ASSIGNMENT - 06

Program Name: B. Tech **Assignment Type:** Lab **Academic Year:**2025-2026

Instructor(s) Name : S Naresh Kumar **Course Title :** AI Assisted Coding

Name : Aravind Reddy **Hall Ticket No :** 2303a51027

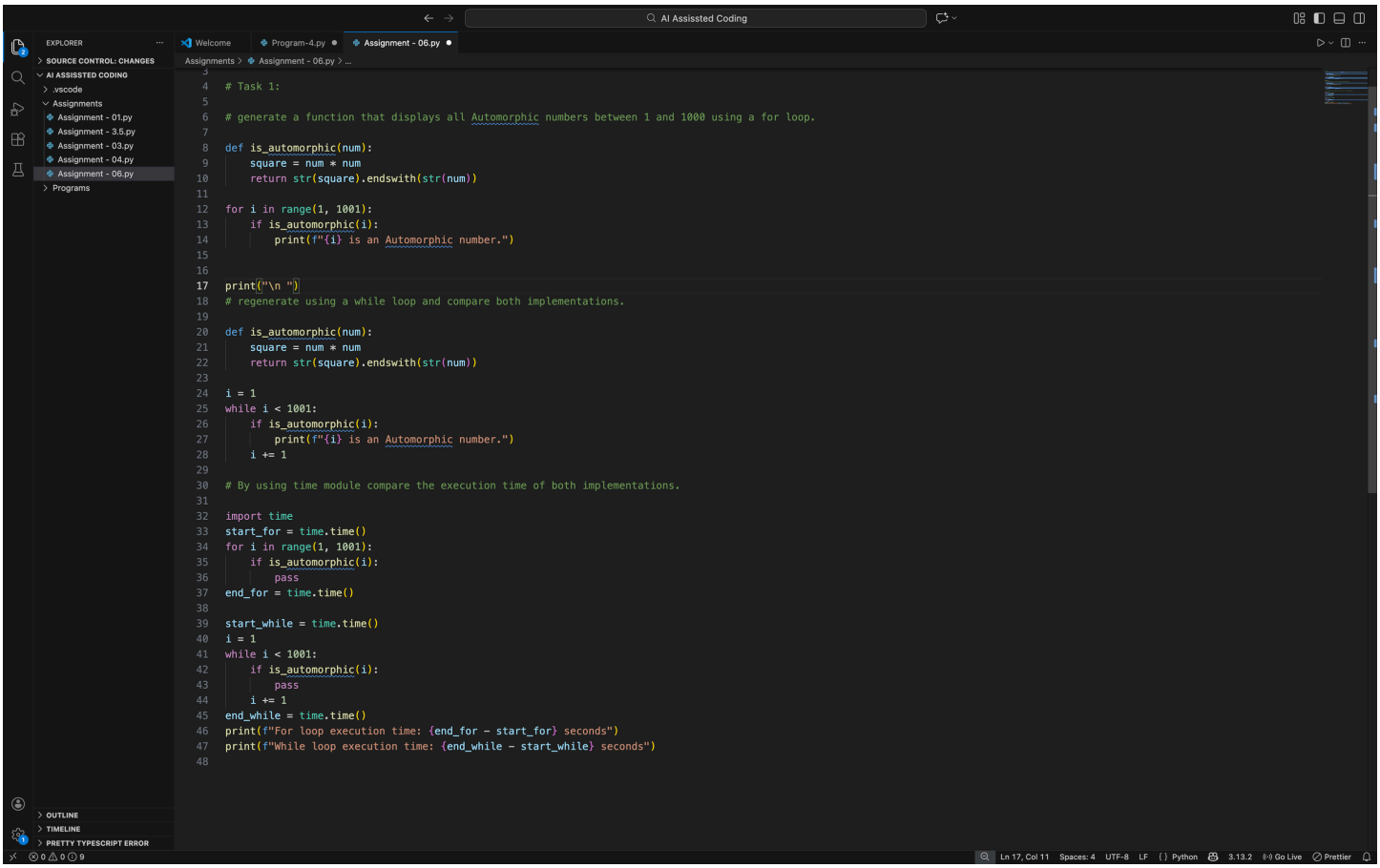
Task Description #1 (Loops – Automorphic Numbers in a Range)

- Task: Prompt AI to generate a function that displays all Automorphic numbers between 1 and 1000 using a for loop.
- Instructions:
- Get AI-generated code to list Automorphic numbers using a for loop.
- Analyse the correctness and efficiency of the generated logic.
- Ask AI to regenerate using a while loop and compare both implementations.

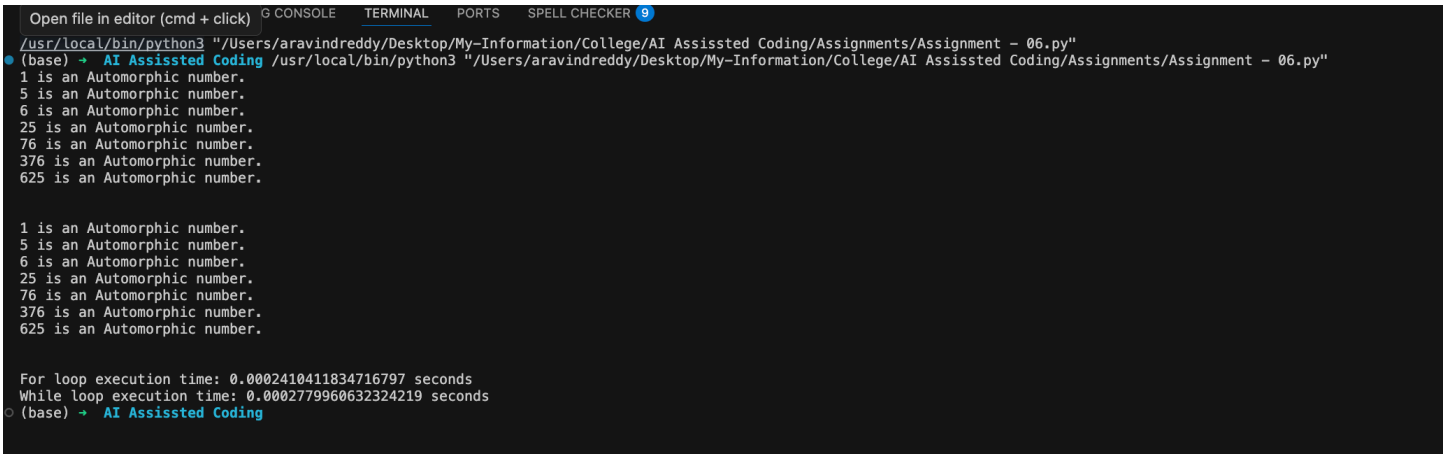
Expected Output #1:

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

Code :



Output :



Justification:

for loop being generally faster than while loop in Python due to optimisations in byte code execution.

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).

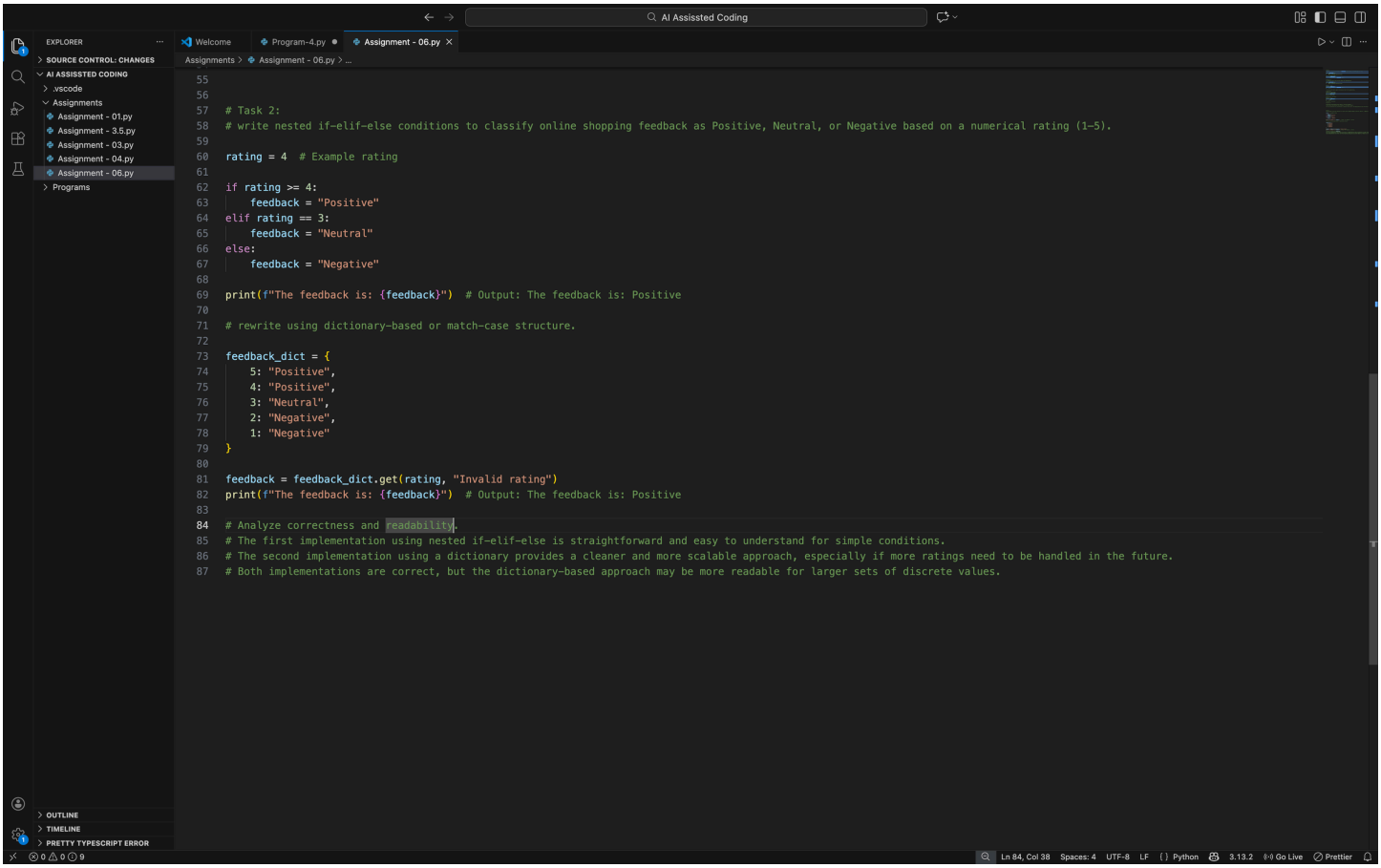
- **Instructions:**

  - Generate initial code using nested if-elif-else.
  - Analyse correctness and readability.
  - Ask AI to rewrite using dictionary-based or match-case structure.

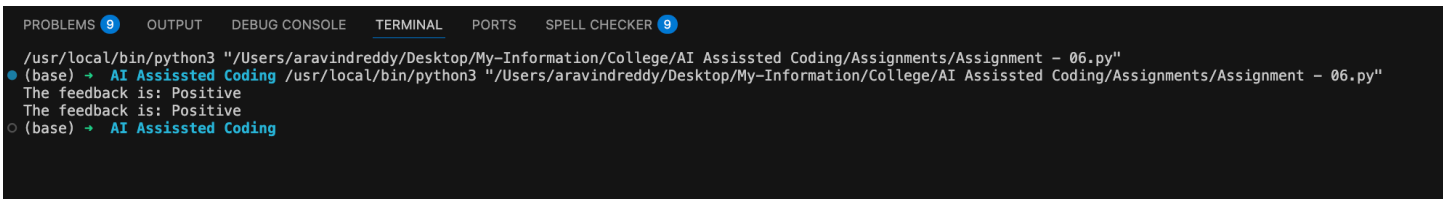
Expected Output #2:

- **Feedback classification function with explanation and an alternative approach.**

Code:



Output :



Justification:

```
# Analyse correctness and readability.

# The first implementation using nested if-elif-else is straightforward and easy to understand for simple conditions.

# The second implementation using a dictionary provides a cleaner and more scalable approach, especially if more ratings need to be handled in the future.

# Both implementations are correct, but the dictionary-based approach may be more readable for larger sets of discrete values.
```

Task 3: Statistical_operations

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

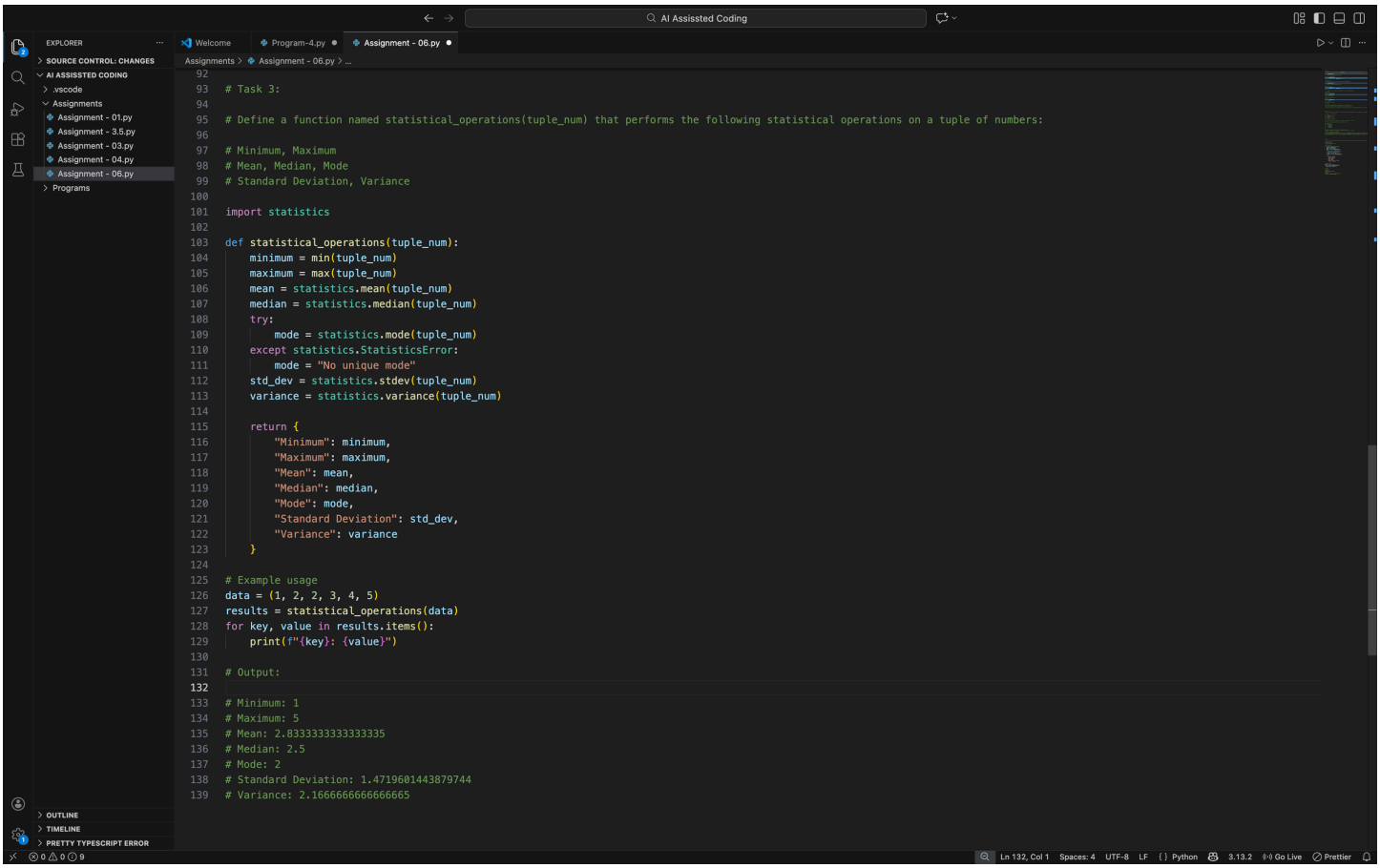
- **Minimum, Maximum**
- **Mean, Median, Mode**
- **Variance, Standard Deviation**

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

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

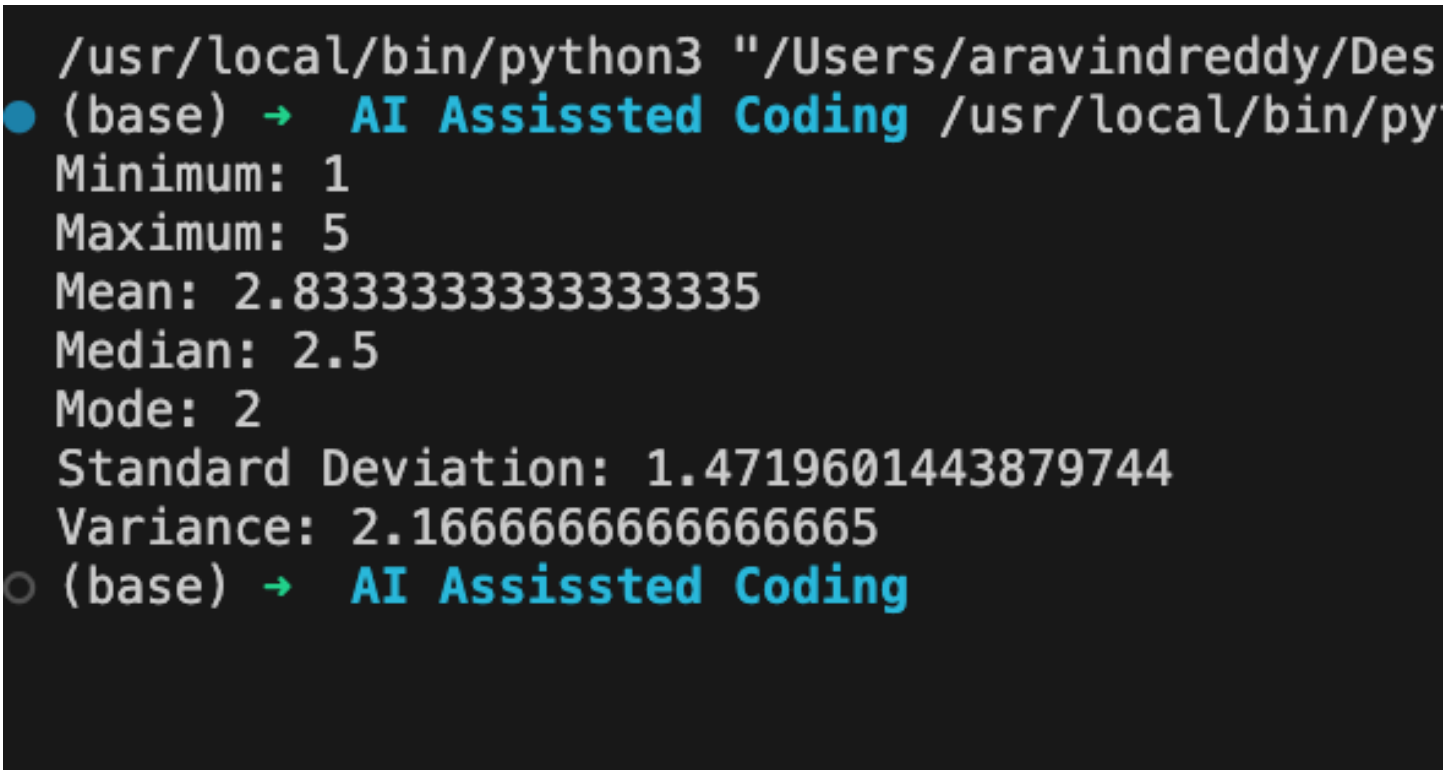
their relevance and correctness.

Code :



```python
# Task 3:

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

# Minimum, Maximum
# Mean, Median, Mode
# Standard Deviation, Variance

import statistics

def statistical_operations(tuple_num):
    minimum = min(tuple_num)
    maximum = max(tuple_num)
    mean = statistics.mean(tuple_num)
    median = statistics.median(tuple_num)
    try:
        mode = statistics.mode(tuple_num)
    except statistics.StatisticsError:
        mode = "No unique mode"
    std_dev = statistics.stdev(tuple_num)
    variance = statistics.variance(tuple_num)

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

# Example usage
data = (1, 2, 2, 3, 4, 5)
results = statistical_operations(data)
for key, value in results.items():
    print(f"{key}: {value}")

# Output:

# Minimum: 1
# Maximum: 5
# Mean: 2.8333333333333335
# Median: 2.5
# Mode: 2
# Standard Deviation: 1.4719601443879744
# Variance: 2.1666666666666665
```

Output :



```
/usr/local/bin/python3 "/Users/aravindreddy/Des
(base) ➜  AI Assissted Coding /usr/local/bin/py
Minimum: 1
Maximum: 5
Mean: 2.8333333333333335
Median: 2.5
Mode: 2
Standard Deviation: 1.4719601443879744
Variance: 2.1666666666666665
(base) ➜  AI Assissted Coding
```

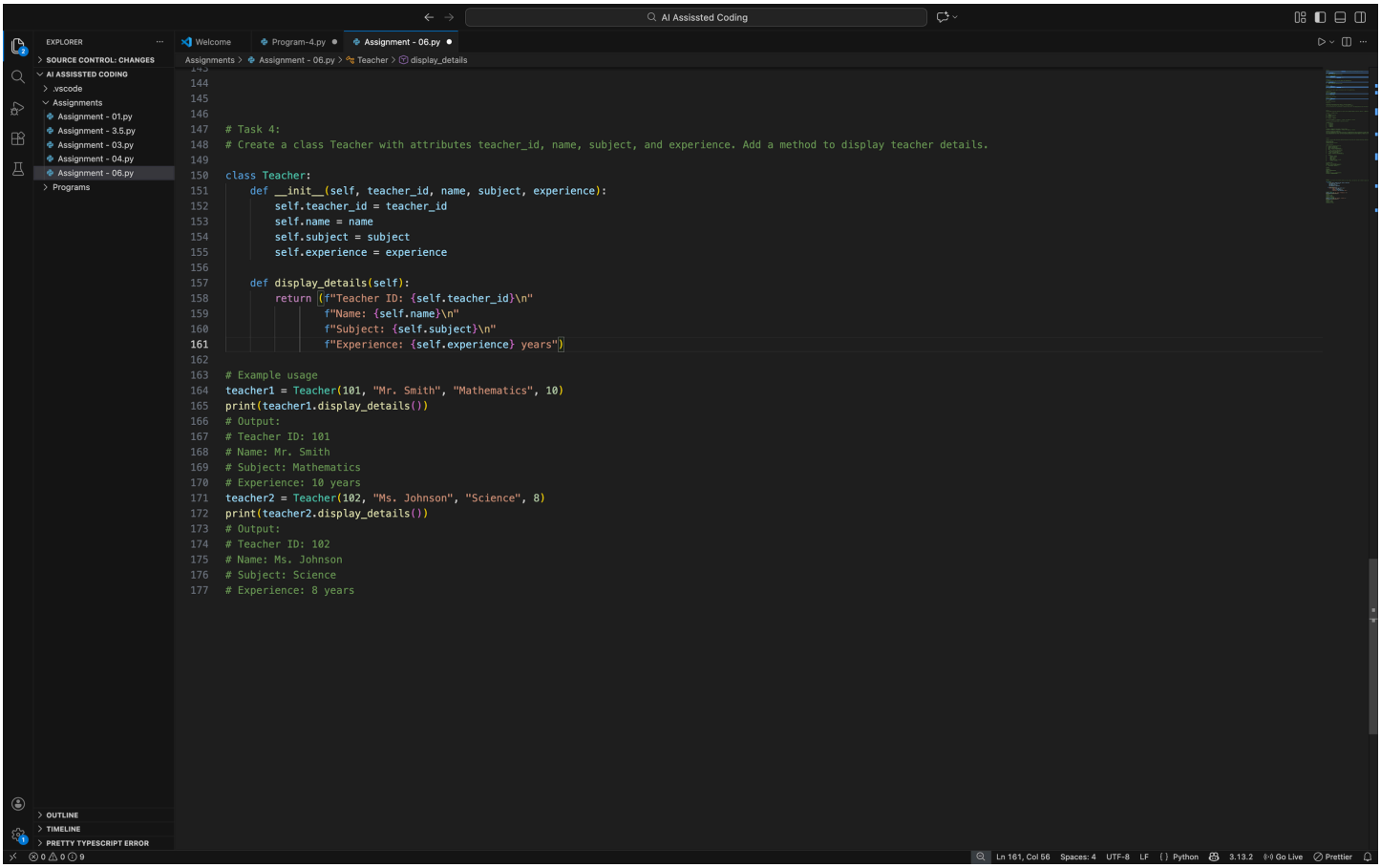
Task 4: Teacher Profile

• **Prompt :** Create a class Teacher with attributes teacher_id, name,

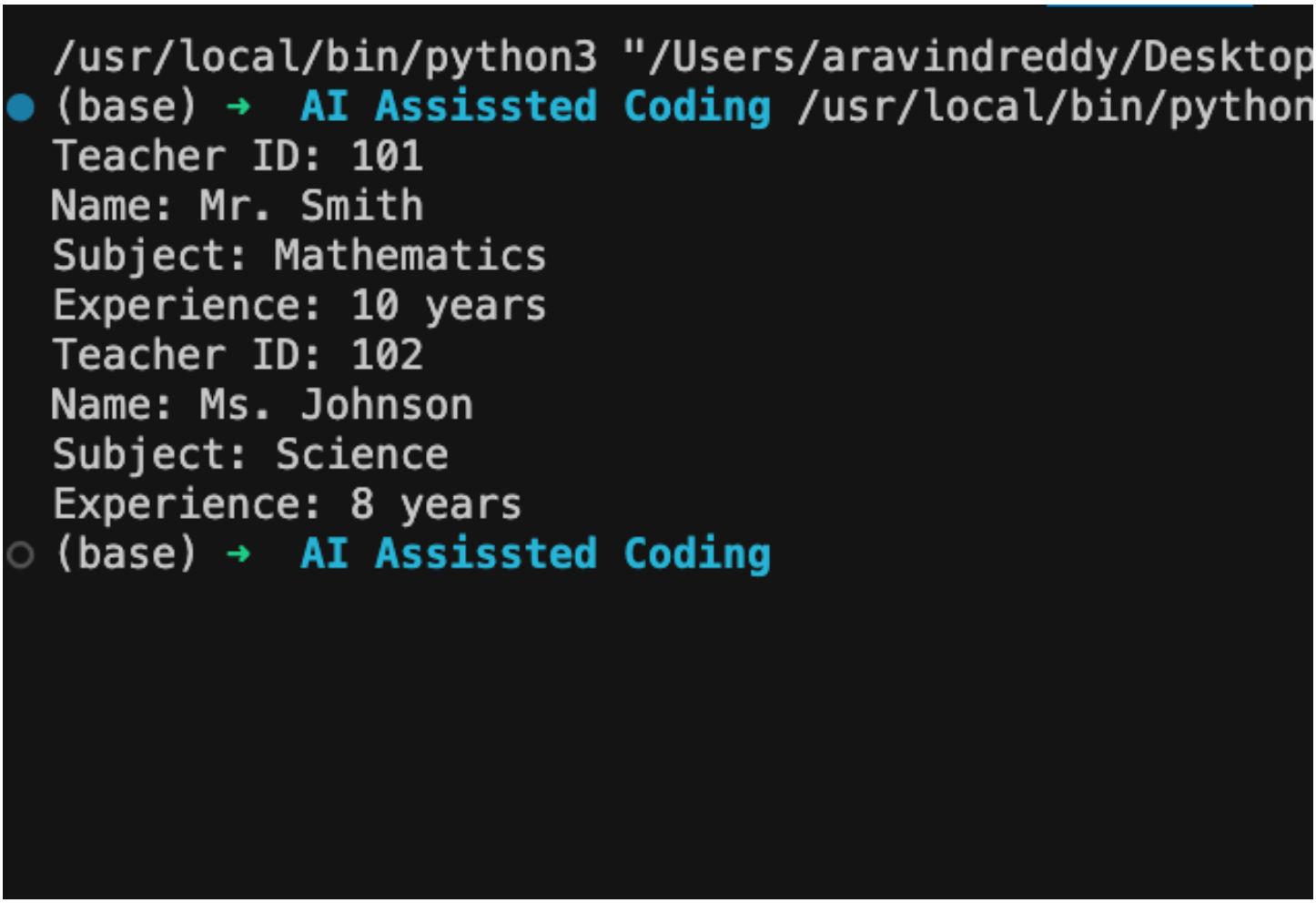
subject, and experience. Add a method to display teacher details.

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

Code :



Output :



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

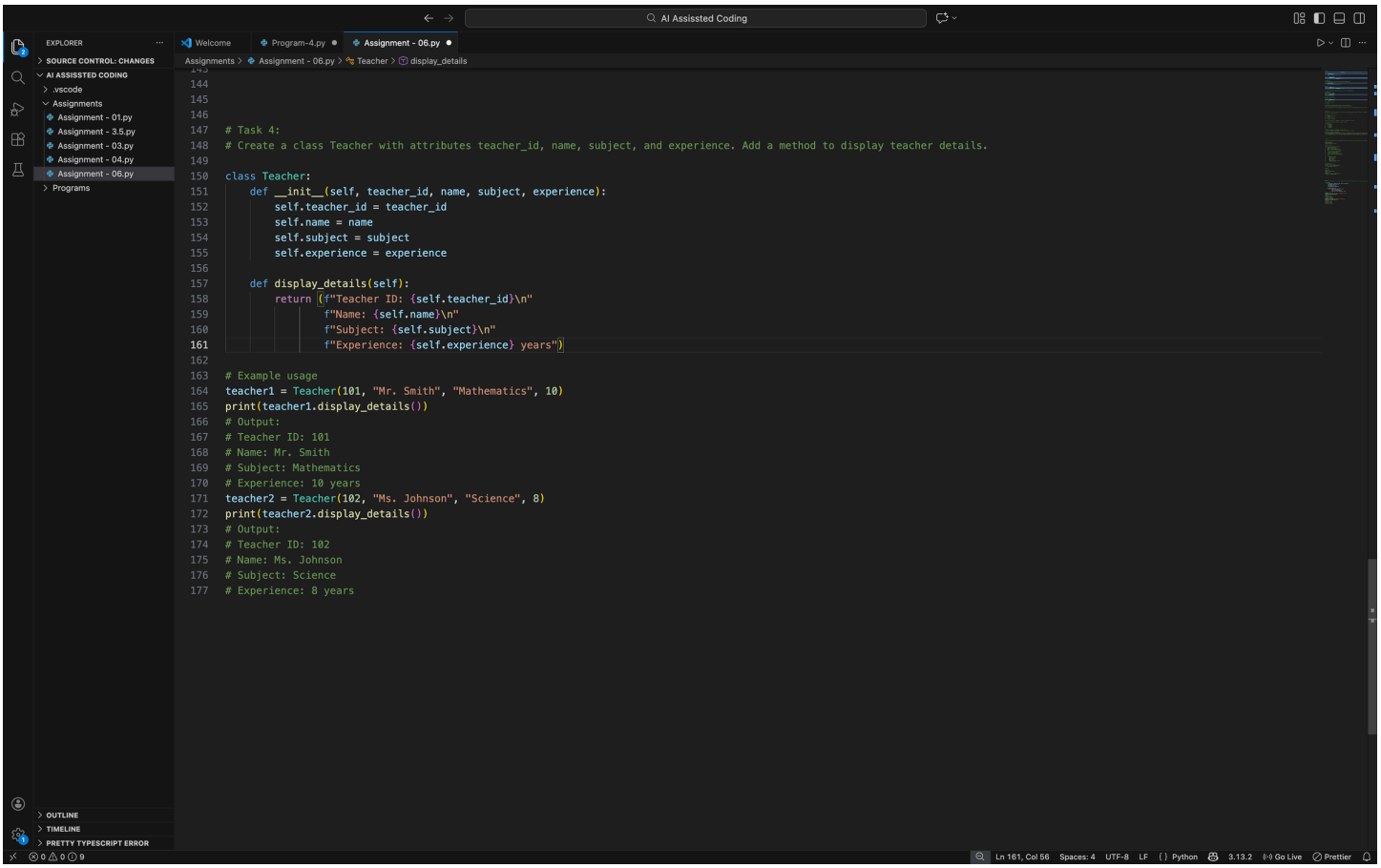
- **The function must ensure the mobile number:**
  - Starts with 6, 7, 8, or 9
  - Contains exactly 10 digits

Expected Output

- A valid Python function that performs all required validations

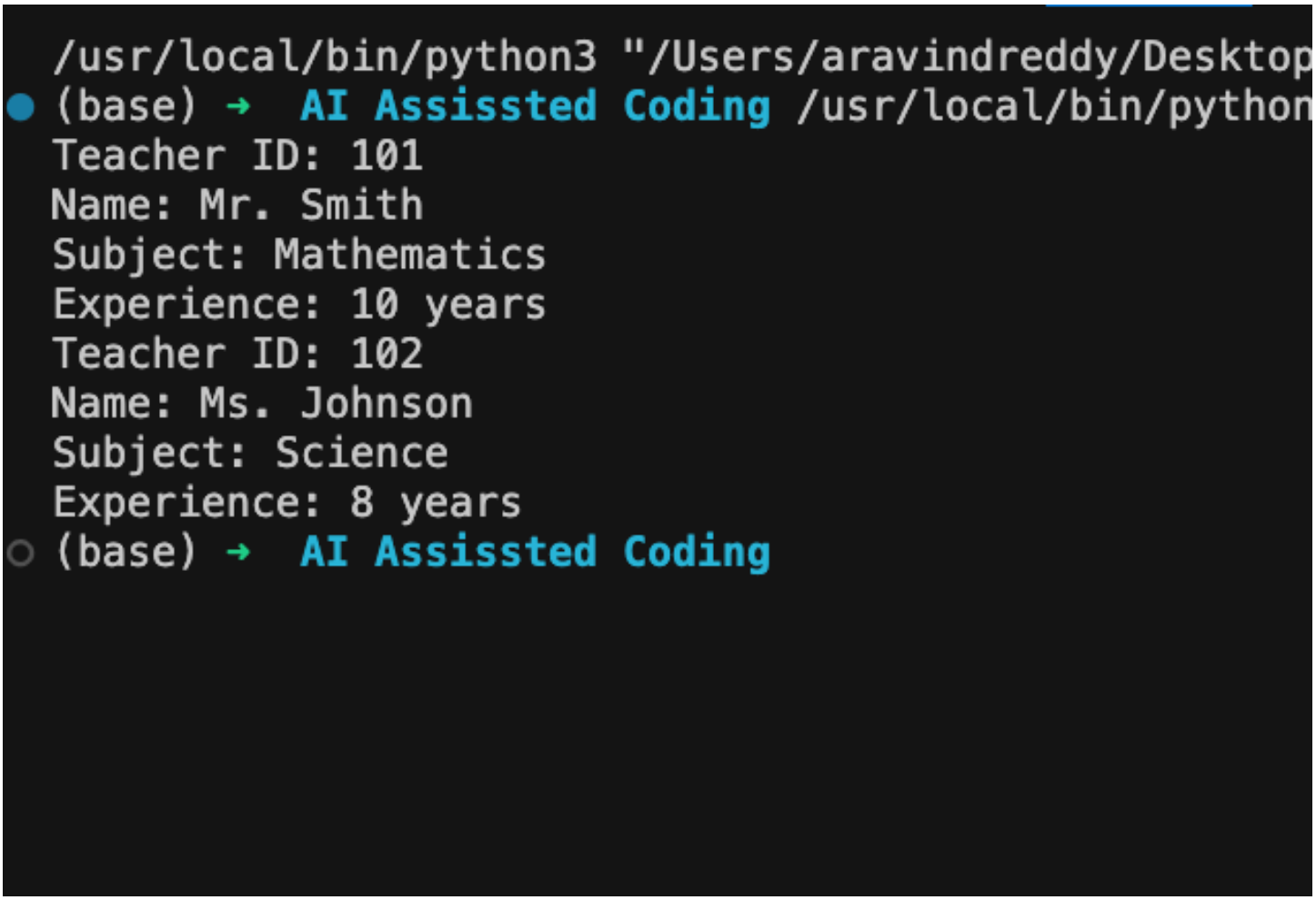
  without using any input-output examples in the prompt.

Code :



```python
# Task 4:
# 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}\n"
                f"Name: {self.name}\n"
                f"Subject: {self.subject}\n"
                f"Experience: {self.experience} years")

# Example usage
teacher1 = Teacher(101, "Mr. Smith", "Mathematics", 10)
print(teacher1.display_details())
# Output:
# Teacher ID: 101
# Name: Mr. Smith
# Subject: Mathematics
# Experience: 10 years
teacher2 = Teacher(102, "Ms. Johnson", "Science", 8)
print(teacher2.display_details())
# Output:
# Teacher ID: 102
# Name: Ms. Johnson
# Subject: Science
# Experience: 8 years
```

Output :



```
/usr/local/bin/python3 "/Users/aravindreddy/Desktop
(base) ➜  AI Assissted Coding /usr/local/bin/python
Teacher ID: 101
Name: Mr. Smith
Subject: Mathematics
Experience: 10 years
Teacher ID: 102
Name: Ms. Johnson
Subject: Science
Experience: 8 years
(base) ➜  AI Assissted Coding
```

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.

Code :

```
# Task 6:

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

# use only 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).

def is_armstrong(num):
    digits = [int(d) for d in str(num)]
    power = len(digits)
    sum_of_powers = sum(d ** power for d in digits)
    return sum_of_powers == num

armstrong_numbers = []

for i in range(1, 1001):
    if is_armstrong(i):
        armstrong_numbers.append(i)

print("Armstrong numbers between 1 and 1000 are:", armstrong_numbers)
# Output: Armstrong numbers between 1 and 1000 are: [1, 2, 3, 4, 5, 6, 7, 8, 9, 153, 370, 371, 407]
# Optimized version using list comprehensions

armstrong_numbers_optimized = [num for num in range(1, 1001) if is_armstrong(num)]
print("Optimized Armstrong numbers between 1 and 1000 are:", armstrong_numbers_optimized)
# Output: Optimized Armstrong numbers between 1 and 1000 are: [1, 2, 3, 4, 5, 6, 7, 8, 9, 153, 370, 371, 407]
```

Output:

```
Armstrong numbers between 1 and 1000 are: [1, 2, 3, 4, 5, 6, 7, 8, 9, 153, 370, 371, 407]
Optimized Armstrong numbers between 1 and 1000 are: [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 optimised 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.
- Optimised version using cycle detection with explanation.

Code :

```python
# Task 7:

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

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

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

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

def 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_numbers = []

for i in range(1, 501):
    if is_happy(i):
        happy_numbers.append(i)
print("Happy numbers between 1 and 500 are:", happy_numbers)

# Output: Happy numbers between 1 and 500 are: [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, 1
# Optimized version using a set to detect cycles

def is_happy_optimized(num):
    seen = set()
    while num != 1:
        if num in seen:
            return False
        seen.add(num)
        num = sum(int(digit) ** 2 for digit in str(num))
    return True

happy_numbers_optimized = [num for num in range(1, 501) if is_happy_optimized(num)]
print("Optimized Happy numbers between 1 and 500 are:", happy_numbers_optimized)

# Output: Optimized Happy numbers between 1 and 500 are: [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, 1
# The optimized version improves cycle detection by using a set to track previously seen numbers, preventing infinite loops and enhancing efficiency.
```

Output:

```
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, 2
, 367, 368, 376, 379, 383, 386, 391, 392, 397, 404, 409, 440, 446, 464, 469, 478, 487, 490, 496]
e: [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, 2
, 362, 365, 367, 368, 376, 379, 383, 386, 391, 392, 397, 404, 409, 440, 446, 464, 469, 478, 487, 490, 496]
```

Justification :

The optimised version improves cycle detection by using a set to track previously seen numbers, preventing infinite loops and enhancing efficiency.

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.

Code :

```python
# Task 8:

'''
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).

'''

import math

def is_strong(num):
    sum_of_factorials = 0
    for digit in str(num):
        sum_of_factorials += math.factorial(int(digit))
    return sum_of_factorials == num

strong_numbers = []

for i in range(1, 1001):
    if is_strong(i):
        strong_numbers.append(i)

print("Strong numbers between 1 and 1000 are:", strong_numbers)
# Output: Strong numbers between 1 and 1000 are: [1, 2, 145, 40585]
# Optimized version with precomputed digit factorials
digit_factorials = {str(i): math.factorial(i) for i in range(10)}

def is_strong_optimized(num):
    sum_of_factorials = sum(digit_factorials[digit] for digit in str(num))
    return sum_of_factorials == num
strong_numbers_optimized = [num for num in range(1, 1001) if is_strong_optimized(num)]
print("Optimized Strong numbers between 1 and 1000 are:", strong_numbers_optimized)
# Output: Optimized Strong numbers between 1 and 1000 are: [1, 2, 145, 40585]
# The optimized version reduces redundant calculations by precomputing the factorials of digits 0–9, improving efficiency.
```

Output :

```
Strong numbers between 1 and 1000 are: [1, 2, 145]
Optimized Strong numbers between 1 and 1000 are: [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:**
  - Full Name
  - Branch
  - SGPA

Expected Output

A reusable Python function that correctly navigates and extracts values

from nested dictionaries based on the provided examples

Code :

```
create a function that parses a nested dictionary representing student information.
Requirements :
• The function should extract and return:
   Full Name, Branch, SGPA

Example 1:
{
    "name": "John Doe",
    "age": 20,
    "branch": "Computer Science",
    "academic": {
        "sgpa": 8.5,
        "year": 3
    }
}

Example 2:
{
    "name": "Jane Smith",
    "age": 22,
    "academic": {
        "sgpa": 9.1,
        "year": 4
    }
}
'''

def extract_student_info(student_dict):
    full_name = student_dict.get("name", "N/A")
    branch = student_dict.get("branch", "N/A")
    sgpa = student_dict.get("academic", {}).get("sgpa", "N/A")

    return {
        "Full Name": full_name,
        "Branch": branch,
        "SGPA": sgpa
    }

# Example usage
student_info = {
    "name": "John Doe",
    "age": 20,
    "branch": "Computer Science",
    "academic": {
        "sgpa": 8.5,
        "year": 3
    }
}

extracted_info = extract_student_info(student_info)
print(extracted_info)
```

Output :

```
{'Full Name': 'John Doe', 'Branch': 'Computer Science', 'SGPA': 8.5}
```

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).
  - 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).

Expected Output #12:

- Python program that lists Perfect Numbers in the given range.
- Optimized version with explanation.

Code :

```
# Task 10:

'''
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)
'''
import math

def is_perfect(num):
    if num < 2:
        return False
    sum_of_divisors = 1  # 1 is a proper divisor of all integers > 1
    for i in range(2, int(math.sqrt(num)) + 1):
        if num % i == 0:
            sum_of_divisors += i
            if i != num // i:
                sum_of_divisors += num // i
    return sum_of_divisors == num
perfect_numbers = []
for i in range(1, 1001):
    if is_perfect(i):
        perfect_numbers.append(i)
print("Perfect numbers between 1 and 1000 are:", perfect_numbers)
# Output: Perfect numbers between 1 and 1000 are: [6, 28
# Optimized version using divisor check only up to √n
def is_perfect_optimized(num):
    if num < 2:
        return False
    sum_of_divisors = 1
    for i in range(2, int(math.sqrt(num)) + 1):
        if num % i == 0:
            sum_of_divisors += i
            if i != num // i:
                sum_of_divisors += num // i
    return sum_of_divisors == num
perfect_numbers_optimized = [num for num in range(1, 1001) if is_perfect_optimized(num)]
print("Optimized Perfect numbers between 1 and 1000 are:", perfect_numbers_optimized)
# Output: Optimized Perfect numbers between 1 and 1000 are: [6, 28, 496]
# The optimized version improves efficiency by limiting the divisor check to the square root of the number, reducing the number of iterations need
# Optimized version using divisor check only up to √n
```

Output :

```
Perfect numbers between 1 and 1000 are: [6, 28, 496]
Optimized Perfect numbers between 1 and 1000 are: [6, 28, 496]
```

Justification :

The optimized version improves efficiency by limiting the divisor check to the square root of the number, reducing the number of iterations needed to find proper divisors., 496]

Optimized version using divisor check only up to √n