AI Assisted Coding

Assignment Number:6.1

Name: A.Deepthi

Ht.No: 2303A52390

Batch: 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:

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

Prompt:

```
'''
list the automorphic numbers between 1 to 1000 using for loop and
while loop then print their time complexities and compare both the
implementations return the efficient approach
'''
```

Code:

```python
# Using for loop
import time
start_time_for = time.time()
automorphic_numbers_for = []
for num in range(1, 1001):
    square = num ** 2
    if str(square).endswith(str(num)):
        automorphic_numbers_for.append(num)
end_time_for = time.time()
time_complexity_for = end_time_for - start_time_for
print("Automorphic numbers using for loop:", automorphic_numbers_for)
print("Time taken using for loop:", time_complexity_for, "seconds")
# Using while loop
start_time_while = time.time()
automorphic_numbers_while = []
num = 1
while num <= 1000:
    square = num ** 2
    if str(square).endswith(str(num)):
        automorphic_numbers_while.append(num)
    num += 1
end_time_while = time.time()
time_complexity_while = end_time_while - start_time_while
print("Automorphic numbers using while loop:", automorphic_numbers_while)
print("Time taken using while loop:", time_complexity_while, "seconds")
# Comparison of both implementations
if time_complexity_for < time_complexity_while:
    print("For loop is more efficient.")
elif time_complexity_for > time_complexity_while:
    print("While loop is more efficient.")
else:
    print("Both loops have the same efficiency.")
```

Output:

```
Automorphic numbers using for loop: [1, 5, 6, 25, 76, 376, 625]
Time taken using for loop: 0.0004477500915527344 seconds
Automorphic numbers using while loop: [1, 5, 6, 25, 76, 376, 625]
Time taken using while loop: 0.0005235671997070312 seconds
For loop is more efficient.
```

Explanation:

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:

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

Prompt:

```
'''
Classify online shopping feedback as Positive,Negative and Neutral
based on their numerical ratings from 1 to 5 using if -elif-else statements after that rewrite code using
dictionary based structure and analyze correctness and readability of both approaches
and return the best readability approach by comparing both
'''
```

Code:

```python
feedbacks = [
    {"text": "Great product, very satisfied!", "rating": 5},
    {"text": "The product is okay, nothing special.", "rating": 3},
    {"text": "Very disappointed with the quality.", "rating": 1},
    {"text": "Average experience, could be better.", "rating": 3},
    {"text": "Exceeded my expectations!", "rating": 5},
    {"text": "Not worth the price.", "rating": 2},
    {"text": "It's fine for everyday use.", "rating": 4},
]
# Using if-elif-else statements
print("Using if-elif-else statements:")
for feedback in feedbacks:
    rating = feedback["rating"]
    if rating >= 4:
        label = "Positive"
    elif rating == 3:
        label = "Neutral"
    else:
        label = "Negative"
    print(f"Feedback: \"{feedback['text']}\" | Rating: {rating} | Label: {label}")
# Using dictionary-based structure
print("\nUsing dictionary-based structure:")
rating_to_label = {
    5: "Positive",
    4: "Positive",
    3: "Neutral",
    2: "Negative",
    1: "Negative"
}
for feedback in feedbacks:
    rating = feedback["rating"]
    label = rating_to_label.get(rating, "Unknown")
    print(f"Feedback: \"{feedback['text']}\" | Rating: {rating} | Label: {label}")
print("\n Analysis of correctness and readability")
print("Both approaches yield the same correctness in classification.")
print("However, the dictionary-based structure is more readable and easier to maintain,")
print("Especially if the classification criteria change or expand in the future.")
print("Therefore, the dictionary-based approach is preferred for better readability.")
print("Best readability approach: Dictionary-based structure")
```

Output:

```
Using if-elif-else statements:
Feedback: "Great product, very satisfied!" | Rating: 5 | Label: Positive
Feedback: "The product is okay, nothing special." | Rating: 3 | Label: Neutral
Feedback: "Very disappointed with the quality." | Rating: 1 | Label: Negative
Feedback: "Average experience, could be better." | Rating: 3 | Label: Neutral
Feedback: "Exceeded my expectations!" | Rating: 5 | Label: Positive
Feedback: "Not worth the price." | Rating: 2 | Label: Negative
Feedback: "It's fine for everyday use." | Rating: 4 | Label: Positive

Using dictionary-based structure:
Feedback: "Great product, very satisfied!" | Rating: 5 | Label: Positive
Feedback: "The product is okay, nothing special." | Rating: 3 | Label: Neutral
Feedback: "Very disappointed with the quality." | Rating: 1 | Label: Negative
Feedback: "Average experience, could be better." | Rating: 3 | Label: Neutral
Feedback: "Exceeded my expectations!" | Rating: 5 | Label: Positive
Feedback: "Not worth the price." | Rating: 2 | Label: Negative
Feedback: "It's fine for everyday use." | Rating: 4 | Label: Positive

 Analysis of correctness and readability
Both approaches yield the same correctness in classification.
However, the dictionary-based structure is more readable and easier to maintain,
Especially if the classification criteria change or expand in the future.
Therefore, the dictionary-based approach is preferred for better readability.
Best readability approach: Dictionary-based structure
```

Explanation:

The above python program tells us about the classification of the online shopping feedback rating between(1-5) by using conditional statements if-elif-else and also by using dictionary based structure then analysis the correctness and readability of both the approaches.

The best readability approach is dictionary based approach as it is used when the classification criteria changes or expand in future.

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:

```
'''
Create a function names statistical_operation(tuple_num) which performs statistical operations on
tuple of numbers for finding maximum number , minimum number, mean,median,mode,variance
and standard deviation
'''
```

Code:

```python
import statistics
def statistical_operation(tuple_num):
    maximum = max(tuple_num)
    minimum = min(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"
    variance = statistics.variance(tuple_num)
    std_deviation = statistics.stdev(tuple_num)

    return {
        "Maximum": maximum,
        "Minimum": minimum,
        "Mean": mean,
        "Median": median,
        "Mode": mode,
        "Variance": variance,
        "Standard Deviation": std_deviation
    }
# Example usage
numbers = (10, 20, 20, 30, 40, 50, 60)
stats = statistical_operation(numbers)
for key, value in stats.items():
    print(f"{key}: {value}")
```

Output:

```
Maximum: 60
Minimum: 10
Mean: 32.857142857142854
Median: 30
Mode: 20
Variance: 323.8095238095238
Standard Deviation: 17.994708216848746
```

Explanation:

The above program defines a function name statistics_operation() runs by importing statistics library to find min,max,mean,median,mode,variance and standard deviation.

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:

```python
class Teacher:
    def __init__(self, teacher_id, name, subject, experience):
        self.teacher_id = teacher_id
        self.name = name
        self.subject = subject
        self.experience = experience  # in years

    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(101, "Alice Smith", "Mathematics", 10)
teacher1.display_details()
teacher2 = Teacher(102, "Bob Johnson", "Science", 8)
teacher2.display_details()
```

Output:

```
Teacher ID: 101
Name: Alice Smith
Subject: Mathematics
Experience: 10 years
Teacher ID: 102
Name: Bob Johnson
Subject: Science
Experience: 8 years
```

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:

- 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.

Prompt:

```
'''
Generate a Python function to validate an Indian mobile number that starts
with six, seven, eight, or nine and contains exactly ten digits.
'''
```

Code:

```python
import re
def validate_indian_mobile_number(mobile_number):
    pattern = r'^[6-9]\d{9}$'
    if re.match(pattern, mobile_number):
        return True
    else:
        return False
# Example usage
mobile_number = input("Enter an Indian mobile number: ")
if validate_indian_mobile_number(mobile_number):
    print("Valid Indian mobile number.")
else:
    print("Invalid Indian mobile number.")
```

Output:

```
Enter an Indian mobile number: 8143992006
Valid Indian mobile number.
```

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:

```
'''
Write a function to find all Armstrong numbers in a user specified range using a for loop and digit
power logic. Regenerate an optimized version using list comprehension
'''
```

Code:

```python
def is_armstrong(number):
    digits = [int(d) for d in str(number)]
    power = len(digits)
    return sum(d ** power for d in digits) == number
# Using for loop
def find_armstrong_numbers_for_loop(start, end):
    armstrong_numbers = []
    for num in range(start, end + 1):
        if is_armstrong(num):
            armstrong_numbers.append(num)
    return armstrong_numbers
# Using list comprehension
def find_armstrong_numbers_list_comprehension(start, end):
    return [num for num in range(start, end + 1) if is_armstrong(num)]
# Example usage
start_range = 1
end_range = 1000
armstrong_numbers_for_loop = find_armstrong_numbers_for_loop(start_range, end_range)
armstrong_numbers_list_comprehension = find_armstrong_numbers_list_comprehension(start_range, end_range)
print("Armstrong numbers using for loop:", armstrong_numbers_for_loop)
print("Armstrong numbers using list comprehension:", armstrong_numbers_list_comprehension)
```

Output:

```
Armstrong numbers using for loop: [1, 2, 3, 4, 5, 6, 7, 8, 9, 153, 370, 371, 407]
Armstrong numbers using list comprehension: [1, 2, 3, 4, 5, 6, 7, 8, 9, 153, 370, 371, 407]
```

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 auser-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 that displays all Happy Numbers within a given range using loop logic,
and regenerate an optimized version using a set to detect cycles.
'''
```

Code:

```python
def is_happy_number(n):
    seen = set()
    while n != 1 and n not in seen:
        seen.add(n)
        n = sum(int(digit) ** 2 for digit in str(n))
    return n == 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
# Example usage
start_range = 1
end_range = 100
happy_numbers = find_happy_numbers(start_range, end_range)
print("Happy numbers between", start_range, "and", end_range, "are:", happy_numbers)
```

Output:

```
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 (pre compute digit factorials).

Expected Output #9:

• Python program that lists Strong Numbers.

• Optimized version with explanation.

Prompt:

```
'''
Generate a function to display all Strong Numbers within a given range using loops,
and regenerate an optimized version using precomputed factorials.
'''
```

Code:

```python
import math
def is_strong_number(n, factorials):
    return n == sum(factorials[int(digit)] for digit in str(n))
def find_strong_numbers(start, end):
    factorials = {i: math.factorial(i) for i in range(10)}
    strong_numbers = []
    for num in range(start, end + 1):
        if is_strong_number(num, factorials):
            strong_numbers.append(num)
    return strong_numbers
# Example usage
start_range = 1
end_range = 50000
strong_numbers = find_strong_numbers(start_range, end_range)
print("Strong numbers between", start_range, "and", end_range, "are:", strong_numbers)
```

Output:

```
Strong numbers between 1 and 50000 are: [1, 2, 145, 40585]
```

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:

- Full Name
- Branch

- SGPA

Expected Output

A reusable Python function that correctly navigates and extracts values from nested dictionaries based on the provided examples

Prompt:

```
'''
Create a Python function that extracts full name, branch, and
SGPA from a nested dictionary representing student information.
Examples:
1.  {"personal": {"full_name": "Naresh"},"academic": {"branch": "CSE","sgpa": 8.0}}
2.{"personal": {"full_name": "Suresh"},"academic": {"branch": "ECE","sgpa": 7.4}}
'''
```

Code:

```python
def extract_student_info(student_dict):
    full_name = student_dict.get("personal", {}).get("full_name", "N/A")
    branch = student_dict.get("academic", {}).get("branch", "N/A")
    sgpa = student_dict.get("academic", {}).get("sgpa", "N/A")
    return full_name, branch, sgpa
# Example usage
student1 = {"personal": {"full_name": "Deepthi"}, "academic": {"branch": "CSE", "sgpa": 8.0}}
student2 = {"personal": {"full_name": "Kavyasri"}, "academic": {"branch": "ECE", "sgpa": 7.4}}
info1 = extract_student_info(student1)
info2 = extract_student_info(student2)
print(f"Student 1 - Full Name: {info1[0]}, Branch: {info1[1]}, SGPA: {info1[2]}")
print(f"Student 2 - Full Name: {info2[0]}, Branch: {info2[1]}, SGPA: {info2[2]}")
```

Output:

```
Student 1 - Full Name: Deepthi, Branch: CSE, SGPA: 8.0
Student 2 - Full Name: Kavyasri, Branch: ECE, SGPA: 7.4
```

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).

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 Python function that prints all Perfect Numbers in a user given range, and
regenerate an optimized version by checking divisors only up to square root of the number.
'''
```

Code:

```python
def is_perfect_number(n):
    if n < 2:
        return False
    divisors_sum = 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(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
start_range = 1
end_range = 10000
perfect_numbers = find_perfect_numbers(start_range, end_range)
print("Perfect numbers between", start_range, "and", end_range, "are:", perfect_numbers)
```

Output:

```
Perfect numbers between 1 and 10000 are: [6, 28, 496, 8128]
```

Explanation:

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