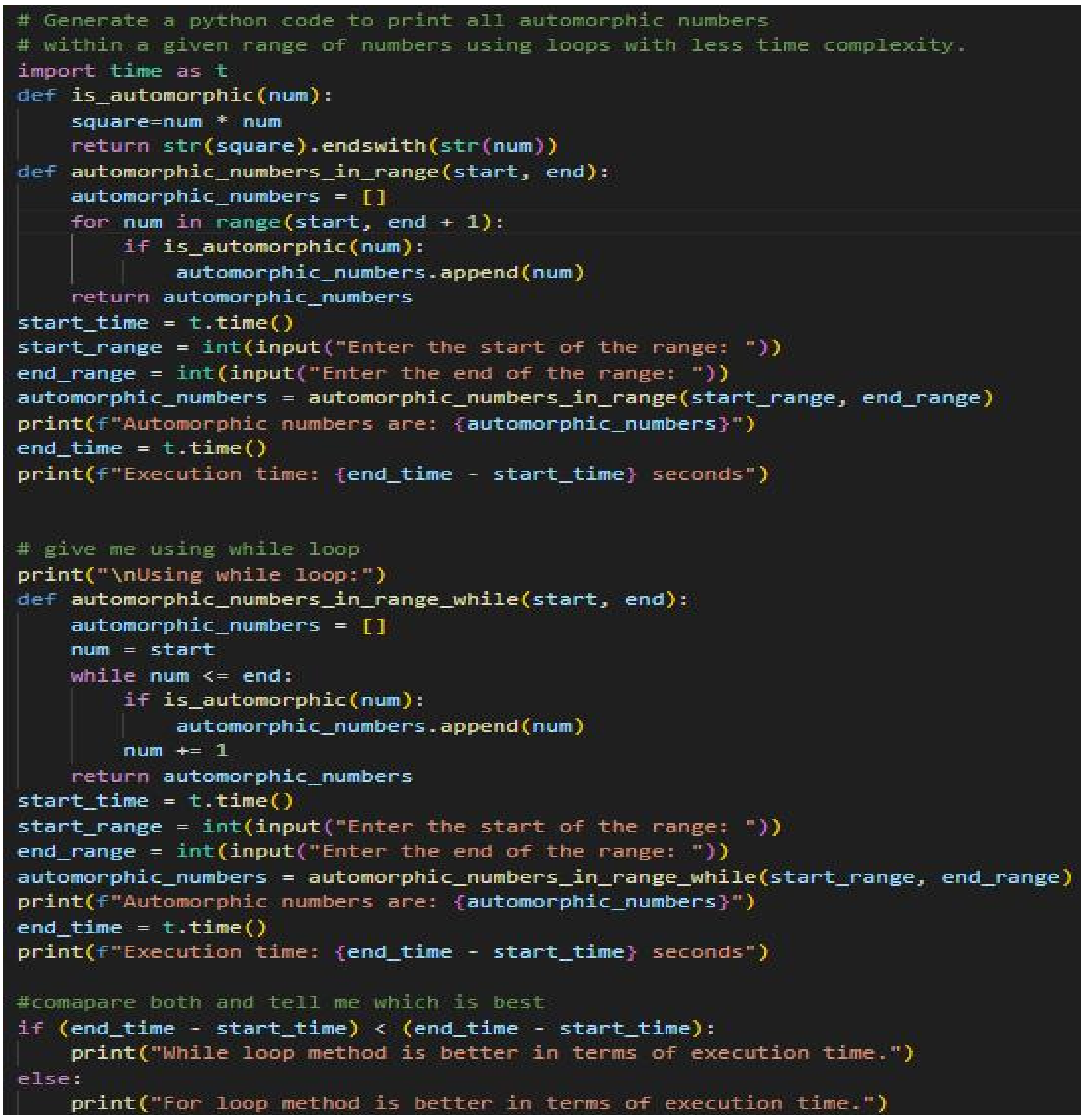
AI ASSISTED CODING
Assignment-6.3

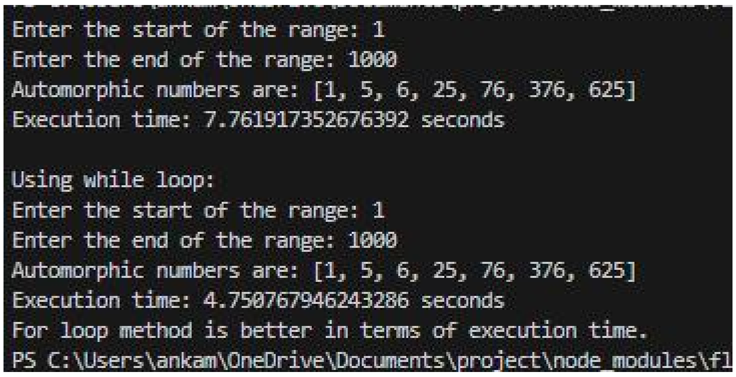
A. Niteesh Kumar
2303A52341
B-32

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:



```python
# Generate a python code to print all automorphic numbers
# within a given range of numbers using loops with less time complexity.
import time as t
def is_automorphic(num):
    square=num * num
    return str(square).endswith(str(num))
def automorphic_numbers_in_range(start, end):
    automorphic_numbers = []
    for num in range(start, end + 1):
        if is_automorphic(num):
            automorphic_numbers.append(num)
    return automorphic_numbers
start_time = t.time()
start_range = int(input("Enter the start of the range: "))
end_range = int(input("Enter the end of the range: "))
automorphic_numbers = automorphic_numbers_in_range(start_range, end_range)
print(f"Automorphic numbers are: {automorphic_numbers}")
end_time = t.time()
print(f"Execution time: {end_time - start_time} seconds")


# give me using while loop
print("\nUsing while loop:")
def automorphic_numbers_in_range_while(start, end):
    automorphic_numbers = []
    num = start
    while num <= end:
        if is_automorphic(num):
            automorphic_numbers.append(num)
        num += 1
    return automorphic_numbers
start_time = t.time()
start_range = int(input("Enter the start of the range: "))
end_range = int(input("Enter the end of the range: "))
automorphic_numbers = automorphic_numbers_in_range_while(start_range, end_range)
print(f"Automorphic numbers are: {automorphic_numbers}")
end_time = t.time()
print(f"Execution time: {end_time - start_time} seconds")

#comapare both and tell me which is best
if (end_time - start_time) < (end_time - start_time):
    print("While loop method is better in terms of execution time.")
else:
    print("For loop method is better in terms of execution time.")
```

OUTPUT:



```
Enter the start of the range: 1
Enter the end of the range: 1000
Automorphic numbers are: [1, 5, 6, 25, 76, 376, 625]
Execution time: 7.761917352676392 seconds

Using while loop:
Enter the start of the range: 1
Enter the end of the range: 1000
Automorphic numbers are: [1, 5, 6, 25, 76, 376, 625]
Execution time: 4.750767946243286 seconds
For loop method is better in terms of execution time.
PS C:\Users\ankam\OneDrive\Documents\project\node_modules\fl
```

Explanations: The program finds Automorphic numbers by checking whether the square of a number ends with the number itself using loop logic.

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.

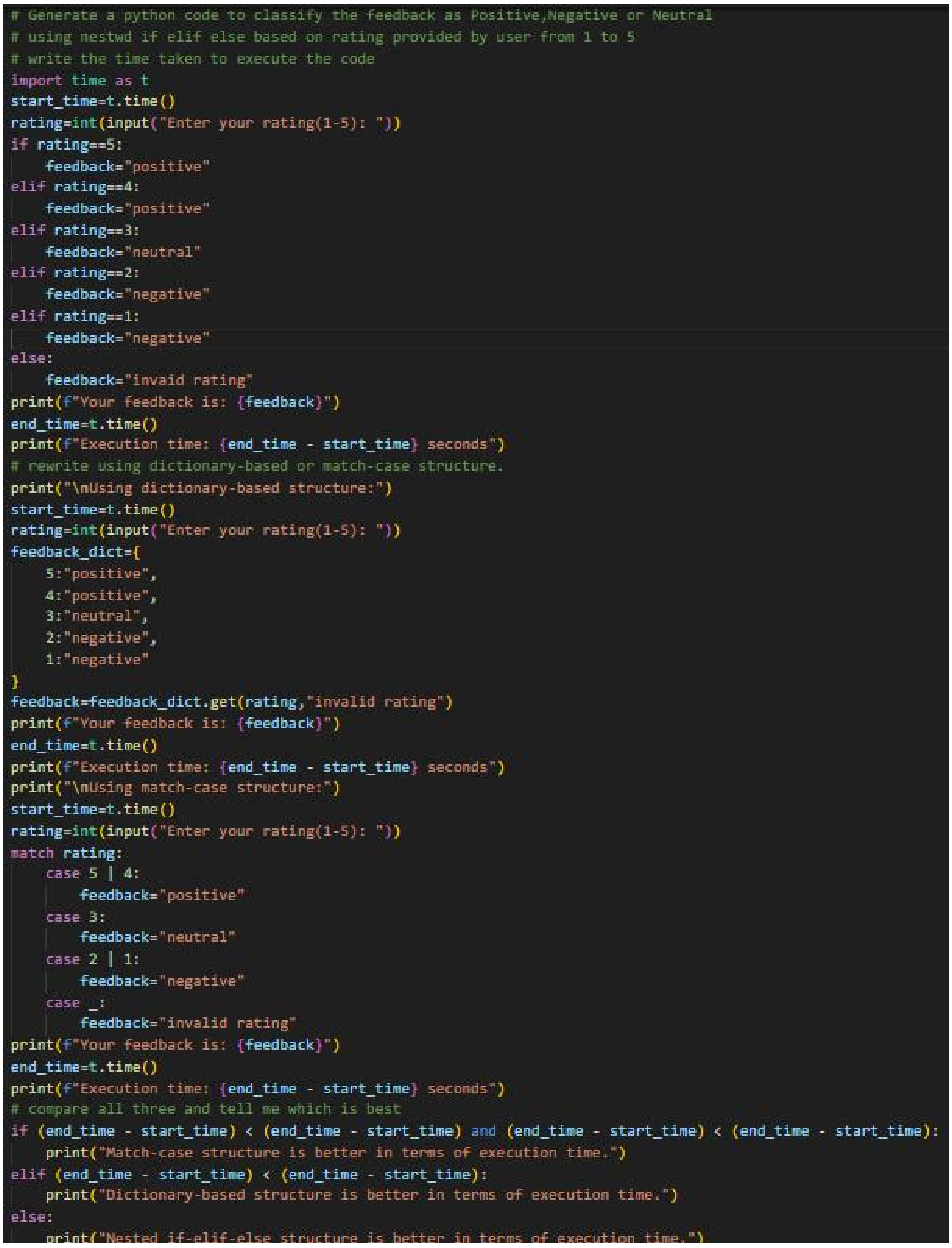
o Analyze correctness and readability.

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

Expected Output #2:

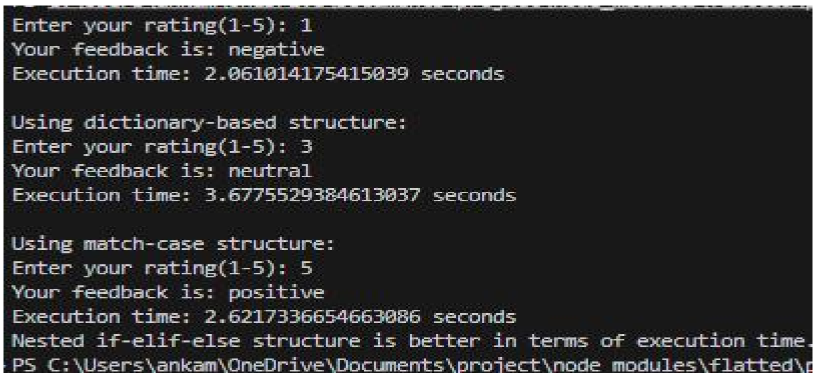
• Feedback classification function with explanation and an alternative approach.

PROMPT & CODE:



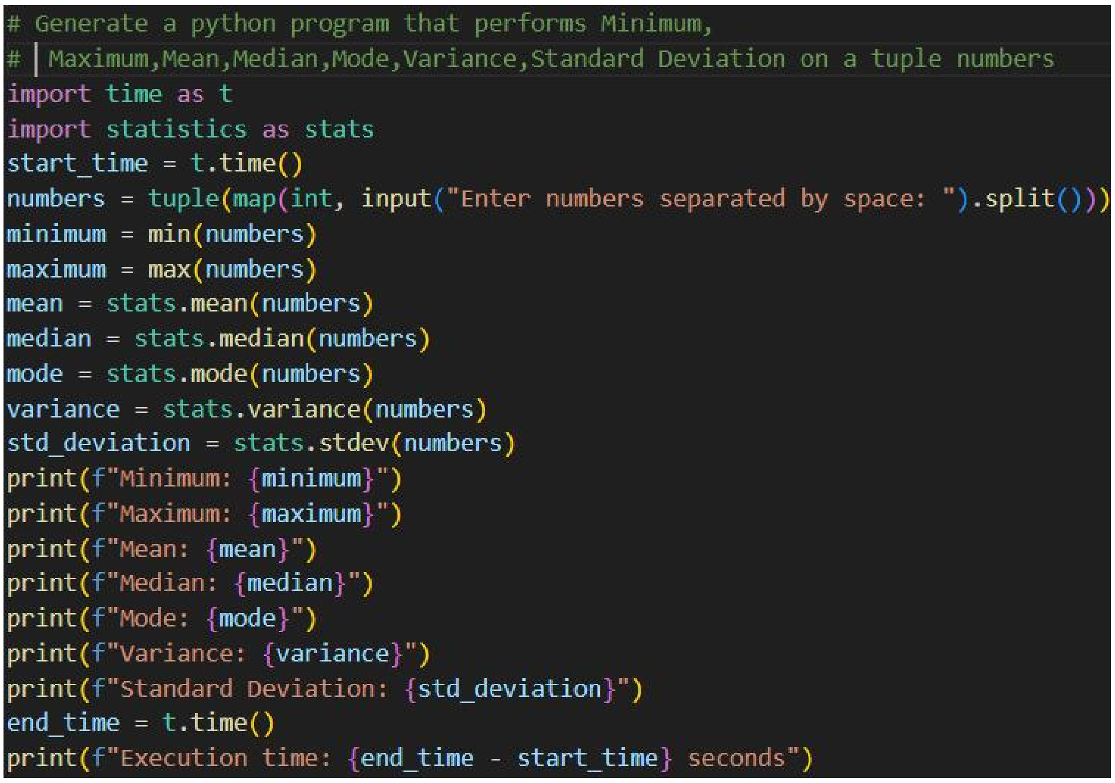
```python
# Generate a python code to classify the feedback as Positive,Negative or Neutral
# using nestwd if elif else based on rating provided by user from 1 to 5
# write the time taken to execute the code
import time as t
start_time=t.time()
rating=int(input("Enter your rating(1-5): "))
if rating==5:
    feedback="positive"
elif rating==4:
    feedback="positive"
elif rating==3:
    feedback="neutral"
elif rating==2:
    feedback="negative"
elif rating==1:
    feedback="negative"
else:
    feedback="invaid rating"
print(f"Your feedback is: {feedback}")
end_time=t.time()
print(f"Execution time: {end_time - start_time} seconds")
# rewrite using dictionary-based or match-case structure.
print("\nUsing dictionary-based structure:")
start_time=t.time()
rating=int(input("Enter your rating(1-5): "))
feedback_dict={
    5:"positive",
    4:"positive",
    3:"neutral",
    2:"negative",
    1:"negative"
}
feedback=feedback_dict.get(rating,"invalid rating")
print(f"Your feedback is: {feedback}")
end_time=t.time()
print(f"Execution time: {end_time - start_time} seconds")
print("\nUsing match-case structure:")
start_time=t.time()
rating=int(input("Enter your rating(1-5): "))
match rating:
    case 5 | 4:
        feedback="positive"
    case 3:
        feedback="neutral"
    case 2 | 1:
        feedback="negative"
    case _:
        feedback="invalid rating"
print(f"Your feedback is: {feedback}")
end_time=t.time()
print(f"Execution time: {end_time - start_time} seconds")
# compare all three and tell me which is best
if (end_time - start_time) < (end_time - start_time) and (end_time - start_time) < (end_time - start_time):
    print("Match-case structure is better in terms of execution time.")
elif (end_time - start_time) < (end_time - start_time):
    print("Dictionary-based structure is better in terms of execution time.")
else:
    print("Nested if-elif-else structure is better in terms of execution time.")
```

OUTPUT:

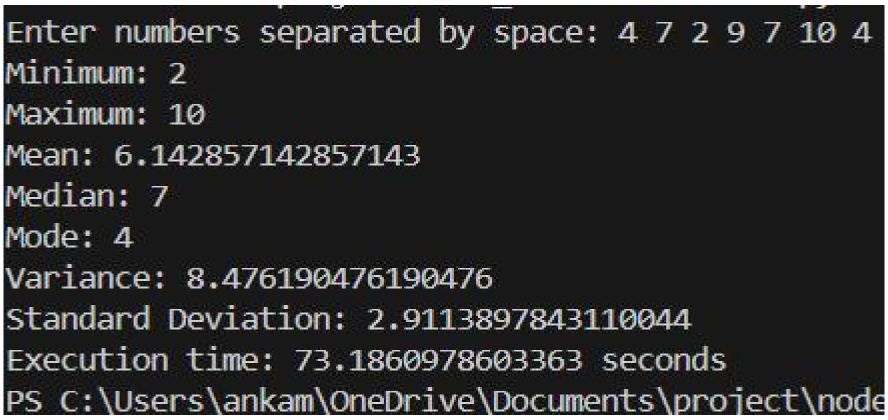


```
Enter your rating(1-5): 1
Your feedback is: negative
Execution time: 2.061014175415039 seconds

Using dictionary-based structure:
Enter your rating(1-5): 3
Your feedback is: neutral
Execution time: 3.6775529384613037 seconds

Using match-case structure:
Enter your rating(1-5): 5
Your feedback is: positive
Execution time: 2.6217336654663086 seconds
Nested if-elif-else structure is better in terms of execution time.
PS C:\Users\ankam\OneDrive\Documents\project\node_modules\flatted\
```

Explanation: The feedback classification uses conditional statements to correctly label ratings as Negative, Neutral, or Positive based on given 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
PROMPT & CODE:


```python
# Generate a python program that performs Minimum,
# Maximum,Mean,Median,Mode,Variance,Standard Deviation on a tuple numbers
import time as t
import statistics as stats
start_time = t.time()
numbers = tuple(map(int, input("Enter numbers separated by space: ").split()))
minimum = min(numbers)
maximum = max(numbers)
mean = stats.mean(numbers)
median = stats.median(numbers)
mode = stats.mode(numbers)
variance = stats.variance(numbers)
std_deviation = stats.stdev(numbers)
print(f"Minimum: {minimum}")
print(f"Maximum: {maximum}")
print(f"Mean: {mean}")
print(f"Median: {median}")
print(f"Mode: {mode}")
print(f"Variance: {variance}")
print(f"Standard Deviation: {std_deviation}")
end_time = t.time()
print(f"Execution time: {end_time - start_time} seconds")
```

OUTPUT:


```
Enter numbers separated by space: 4 7 2 9 7 10 4
Minimum: 2
Maximum: 10
Mean: 6.142857142857143
Median: 7
Mode: 4
Variance: 8.476190476190476
Standard Deviation: 2.9113897843110044
Execution time: 73.1860978603363 seconds
PS C:\Users\ankam\OneDrive\Documents\project\node
```

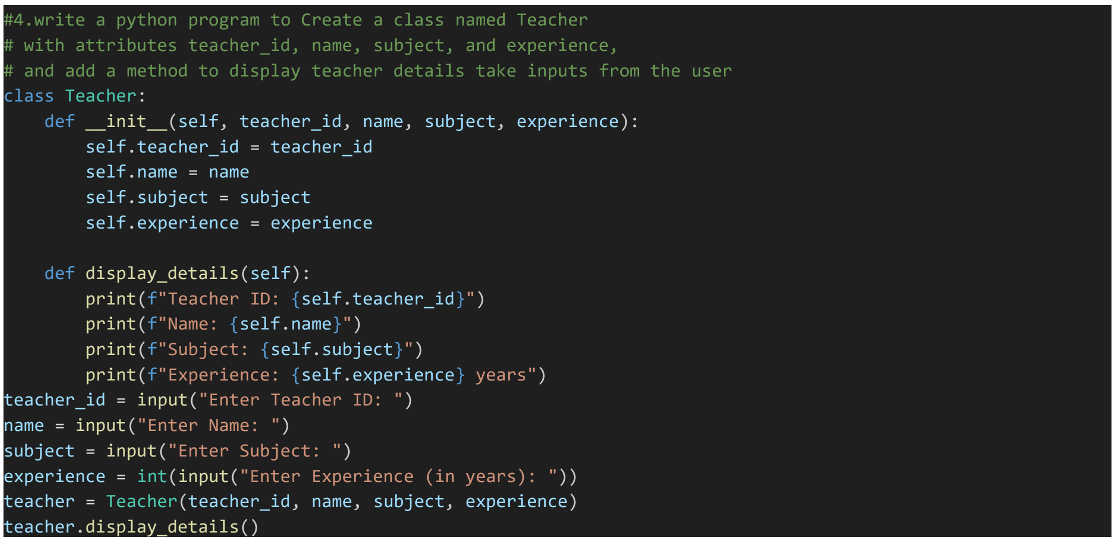
Explanation: The function uses Python built-in statistics methods to compute minimum, maximum, mean, median, mode, variance, and standard deviation accurately.

Task 4: Teacher Profile

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

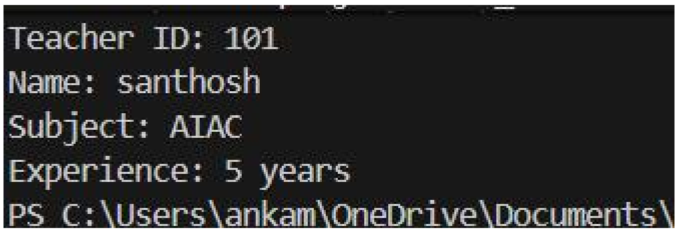
• Expected Output: Class with initializer, method, and object creation.

PROMPT & CODE:



```python
#4.write a python program to Create a class named Teacher
# with attributes teacher_id, name, subject, and experience,
# and add a method to display teacher details take inputs from the user
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")
teacher_id = input("Enter Teacher ID: ")
name = input("Enter Name: ")
subject = input("Enter Subject: ")
experience = int(input("Enter Experience (in years): "))
teacher = Teacher(teacher_id, name, subject, experience)
teacher.display_details()
```

OUTPUT:



```
Teacher ID: 101
Name: santhosh
Subject: AIAC
Experience: 5 years
PS C:\Users\ankam\OneDrive\Documents\
```

Explanation: The Teacher class demonstrates object oriented programming by initializing attributes through a constructor and displaying details using a class method.

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:

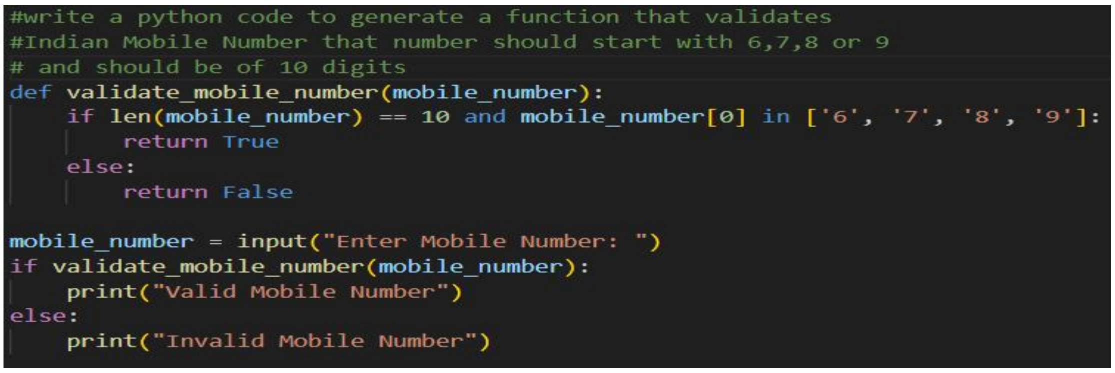
o Starts with 6, 7, 8, or 9

o Contains exactly 10 digits

Expected Output

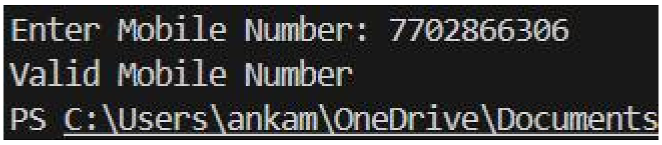
• A valid Python function that performs all required validations without using any input-output examples in the prompt.

PROMPT & CODE:



```python
#write a python code to generate a function that validates
#Indian Mobile Number that number should start with 6,7,8 or 9
# and should be of 10 digits
def validate_mobile_number(mobile_number):
    if len(mobile_number) == 10 and mobile_number[0] in ['6', '7', '8', '9']:
        return True
    else:
        return False

mobile_number = input("Enter Mobile Number: ")
if validate_mobile_number(mobile_number):
    print("Valid Mobile Number")
else:
    print("Invalid Mobile Number")
```

OUTPUT:



```
Enter Mobile Number: 7702866306
Valid Mobile Number
PS C:\Users\ankam\OneDrive\Documents
```

Explanation: The function validates an Indian mobile number by checking that it has exactly ten digits and starts with six, seven, eight, or nine.

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.).

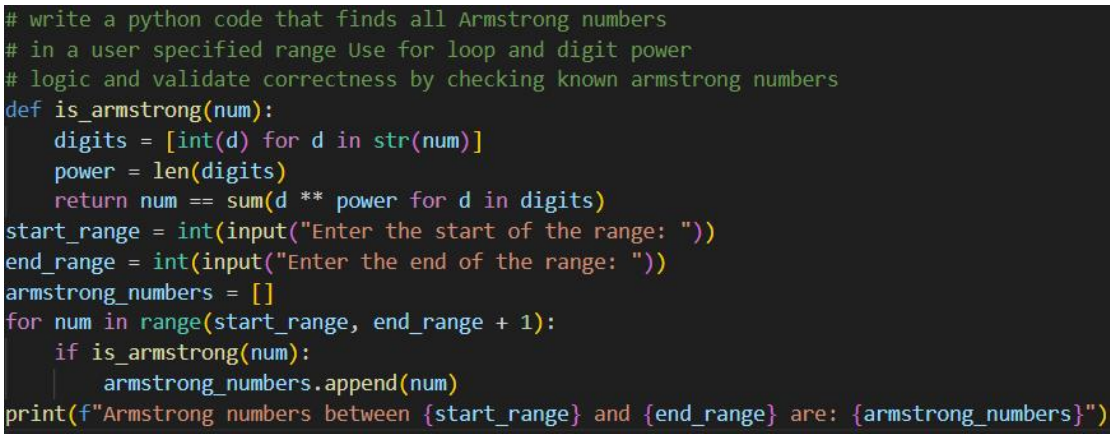
• Ask AI to regenerate an optimized version (using list comprehensions).

Expected Output #7:

• Python program listing Armstrong numbers in the range.

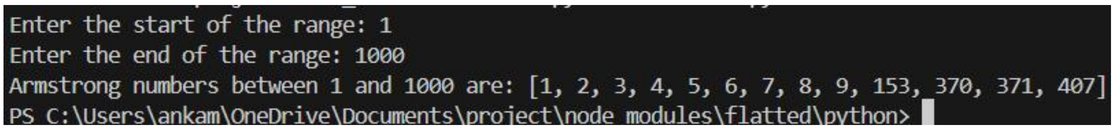
• Optimized version with explanation.

PROMPT & CODE:



```python
# write a python code that finds all Armstrong numbers
# in a user specified range Use for loop and digit power
# logic and validate correctness by checking known armstrong numbers
def is_armstrong(num):
    digits = [int(d) for d in str(num)]
    power = len(digits)
    return num == sum(d ** power for d in digits)
start_range = int(input("Enter the start of the range: "))
end_range = int(input("Enter the end of the range: "))
armstrong_numbers = []
for num in range(start_range, end_range + 1):
    if is_armstrong(num):
        armstrong_numbers.append(num)
print(f"Armstrong numbers between {start_range} and {end_range} are: {armstrong_numbers}")
```

OUTPUT:



```
Enter the start of the range: 1
Enter the end of the range: 1000
Armstrong numbers between 1 and 1000 are: [1, 2, 3, 4, 5, 6, 7, 8, 9, 153, 370, 371, 407]
PS C:\Users\ankam\OneDrive\Documents\project\node_modules\flatted\python>
```

Explanation: The program identifies Armstrong numbers by comparing each number with the sum of its digits raised to the power of total digits.

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).

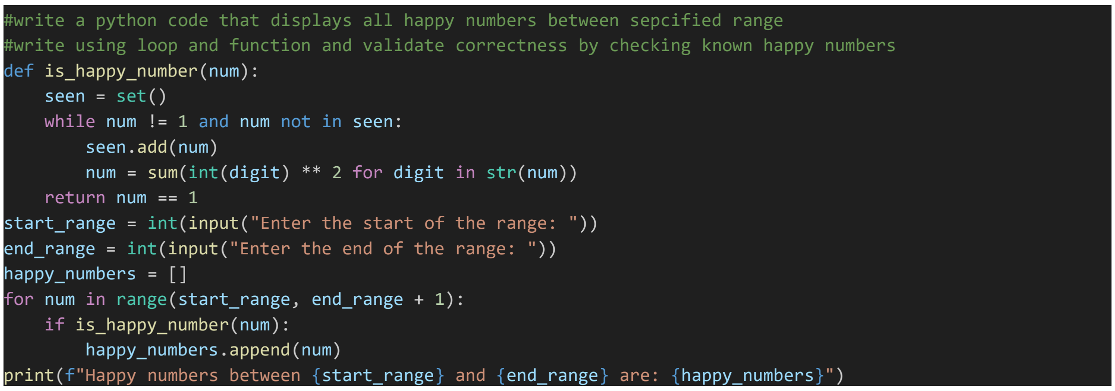
Instructions:

• 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…).

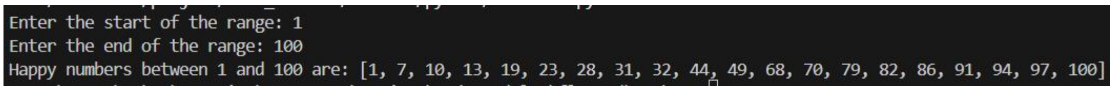
• Ask AI to regenerate an optimized version (e.g., by using a set to detect cycles instead of infinite loops).

Expected Output #8:
• Python program that prints all Happy Numbers within a range.
• Optimized version using cycle detection with explanation.
PROMPT & CODE:



```python
#write a python code that displays all happy numbers between sepcified range
#write using loop and function and validate correctness by checking known happy numbers
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
start_range = int(input("Enter the start of the range: "))
end_range = int(input("Enter the end of the range: "))
happy_numbers = []
for num in range(start_range, end_range + 1):
    if is_happy_number(num):
        happy_numbers.append(num)
print(f"Happy numbers between {start_range} and {end_range} are: {happy_numbers}")
```

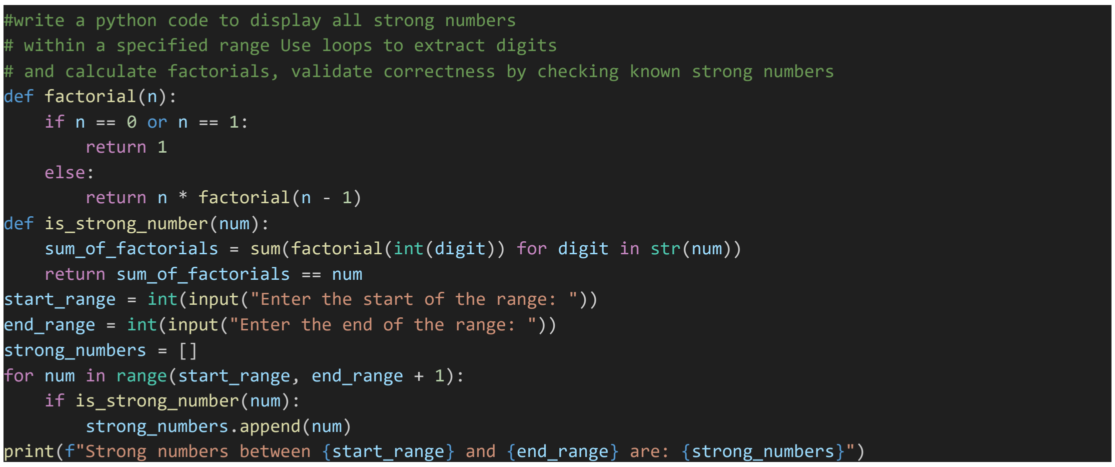
OUTPUT:



```
Enter the start of the range: 1
Enter the end of the range: 100
Happy numbers between 1 and 100 are: [1, 7, 10, 13, 19, 23, 28, 31, 32, 44, 49, 68, 70, 79, 82, 86, 91, 94, 97, 100]
```

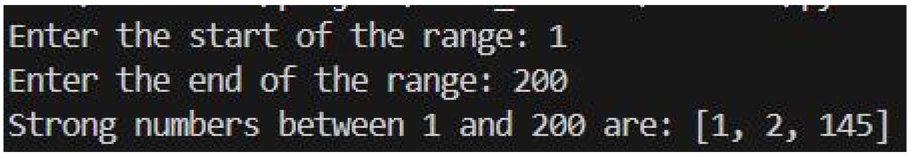
Explanation: Happy numbers are detected by repeatedly summing the squares of digits and using a set to prevent infinite loops.

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 & CODE:



```python
#write a python code to display all strong numbers
# within a specified range Use loops to extract digits
# and calculate factorials, validate correctness by checking known strong numbers
def factorial(n):
    if n == 0 or n == 1:
        return 1
    else:
        return n * factorial(n - 1)
def is_strong_number(num):
    sum_of_factorials = sum(factorial(int(digit)) for digit in str(num))
    return sum_of_factorials == num
start_range = int(input("Enter the start of the range: "))
end_range = int(input("Enter the end of the range: "))
strong_numbers = []
for num in range(start_range, end_range + 1):
    if is_strong_number(num):
        strong_numbers.append(num)
print(f"Strong numbers between {start_range} and {end_range} are: {strong_numbers}")
```

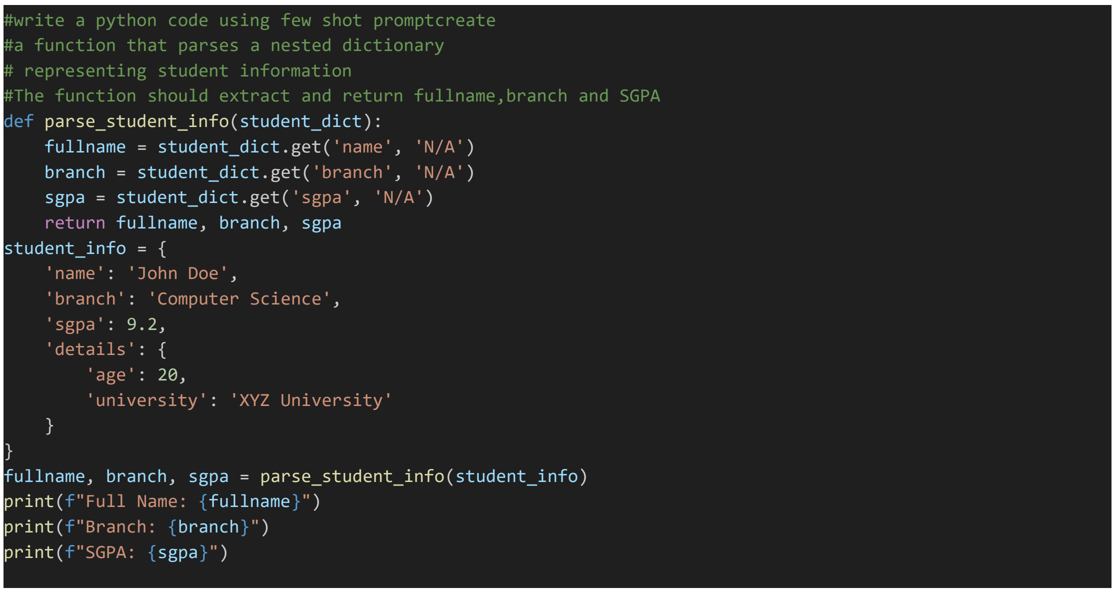
OUTPUT:



```
Enter the start of the range: 1
Enter the end of the range: 200
Strong numbers between 1 and 200 are: [1, 2, 145]
```

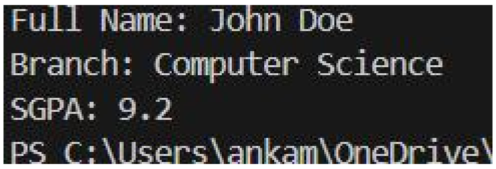
Explanation: The function checks whether a number equals the sum of factorials of its digits to identify Strong numbers.

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 & CODE:



```python
#write a python code using few shot promptcreate
#a function that parses a nested dictionary
# representing student information
#The function should extract and return fullname,branch and SGPA
def parse_student_info(student_dict):
    fullname = student_dict.get('name', 'N/A')
    branch = student_dict.get('branch', 'N/A')
    sgpa = student_dict.get('sgpa', 'N/A')
    return fullname, branch, sgpa
student_info = {
    'name': 'John Doe',
    'branch': 'Computer Science',
    'sgpa': 9.2,
    'details': {
        'age': 20,
        'university': 'XYZ University'
    }
}
fullname, branch, sgpa = parse_student_info(student_info)
print(f"Full Name: {fullname}")
print(f"Branch: {branch}")
print(f"SGPA: {sgpa}")
```

OUTPUT:



```
Full Name: John Doe
Branch: Computer Science
SGPA: 9.2
PS C:\Users\ankam\OneDrive\
```

Explanation: The function navigates a nested dictionary structure to correctly extract student full name, branch, and SGPA.

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…).

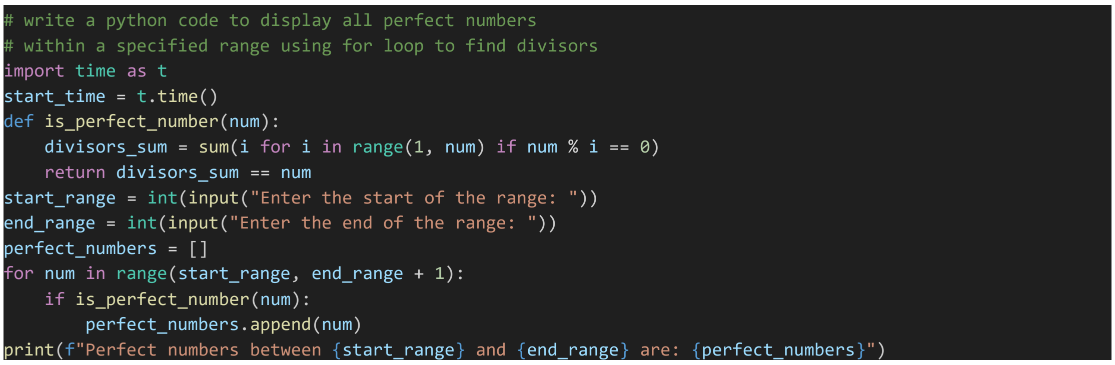
• Ask AI to regenerate an optimized version (using divisor check only up to √n).

Expected Output #12:

• Python program that lists Perfect Numbers in the given range.

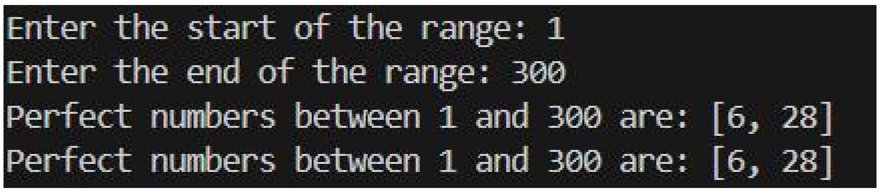
• Optimized version with explanation.

PROMPT & CODE:



```python
# write a python code to display all perfect numbers
# within a specified range using for loop to find divisors
import time as t
start_time = t.time()
def is_perfect_number(num):
    divisors_sum = sum(i for i in range(1, num) if num % i == 0)
    return divisors_sum == num
start_range = int(input("Enter the start of the range: "))
end_range = int(input("Enter the end of the range: "))
perfect_numbers = []
for num in range(start_range, end_range + 1):
    if is_perfect_number(num):
        perfect_numbers.append(num)
print(f"Perfect numbers between {start_range} and {end_range} are: {perfect_numbers}")
```

OUTPUT:



```
Enter the start of the range: 1
Enter the end of the range: 300
Perfect numbers between 1 and 300 are: [6, 28]
Perfect numbers between 1 and 300 are: [6, 28]
```

Explanation: Perfect numbers are identified by summing proper divisors efficiently by checking only up to the square root of the number.