AI Assistant Coding

Assignment-6.3

Name:K.Trinay Prasad

HT No:2303A52070

Batch:32

Q1. 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

Code:

```
'''
1.Write a python function that generates automorphic numbers in a given range from 1 to 1000 using for loop.
'''
import time
def is_automorphic(num):
    square = num ** 2
    return str(square).endswith(str(num))
def generate_automorphic_numbers(start, end):
    automorphic_numbers = []
    for num in range(start, end + 1):
        if is_automorphic(num):
            automorphic_numbers.append(num)
    return automorphic_numbers
t1 = time.time()
automorphic_numbers = generate_automorphic_numbers(1, 1000)
t2 = time.time()
print(f"Automorphic numbers between 1 and 1000: {automorphic_numbers}")
print(f"Time taken: {t2 - t1} seconds")
```

```
'''
2.Write a python function that generates automorphic numbers in a given range from 1 to 1000 using while loop.
'''
def generate_automorphic_numbers_while(start, end):
    automorphic_numbers = []
    num = start
    while num <= end:
        if is_automorphic(num):
            automorphic_numbers.append(num)
        num += 1
    return automorphic_numbers
t3 = time.time()
automorphic_numbers_while = generate_automorphic_numbers_while(1, 1000)
t4 = time.time()
print(f"Automorphic numbers between 1 and 1000 (while loop): {automorphic_numbers_while}")
print(f"Time taken: {t4 - t3} seconds")
```

Output:

```
PS D:\AI Assicoding> & C:/Python313/python.exe "d:/AI Assicoding/Ass6.3.py"
Automorphic numbers between 1 and 1000: [1, 5, 6, 25, 76, 376, 625]
Time taken: 0.0002777576446533203 seconds
Automorphic numbers between 1 and 1000 (while loop): [1, 5, 6, 25, 76, 376, 625]
Time taken: 0.00030231475830078125 seconds
PS D:\AI Assicoding>
```

Explanation:

For loop is taking less time then while loop. Use for loop when you know when to stop if not use while loop.

Q2. 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.

Code:

```python
import time
#Write a python function that classify online shopping feedback into positive,negative and neutral based
#on numerical rating (1-5) using nested if else statements.
def classify_feedback(rating):
    if rating >= 1 and rating <= 5:
        if rating == 1:
            return "Negative"
        elif rating == 2:
            return "Negative"
        elif rating == 3:
            return "Neutral"
        elif rating == 4:
            return "Positive"
        else: # rating == 5
            return "Positive"
    else:
        return "Invalid Rating"
# Example usage
t1 = time.time()
print(classify_feedback(5))  # Output: Positive
print("Time taken: ", time.time() - t1)#output: Positive
print(classify_feedback(3))  # Output: Neutral
print(classify_feedback(1))  # Output: Negative
print(classify_feedback(6))  # Output: Invalid Rating

print("\n")
#Write a python function that classify online shopping feedback into positive,negative and neutral based on numerical rating (1-5) using dictionary or match case.
def classify_feedback_dict(rating):
    feedback_dict = {
        1: "Negative",
        2: "Negative",
        3: "Neutral",
        4: "Positive",
        5: "Positive"
    }
    return feedback_dict.get(rating, "Invalid Rating")
# Example usage
t2 = time.time()
print(classify_feedback_dict(5))  # Output: Positive
print("Time taken: ", time.time() - t2) #output: Positive
print(classify_feedback_dict(3))  # Output: Neutral
print(classify_feedback_dict(1))  # Output: Negative
print(classify_feedback_dict(6))  # Output: Invalid Rating
```

Output:

```
PS D:\AI Assicoding> & C:/Python313/python.exe "d:/AI Assicoding/Ass6.3.py"
Positive
Time taken:  0.00013136863708496094
Neutral
Negative
Invalid Rating


Positive
Time taken:  3.266345336914406e-05
Neutral
Negative
Invalid Rating
PS D:\AI Assicoding>
```

Explanation:

Condition ensures correctness of the code, Dictionary improves readability.

Q4. 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.

Code:

```
'''
Create a class with Teacher with attributes teacher_id, name,subject and experence 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):
        print(f"Teacher ID: {self.teacher_id}")
        print(f"Name: {self.name}")
        print(f"Subject: {self.subject}")
        print(f"Experience: {self.experience} years")
# Example usage
teacher1 = Teacher(1, "Alice Smith", "Mathematics", 10)
teacher1.display_details()
teacher2 = Teacher(2, "Bob Johnson", "Science", 8)
teacher2.display_details()
teacher3 = Teacher(3, "Charlie Brown", "History", 5)
teacher3.display_details()
teacher4 = Teacher(4, "Diana Prince", "English", 12)
teacher4.display_details()
```

Output:

```
PS D:\AI Assicoding> & C:/Python313/python.exe "d:/AI Assicoding/Ass6.3.py"
Teacher ID: 1
Name: Alice Smith
Subject: Mathematics
Experience: 10 years
Teacher ID: 2
Name: Bob Johnson
Subject: Science
Experience: 8 years
Teacher ID: 3
Name: Charlie Brown
Subject: History
Experience: 5 years
Teacher ID: 4
Name: Diana Prince
Subject: English
Experience: 12 years
PS D:\AI Assicoding>
```

Q5. 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.

Code:

```
'''
Write a python function name vaild indian number and check number starts with 6 7 8 9
and length is 10 digits.
'''
def valid_indian_number(number):
    str_number = str(number)
    if len(str_number) == 10 and str_number[0] in '6789' and str_number.isdigit():
        return True
    return False
print(valid_indian_number(9876543210))  # True
print(valid_indian_number(1234567890))  # False
```

Output:

```
PS D:\AI Assicoding> & C:/Python313/python.exe "d:/AI Assicoding/Ass6.3.py"
True
False
PS D:\AI Assicoding>
```

Q6. Task Description #6 (Loops – Armstrong Numbers in a Range)

Task: Write a function using AI that finds all Armstrong numbers in a userspecified

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:

```
'''
Write a python function that finds armstrong numbers in a given range from 1 to 1000
 using for loop and digit power logic.
'''
def find_armstrong_numbers():
    armstrong_numbers = []
    for num in range(1, 1001):
        num_str = str(num)
        num_digits = len(num_str)
        num_sum = sum(int(digit) ** num_digits for digit in num_str)
        if num == num_sum:
            armstrong_numbers.append(num)
    return armstrong_numbers
armstrong_numbers = find_armstrong_numbers()
print(armstrong_numbers)

'''
Write a python function that finds armstrong numbers from 1 to 1000
using while loop and digit power logic.optimized version using list comprehension.
'''
def find_armstrong_numbers_optimized():
    return [num for num in range(1, 1001) if sum(int(digit) ** len(str(num)) for digit in str(num)) == num]
armstrong_numbers = find_armstrong_numbers_optimized()
print(armstrong_numbers)
```

Output:

```
PS D:\AI Assicoding> & C:/Python313/python.exe "d:/AI Assicoding/Ass6.3.py"
[1, 2, 3, 4, 5, 6, 7, 8, 9, 153, 370, 371, 407]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 153, 370, 371, 407]
PS D:\AI Assicoding>
```

Explanation:

Using for loop it is easy to understand and debug the code best for bigginer.

List comprehension is minimal and hard to debug.

Q7. 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.

Code:

```
'''
Write a python to display all happy numbers in range of 1 to 500.
'''
import time
def is_happy_number(num):
    seen_numbers = set()
    while num != 1 and num not in seen_numbers:
        seen_numbers.add(num)
        num = sum(int(digit) ** 2 for digit in str(num))
    return num == 1
def find_happy_numbers(start, end):
    happy_numbers = []
    for num in range(start, end + 1):
        if is_happy_number(num):
            happy_numbers.append(num)
    return happy_numbers
t1 = time.time()
happy_numbers = find_happy_numbers(1, 500)
t2 = time.time()
print(f"Happy numbers between 1 and 500: {happy_numbers}")
print(f"Time taken: {t2 - t1} seconds")
```

```
'''
Write a python function to display all happy numbers in range of 1 to 500
optimize the above code using cycle detection.
'''
import time
def is_happy_number_optimized(num):
    def get_next(n):
        return sum(int(digit) ** 2 for digit in str(n))
    slow = num
    fast = get_next(num)
    while fast != 1 and slow != fast:
        slow = get_next(slow)
        fast = get_next(get_next(fast))
    return fast == 1
def find_happy_numbers_optimized(start, end):
    happy_numbers = []
    for num in range(start, end + 1):
        if is_happy_number_optimized(num):
            happy_numbers.append(num)
    return happy_numbers
t3 = time.time()
happy_numbers_optimized = find_happy_numbers_optimized(1, 500)
t4 = time.time()
print(f"Happy numbers between 1 and 500 (optimized): {happy_numbers_optimized}")
print(f"Time taken: {t4 - t3} seconds")
```

Output:

```
PS D:\AI Assicoding> & C:/Python313/python.exe "d:/AI Assicoding/Ass6.3.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]
Time taken: 0.003149747848510742 seconds
Happy numbers between 1 and 500 (optimized): [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, 4
04, 409, 440, 446, 464, 469, 478, 487, 490, 496]
Time taken: 0.005888462066650391 seconds
PS D:\AI Assicoding>
```

Explanation:

The optimized cycle detection method is the best choice because it reduces memory usage, improves efficiency, and scales better while still producing correct results. The set-based approach is easier for beginners, but the optimized method is more professional and performance-oriented.

Q8. 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:

```
'''
Write a python function to display strong numbers with in a given range.
'''
import time
def factorial(n):
    if n == 0 or n == 1:
        return 1
    else:
        return n * factorial(n - 1)
def is_strong_number(num):
    return num == sum(factorial(int(digit)) for digit in str(num))
def find_strong_numbers(start, end):
    strong_numbers = []
    for num in range(start, end + 1):
        if is_strong_number(num):
            strong_numbers.append(num)
    return strong_numbers
t1 = time.time()
strong_numbers = find_strong_numbers(1, 500)
t2 = time.time()
print(f"Strong numbers between 1 and 500: {strong_numbers}")
print(f"Time taken: {t2 - t1} seconds")
```

```
'''
Write a python function to display strong numbers with in a given range optimized version precomputing factorials.
'''
def precompute_factorials(max_digits):
    factorials = [1] * (max_digits + 1)
    for i in range(1, max_digits + 1):
        factorials[i] = factorials[i - 1] * i
    return factorials
def is_strong_number_optimized(num, factorials):
    return num == sum(factorials[int(digit)] for digit in str(num))
def find_strong_numbers_optimized(start, end, factorials):
    strong_numbers = []
    for num in range(start, end + 1):
        if is_strong_number_optimized(num, factorials):
            strong_numbers.append(num)
    return strong_numbers
factorials = precompute_factorials(9)
t3 = time.time()
strong_numbers_optimized = find_strong_numbers_optimized(1, 500, factorials)
t4 = time.time()
print(f"Strong numbers between 1 and 500 (optimized): {strong_numbers_optimized}")
print(f"Time taken: {t4 - t3} seconds")
```

Output:

```
PS D:\AI Assicoding> & C:/Python313/python.exe "d:/AI Assicoding/Ass6.3.py"
Strong numbers between 1 and 500: [1, 2, 145]
Time taken: 0.000698089599609375 seconds
Strong numbers between 1 and 500 (optimized): [1, 2, 145]
Time taken: 0.00038552284240722656 seconds
PS D:\AI Assicoding>
```

Explanation:

The basic approach identifies strong numbers correctly but recalculates factorials repeatedly, leading to unnecessary computation.

The optimized method precomputes factorials of digits (0–9), reducing repeated calculations and improving efficiency.

Hence, the optimized approach is more suitable for larger ranges due to better performance and scalability.

Q9. 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

Code:

```
'''
Write a python function that parses a nested dictionary of student
information requirements:
Full Name
Branch
SGPA
example input:
"full_name": "John Doe",
"branch": "Computer Science",
"sgpa": 8.5
"full_name": "Jane Smith",
"branch": "Mechanical Engineering",
"sgpa": 9.2
'''
```

```python
def parse_student_info(student):
    parsed_info=[]
    for student_id,info in student.items():
        full_name=info.get("full_name")
        branch=info.get("branch")
        sgpa=info.get("sgpa")
        parsed_info.append({
            "Student ID":student_id,
            "Full Name":full_name,
            "Branch":branch,
            "SGPA":sgpa
        })
    return parsed_info
# Example usage
students = {
    "student_1": {
        "full_name": "John Doe",
        "branch": "Computer Science",
        "sgpa": 8.5
    },
    "student_2": {
        "full_name": "Jane Smith",
        "branch": "Mechanical Engineering",
        "sgpa": 9.2
    }
}
parsed_students = parse_student_info(students)
for student in parsed_students:
    print(student)
```

Output:

```
PS D:\AI Assicoding> & C:/Python313/python.exe "d:/AI Assicoding/Ass6.3.py"
{'Student ID': 'student_1', 'Full Name': 'John Doe', 'Branch': 'Computer Science', 'SGPA': 8.5}
{'Student ID': 'student_2', 'Full Name': 'Jane Smith', 'Branch': 'Mechanical Engineering', 'SGPA': 9.2}
PS D:\AI Assicoding>
```

Q10. 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)

Code:

```
'''
Write a python function that display all perfect numbers in a given range from 1 to 1000.
'''
import time
def is_perfect_number(num):
    sum_of_divisors = sum(i for i in range(1, num) if num % i == 0)
    return sum_of_divisors == num
def find_perfect_numbers(start, end):
    perfect_numbers = []
    for num in range(start, end + 1):
        if is_perfect_number(num):
            perfect_numbers.append(num)
    return perfect_numbers
# Example usage
if __name__ == "__main__":
    t1 = time.time()
    perfect_numbers = find_perfect_numbers(1, 1000)
    t2 = time.time()
    print(f"Perfect numbers between 1 and 1000: {perfect_numbers}")
    print(f"Time taken: {t2 - t1} seconds")
```

```
'''
Write a python function that display all perfect numbers in a given range from 1 to 1000
optimized using divisor check only up to the square root of the number.
'''
import time
import math
def is_perfect_number_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
def find_perfect_numbers_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
if __name__ == "__main__":
    t3 = time.time()
    perfect_numbers_optimized = find_perfect_numbers_optimized(1, 1000)
    t4 = time.time()
    print(f"Perfect numbers between 1 and 1000 (optimized): {perfect_numbers_optimized}")
    print(f"Time taken: {t4 - t3} seconds")
```

Output:

```
PS D:\AI Assicoding> & C:/Python313/python.exe "d:/AI Assicoding/Ass6.3.py"
Perfect numbers between 1 and 1000: [6, 28, 496]
Time taken: 0.0167391300201416 seconds
Perfect numbers between 1 and 1000 (optimized): [6, 28, 496]
Time taken: 0.0011546611785888672 seconds
PS D:\AI Assicoding>
```