating in ratings:
    classification = classify_feedback(rating)
    print(f"{rating} - {classification}")

# Dictionary-based classification
classification_dict = {
    5: "Positive",
    4: "Positive",
    3: "Neutral",
    2: "Negative",
    1: "Negative"
    }

print("\nDictionary-based classification:")
for rating in ratings:
    classification = classification_dict.get(rating, "Unknown")
    print(f"{rating} - {classification}")

# Using Match case
def classify_feedback_match(rating):
    """
    Classify based on rating using match-case.

    Args:
        rating (int): Rating from 1 to 5
    Returns:
        str: Classification (positive, negative, neutral)
    """
    match rating:
        case 5 | 4:
            return "Positive"
        case 3:
            return "Neutral"
        case 2 | 1:
            return "Negative"
        case _:
            return "Unknown"

print("\nMatch-case based classification:")
for rating in ratings:
    classification = classify_feedback_match(rating)
    print(f"{rating} - {classification}")
```

Output:



```
PS C:\Users\bogas\OneDrive\Desktop\AIAC> python -u "c:\Users\bogas\OneDrive\Desktop\AIAC\online.py"
5 - Positive
4 - Positive
3 - Neutral
2 - Negative
1 - Negative

Dictionary-based classification:
5 - Positive
4 - Positive
3 - Neutral
2 - Negative
1 - Negative

Match-case based classification:
5 - Positive
4 - Positive
3 - Neutral
2 - Negative
1 - Negative
PS C:\Users\bogas\OneDrive\Desktop\AIAC>
```

Justification:

This program classifies online shopping feedback based on ratings from 1 to 5 using conditional statements.

It correctly maps higher ratings to Positive, 3 to Neutral, and lower ratings to Negative, ensuring clear and readable logic.

Question:

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

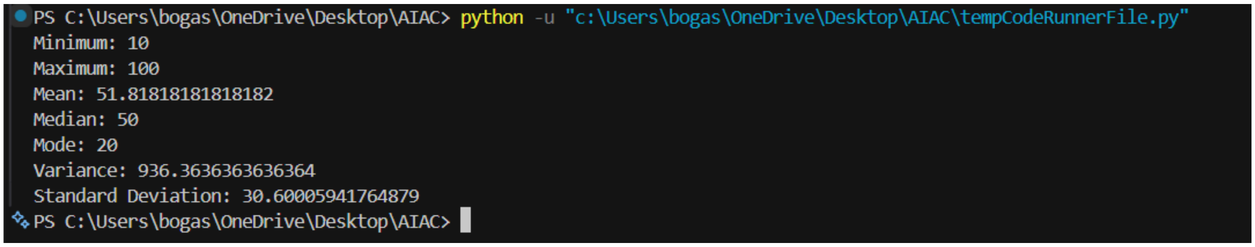
Prompt:

```
# Define a function named statistical_operations(tuple_num)
# that performs the statistical operations on a tuple of numbers:
# Minimum, Maximum
# Mean, Median, Mode
# Variance, Standard Deviation
```

Code:

```python
import statistics
def statistical_operations(tuple_num):
    results = {}
    results['Minimum'] = min(tuple_num)
    results['Maximum'] = max(tuple_num)
    results['Mean'] = statistics.mean(tuple_num)
    results['Median'] = statistics.median(tuple_num)
    try:
        results['Mode'] = statistics.mode(tuple_num)
    except statistics.StatisticsError:
        results['Mode'] = "No unique mode"
    results['Variance'] = statistics.variance(tuple_num)
    results['Standard Deviation'] = statistics.stdev(tuple_num)
    return results
# Sample tuple of numbers
numbers = (10, 20, 20, 30, 40, 50, 60, 70, 80, 90, 100)
# Perform statistical operations
stats = statistical_operations(numbers)
# Print the results
for key, value in stats.items():
    print(f"{key}: {value}")
```

Output:



```
PS C:\Users\bogas\OneDrive\Desktop\AIAC> python -u "c:\Users\bogas\OneDrive\Desktop\AIAC\tempCodeRunnerFile.py"
Minimum: 10
Maximum: 100
Mean: 51.81818181818182
Median: 50
Mode: 20
Variance: 936.3636363636364
Standard Deviation: 30.60005941764879
PS C:\Users\bogas\OneDrive\Desktop\AIAC> 
```

Justification:

The suggested use of Python’s `statistics` module is correct and efficient, so it was accepted.A `try–except` block for `mode` was added to safely handle cases with no unique mode.Built-in functions like `min`, `max`, and library methods improve readability and reduce errors.

Question:

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.

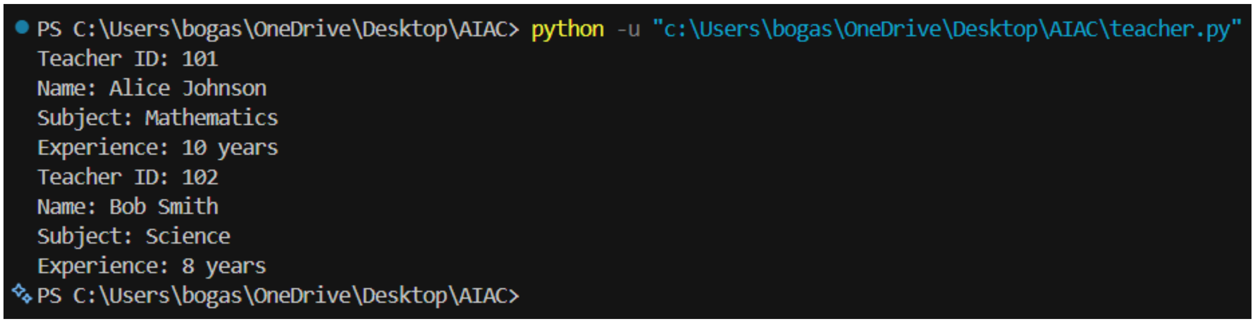
Prompt:

```
# Create a class Teacher with attributes teacher_id, name,subject, and experience.
# Add a method to display teacher details.
```

Code:

```
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")
# Create an instance of the Teacher class and display details
teacher1 = Teacher(101, "Alice Johnson", "Mathematics", 10)
teacher2 = Teacher(102, "Bob Smith", "Science", 8)
teacher1.display_details()
teacher2.display_details()
```

Output:



```
PS C:\Users\bogas\OneDrive\Desktop\AIAC> python -u "c:\Users\bogas\OneDrive\Desktop\AIAC\teacher.py"
Teacher ID: 101
Name: Alice Johnson
Subject: Mathematics
Experience: 10 years
Teacher ID: 102
Name: Bob Smith
Subject: Science
Experience: 8 years
PS C:\Users\bogas\OneDrive\Desktop\AIAC>
```

Justification:

The `Teacher` class correctly initializes all required attributes using a constructor and includes a method to display teacher details.
Creating objects and calling the method produces the expected output, confirming correct class design and usage.

Question:

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.

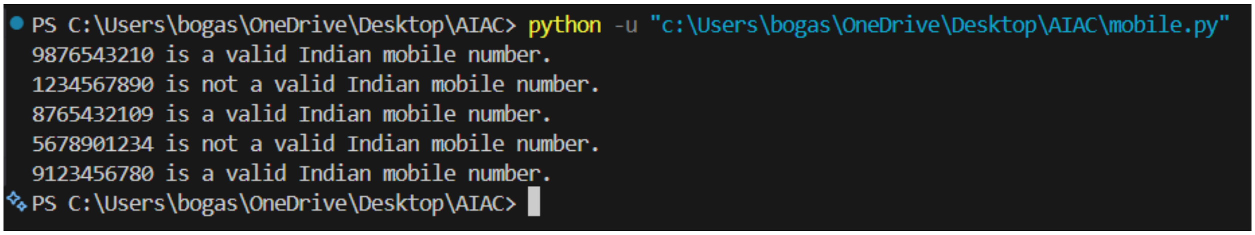
Prompt:

```
# generate a function that validates an Indian mobile number starts
with 6,7,8,9 and is of 10 digits
```

Code:

```
import re
def validate_mobile_number(mobile_number):
    pattern = r'^[6-9]\d{9}$'
    if re.match(pattern, mobile_number):
        return True
    else:
        return False
# Sample mobile numbers for testing
mobile_numbers = ["9876543210", "1234567890", "8765432109",
"5678901234", "9123456780"]
for number in mobile_numbers:
    if validate_mobile_number(number):
        print(f"{number} is a valid Indian mobile number.")
    else:
        print(f"{number} is not a valid Indian mobile number.")
```

Output:



```
PS C:\Users\bogas\OneDrive\Desktop\AIAC> python -u "c:\Users\bogas\OneDrive\Desktop\AIAC\mobile.py"
9876543210 is a valid Indian mobile number.
1234567890 is not a valid Indian mobile number.
8765432109 is a valid Indian mobile number.
5678901234 is not a valid Indian mobile number.
9123456780 is a valid Indian mobile number.
PS C:\Users\bogas\OneDrive\Desktop\AIAC> 
```

Justification:

The regular expression ensures the mobile number starts with 6, 7, 8, or 9 and contains exactly 10 digits. Using `re.match` validates the entire string format and returns a clear True or False result.

Question:

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.).

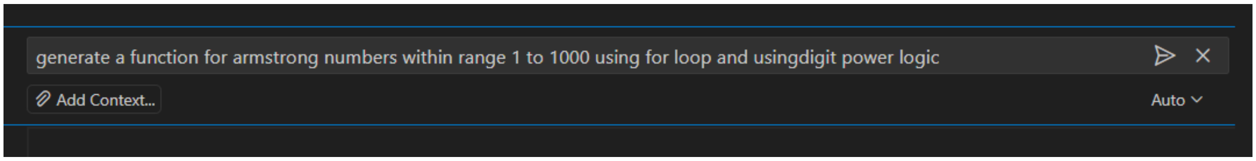
- Ask AI to regenerate an optimized version (using list comprehensions).

Expected Output #7:

- Python program listing Armstrong numbers in the range.

- Optimized version with explanation.

Prompt:



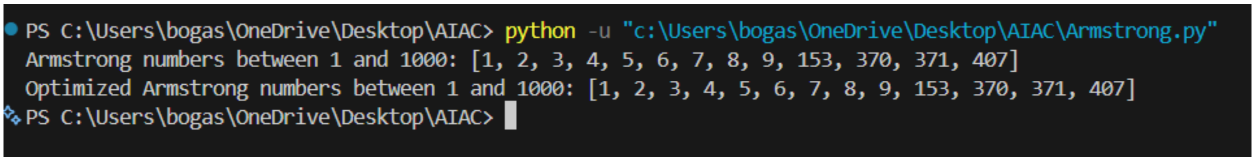
generate a function for armstrong numbers within range 1 to 1000 using for loop and usingdigit power logic

Code:

```
def armstrong_numbers(start, end):

    """

    Find all Armstrong numbers in the given range.
    An Armstrong number is a number that is equal to the sum of its digits each raised to the power of the number of digits.
    """
    armstrong_list = []

    for num in range(start, end + 1):
        # Convert number to string to get digits
        digits = str(num)
        num_digits = len(digits)

        # Calculate sum of each digit raised to the power of number of digits
        digit_sum = sum(int(digit) ** num_digits for digit in digits)

        # Check if number is Armstrong number
        if digit_sum == num:
            armstrong_list.append(num)

    return armstrong_list
```

```python
# Test the function
result = armstrong_numbers(1, 1000)
print(f"Armstrong numbers between 1 and 1000: {result}")


#optimized version using list comprehension
def armstrong_numbers_optimized(start, end):
    """
    Find all Armstrong numbers in the given range using list comprehension.
    An Armstrong number is a number that is equal to the sum of its digits each raised to the power of the number of digits.
    """
    return [num for num in range(start, end + 1)
            if sum(int(digit) ** len(str(num)) for digit in str(num)) == num]


# Test the optimized function
optimized_result = armstrong_numbers_optimized(1, 1000)
print(f"Optimized Armstrong numbers between 1 and 1000: {optimized_result}")
```

Output:



```
PS C:\Users\bogas\OneDrive\Desktop\AIAC> python -u "c:\Users\bogas\OneDrive\Desktop\AIAC\Armstrong.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]
PS C:\Users\bogas\OneDrive\Desktop\AIAC>
```

Justification:

The program checks each number to see if it equals the sum of its digits raised to the power of the number of digits. This correctly finds all Armstrong numbers, including single digits (1–9) and known 3-digit ones like 153, 370, 371, and 407.

Question:

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.

Prompt:

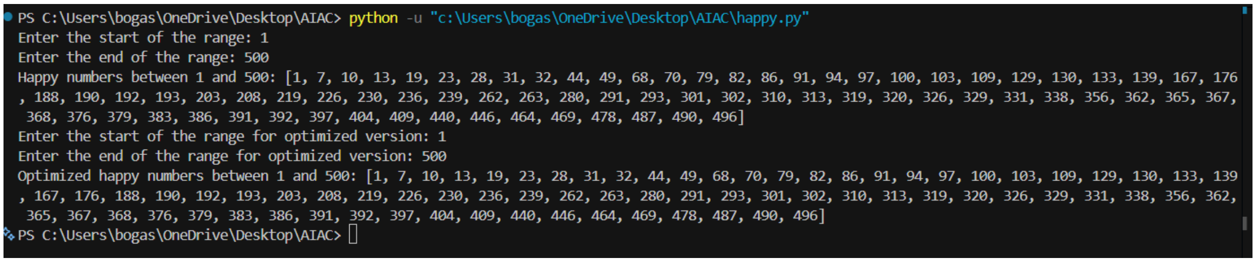
```
#Generate a function using AI that displays all Happy Numbers within a user-specified range (e.g., 1 to 500).

#repeatedly replace a number with the sum of the squares of its digits until the result is either 1 or enters a cycle
```

Code:

```
def is_happy_number(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

def happy_numbers_in_range(start, end):

    happy_numbers = []

    for num in range(start, end + 1):

        if is_happy_number(num):

            happy_numbers.append(num)

    return happy_numbers

# Get user input for range
```

```python
start_range = int(input("Enter the start of the range: "))
end_range = int(input("Enter the end of the range: "))
happy_numbers = happy_numbers_in_range(start_range, end_range)
print(f"Happy numbers between {start_range} and {end_range}: {happy_numbers}")


# optimized version
def happy_numbers_in_range_optimized(start, end):
    return [num for num in range(start, end + 1) if is_happy_number(num)]
# Get user input for range
start_range_opt = int(input("Enter the start of the range for optimized version: "))
end_range_opt = int(input("Enter the end of the range for optimized version: "))
happy_numbers_opt = happy_numbers_in_range_optimized(start_range_opt, end_range_opt)
print(f"Optimized happy numbers between {start_range_opt} and {end_range_opt}: {happy_numbers_opt}")
```

Output:



```
PS C:\Users\bogas\OneDrive\Desktop\AIAC> python -u "c:\Users\bogas\OneDrive\Desktop\AIAC\happy.py"
Enter the start of the range: 1
Enter the end of the range: 500
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]
Enter the start of the range for optimized version: 1
Enter the end of the range for optimized version: 500
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]
PS C:\Users\bogas\OneDrive\Desktop\AIAC> 
```

Justification:

A Happy Number is a number that eventually reaches 1 when you keep replacing it with the sum of the squares of its digits, and this program finds all such numbers in a range by keeping track of repeats to avoid infinite loops.

Question:

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).

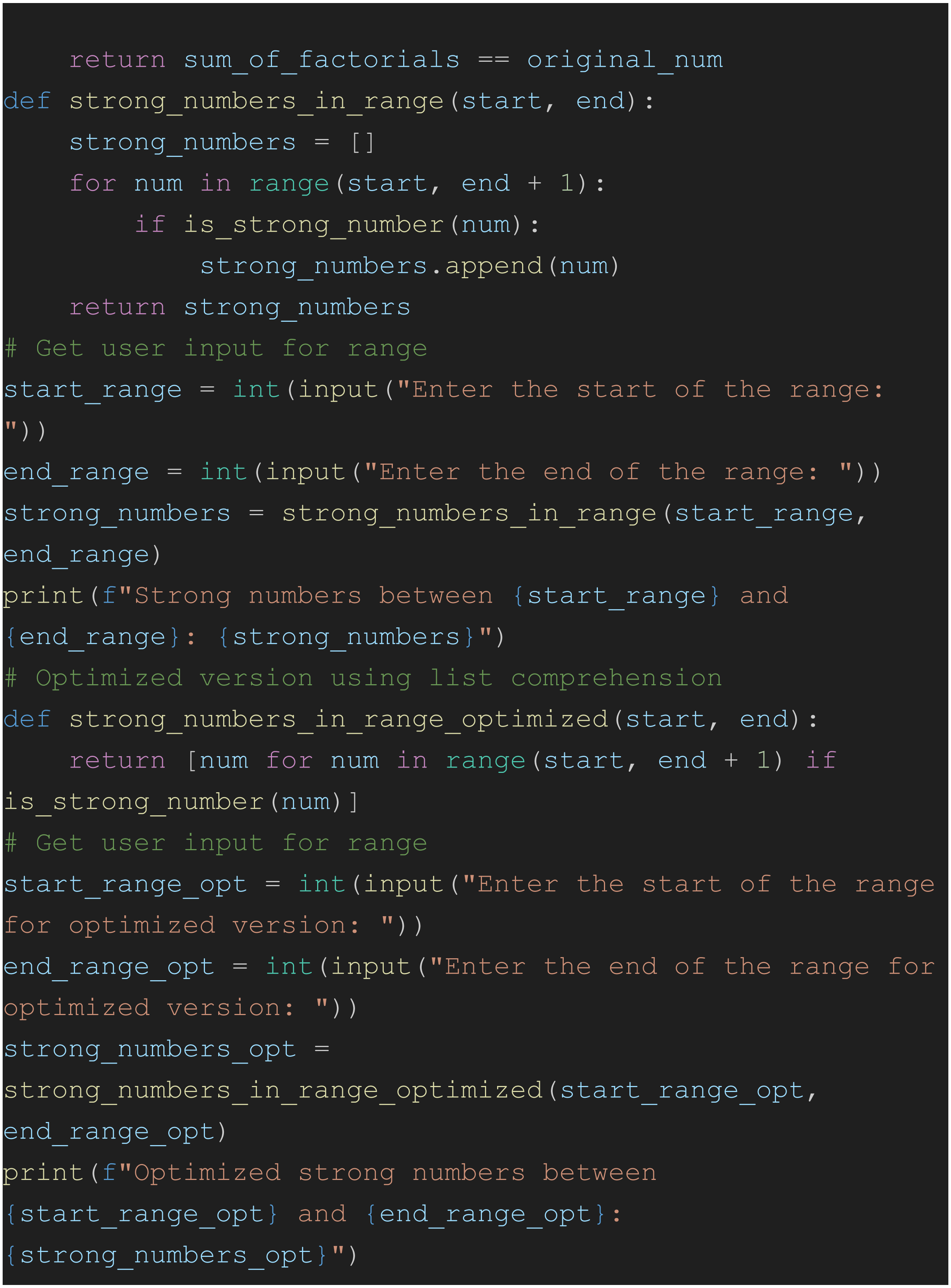
Expected Output #9:
- Python program that lists Strong Numbers.
- Optimized version with explanation.

Prompt:

```
# Generate a function that displays all Strong Numbers
within a given range.
# Use loops to extract digits and calculate factorials.
```

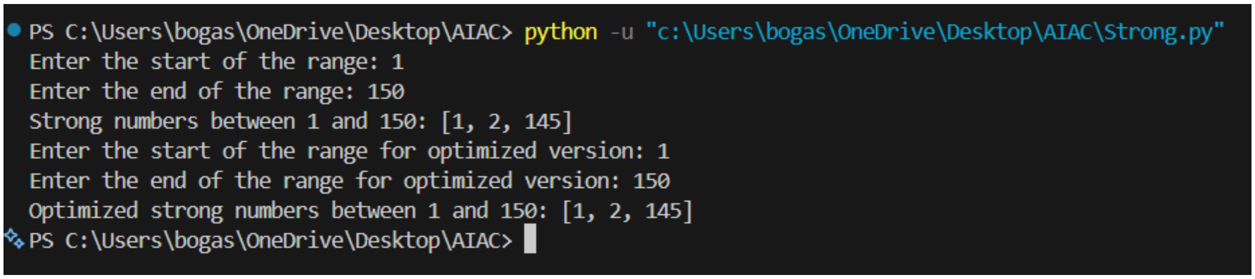
Code:

```
import math
def is_strong_number(num):
    original_num = num
    sum_of_factorials = 0

    while num > 0:
        digit = num % 10
        sum_of_factorials += math.factorial(digit)
        num //= 10
```



```
    return sum_of_factorials == original_num
def strong_numbers_in_range(start, end):
    strong_numbers = []
    for num in range(start, end + 1):
        if is_strong_number(num):
            strong_numbers.append(num)
    return strong_numbers
# Get user input for range
start_range = int(input("Enter the start of the range:
"))
end_range = int(input("Enter the end of the range: "))
strong_numbers = strong_numbers_in_range(start_range,
end_range)
print(f"Strong numbers between {start_range} and
{end_range}: {strong_numbers}")
# Optimized version using list comprehension
def strong_numbers_in_range_optimized(start, end):
    return [num for num in range(start, end + 1) if
is_strong_number(num)]
# Get user input for range
start_range_opt = int(input("Enter the start of the range
for optimized version: "))
end_range_opt = int(input("Enter the end of the range for
optimized version: "))
strong_numbers_opt =
strong_numbers_in_range_optimized(start_range_opt,
end_range_opt)
print(f"Optimized strong numbers between
{start_range_opt} and {end_range_opt}:
{strong_numbers_opt}")
```

Output:



```
PS C:\Users\bogas\OneDrive\Desktop\AIAC> python -u "c:\Users\bogas\OneDrive\Desktop\AIAC\Strong.py"
Enter the start of the range: 1
Enter the end of the range: 150
Strong numbers between 1 and 150: [1, 2, 145]
Enter the start of the range for optimized version: 1
Enter the end of the range for optimized version: 150
Optimized strong numbers between 1 and 150: [1, 2, 145]
PS C:\Users\bogas\OneDrive\Desktop\AIAC>
```

Justification:

A Strong Number equals the sum of the factorials of its digits, and the program checks this for each number in the range.It correctly identifies known Strong Numbers like 1, 2, and 145, and the optimized version does the same more concisely.

Question:

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

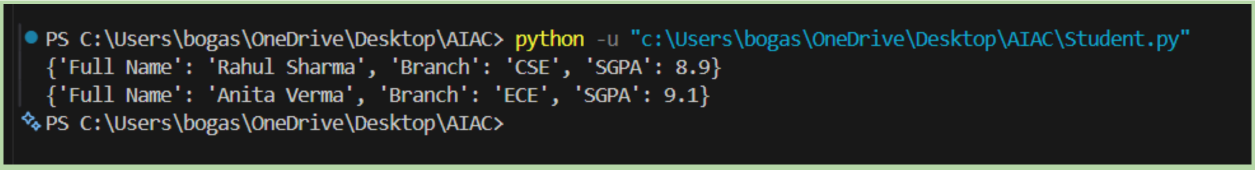
Prompt:

```
# You are given a nested dictionary containing student information.
#Write a Python function that extracts and returns the Full Name, Branch, and SGPA.
#eg:Input: A student whose first name is Rahul, last name is Sharma, branch is CSE, and SGPA is 8.9.
#eg:Input: A student whose first name is Anita, last name is Verma, branch is ECE, and SGPA is 9.1.
```

Code:

```python
def extract_student_info(student_dict):
    first_name = student_dict.get('first_name', '')
    last_name = student_dict.get('last_name', '')
    branch = student_dict.get('branch', '')
    sgpa = student_dict.get('sgpa', '')

    full_name = f"{first_name} {last_name}"

    return {
        'Full Name': full_name,
        'Branch': branch,
        'SGPA': sgpa
    }
# Example usage:
student_1 = {
    'first_name': 'Rahul',
    'last_name': 'Sharma',
    'branch': 'CSE',
    'sgpa': 8.9
}
student_2 = {
    'first_name': 'Anita',
    'last_name': 'Verma',
    'branch': 'ECE',
    'sgpa': 9.1
}
info_1 = extract_student_info(student_1)
info_2 = extract_student_info(student_2)
print(info_1)
print(info_2)
```

Output:



```
PS C:\Users\bogas\OneDrive\Desktop\AIAC> python -u "c:\Users\bogas\OneDrive\Desktop\AIAC\Student.py"
 {'Full Name': 'Rahul Sharma', 'Branch': 'CSE', 'SGPA': 8.9}
 {'Full Name': 'Anita Verma', 'Branch': 'ECE', 'SGPA': 9.1}
PS C:\Users\bogas\OneDrive\Desktop\AIAC>
```

Justification:

The function correctly extracts the first name, last name, branch, and SGPA from the given student data and combines the names to form the full name.

Using the few-shot examples as guidance, it successfully returns the required details in a reusable and structured format.

Question:

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).

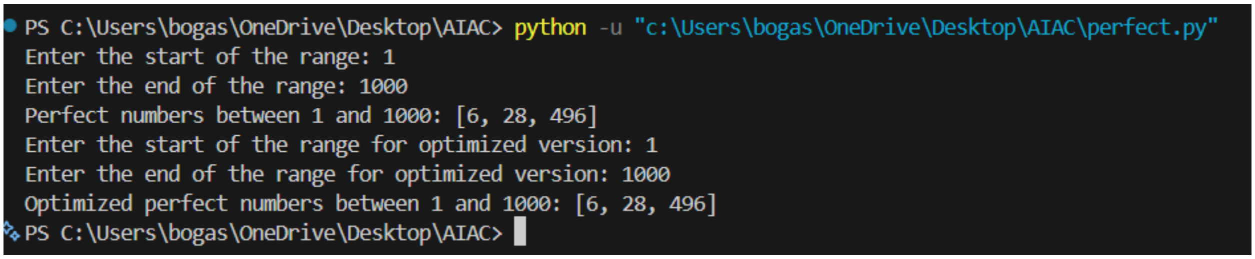
Prompt:

```
#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.
```

Code:

```
def is_perfect_number(num):
    if num < 2:
        return False
    sum_of_divisors = 0
    for i in range(1, num):
        if num % i == 0:
            sum_of_divisors += i
    return sum_of_divisors == num
def perfect_numbers_in_range(start, end):
    perfect_numbers = []
    for num in range(start, end + 1):
        if is_perfect_number(num):
            perfect_numbers.append(num)
    return perfect_numbers
# Get user input for range
start_range = int(input("Enter the start of the range: "))
end_range = int(input("Enter the end of the range: "))
perfect_numbers = perfect_numbers_in_range(start_range, end_range)
print(f"Perfect numbers between {start_range} and {end_range}: {perfect_numbers}")
# Optimized version by checking upto root n
def is_perfect_number_optimized(num):
    if num < 2:
        return False
    sum_of_divisors = 1  # 1 is a proper divisor of all numbers > 1
    for i in range(2, int(num**0.5) + 1):
        if num % i == 0:
            sum_of_divisors += i
            if i != num // i:
                sum_of_divisors += num // i
    return sum_of_divisors == num
def perfect_numbers_in_range_optimized(start, end):
    perfect_numbers = []
    for num in range(start, end + 1):
        if is_perfect_number_optimized(num):
            perfect_numbers.append(num)
    return perfect_numbers
# Get user input for range
start_range_opt = int(input("Enter the start of the range for optimized version: "))
end_range_opt = int(input("Enter the end of the range for optimized version: "))
perfect_numbers_opt = perfect_numbers_in_range_optimized(start_range_opt, end_range_opt)
print(f"Optimized perfect numbers between {start_range_opt} and {end_range_opt}: {perfect_numbers_opt}")
```

Output:



```
PS C:\Users\bogas\OneDrive\Desktop\AIAC> python -u "c:\Users\bogas\OneDrive\Desktop\AIAC\perfect.py"
Enter the start of the range: 1
Enter the end of the range: 1000
Perfect numbers between 1 and 1000: [6, 28, 496]
Enter the start of the range for optimized version: 1
Enter the end of the range for optimized version: 1000
Optimized perfect numbers between 1 and 1000: [6, 28, 496]
PS C:\Users\bogas\OneDrive\Desktop\AIAC>
```

Justification:

The program checks whether a number equals the sum of its proper divisors, which correctly identifies known Perfect Numbers like 6, 28, and 496.The optimized version improves efficiency by finding divisors only up to √n while producing the same correct results.