

ASSIGNMENT -6.3

2303A52364

A.Vijaya Karunya

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:
 - 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:

```
#Write a python program to generate a function that displays all automorphic numbers between 1 to 1000 using loops
```

Code:

```
import time as time
def is_automorphic(num):
    square = num * num
    num_str = str(num)
    square_str = str(square)
    return square_str.endswith(num_str)
def display_automorphic_numbers(limit):
    automorphic_numbers = []
    for i in range(1, limit + 1):
        if is_automorphic(i):
            automorphic_numbers.append(i)
    return automorphic_numbers
start_time = time.time()
limit = 1000
automorphic_numbers = display_automorphic_numbers(limit)
print("Automorphic numbers between 1 and 100 using for loop ", limit, "are:", automorphic_numbers)
# Function to check if a number is automorphic
end_time = time.time()
execution_time = end_time - start_time
print("Execution time using for loop: {:.6f} seconds".format(execution_time))
#now write the same program using while loop
def display_automorphic_numbers_while(limit):
    automorphic_numbers = []
    i = 1
    while i <= limit:
        if is_automorphic(i):
            automorphic_numbers.append(i)
        i += 1
    return automorphic_numbers
start_time = time.time()
limit = 1000
automorphic_numbers_while = display_automorphic_numbers_while(limit)
print("Automorphic numbers between 1 and", limit, "using while loop are:", automorphic_numbers_while)
end_time = time.time()
execution_time_while = end_time - start_time
print("Execution time using while loop: {:.6f} seconds".format(execution_time_while))
```

Output:

```
Automorphic numbers between 1 and 100 using for loop 1000 are: [1, 5, 6, 25, 76, 376, 625]
Execution time using for loop: 0.000604 seconds
Automorphic numbers between 1 and 1000 using while loop are: [1, 5, 6, 25, 76, 376, 625]
Execution time using while loop: 0.000600 seconds
```

Suggestion:

Ask the AI to generate a Python program that lists all Automorphic numbers between 1 and 1000 using a for loop, where an Automorphic number is one whose square ends with the number itself. After generating the code, analyze whether the logic correctly checks this condition and whether the approach is efficient. Then ask the AI to regenerate the program using a while loop instead of a for loop, and finally compare both implementations in simple terms by explaining which loop is easier to understand and why.

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

Promot:

```
#2
#Write a python program to generate nested if-elif-else conditions to classify online shopping feedback as Positive, Neutral, or Negative based on a numerical rating (1-5)
```

Code:

```

import time as time
def classify_feedback(rating):
    if rating >= 4 and rating <= 5:
        return "Positive"
    elif rating == 3:
        return "Neutral"
    elif rating >= 1 and rating <= 2:
        return "Negative"
    else:
        return "Invalid rating. Please provide a rating between 1 and 5."
# Example usage
start_time = time.time()
ratings = [5, 4, 3, 2, 1, 6, 0]
for rating in ratings:
    classification = classify_feedback(rating)
    print(f"Rating: {rating} - Feedback: {classification}")
end_time = time.time()
execution_time = end_time - start_time
print("Execution time using if-elif-else: {:.6f} seconds".format(execution_time))

```

```

#now do it using dictionary-based or match-case structure.

def classify_feedback_dict(rating):
    feedback_dict = {
        (4, 5): "Positive",
        (3, 3): "Neutral",
        (1, 2): "Negative"
    }
    for key in feedback_dict:
        if rating >= key[0] and rating <= key[1]:
            return feedback_dict[key]
    return "Invalid rating. Please provide a rating between 1 and 5."
start_time = time.time()
# Example usage
ratings = [5, 4, 3, 2, 1, 6, 0]
for rating in ratings:
    classification = classify_feedback_dict(rating)
    print(f"Rating: {rating} - Feedback: {classification}")
end_time = time.time()
execution_time = end_time - start_time
print("Execution time using dictionary-based approach: {:.6f} seconds".format(execution_time))

```

```

# Using match-case structure (Python 3.10+)
def classify_feedback_match(rating):
    match rating:
        case 4 | 5:
            return "Positive"
        case 3:
            return "Neutral"
        case 1 | 2:
            return "Negative"
        case _:
            return "Invalid rating. Please provide a rating between 1 and 5."
start_time = time.time()
# Example usage
ratings = [5, 4, 3, 2, 1, 6, 0]
for rating in ratings:
    classification = classify_feedback_match(rating)
    print(f"Rating: {rating} - Feedback: {classification}")
end_time = time.time()
execution_time = end_time - start_time
print("Execution time using match-case structure: {:.6f} seconds".format(execution_time))

```

Output:

```

Rating: 5 - Feedback: Positive
Rating: 4 - Feedback: Positive
Rating: 3 - Feedback: Neutral
Rating: 2 - Feedback: Negative
Rating: 1 - Feedback: Negative
Rating: 6 - Feedback: Invalid rating. Please provide a rating between 1 and 5.
Rating: 0 - Feedback: Invalid rating. Please provide a rating between 1 and 5.
Execution time using if-elif-else: 0.000316 seconds
Rating: 5 - Feedback: Positive
Rating: 4 - Feedback: Positive
Rating: 3 - Feedback: Neutral
Rating: 2 - Feedback: Negative
Rating: 1 - Feedback: Negative
Rating: 6 - Feedback: Invalid rating. Please provide a rating between 1 and 5.
Rating: 0 - Feedback: Invalid rating. Please provide a rating between 1 and 5.
Execution time using dictionary-based approach: 0.000971 seconds
Rating: 5 - Feedback: Positive
Rating: 4 - Feedback: Positive
Rating: 3 - Feedback: Neutral
Rating: 2 - Feedback: Negative
Rating: 1 - Feedback: Negative
Rating: 6 - Feedback: Invalid rating. Please provide a rating between 1 and 5.
Rating: 0 - Feedback: Invalid rating. Please provide a rating between 1 and 5.
Execution time using match-case structure: 0.000667 seconds

```

Suggestion:

Ask the AI to write a Python function that uses **nested if-elif-else statements** to classify online shopping feedback as **Positive**, **Neutral**, or **Negative** based on a rating from 1 to 5. Check whether the conditions are correct and easy to understand. Then request the AI to rewrite the same logic using a **dictionary** or **match-case** structure. Finally, ask for a simple explanation comparing both approaches and which one is clearer.

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:

```
#3
#Write a python program to generate 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
```

code:

```
import statistics as stats
def statistical_operations(tuple_num):
    minimum = min(tuple_num)
    maximum = max(tuple_num)
    mean = stats.mean(tuple_num)
    median = stats.median(tuple_num)
    try:
        mode = stats.mode(tuple_num)
    except stats.StatisticsError:
        mode = "No unique mode"
    variance = stats.variance(tuple_num)
    std_deviation = stats.stdev(tuple_num)
    return {
        "Minimum": minimum,
        "Maximum": maximum,
        "Mean": mean,
        "Median": median,
        "Mode": mode,
        "Variance": variance,
        "Standard Deviation": std_deviation
    }
# Example usage
data = (1, 2, 2, 3, 4, 5, 5, 5)
results = statistical_operations(data)
for operation, value in results.items():
    print(f"{operation}: {value}")
```

output:

```
Minimum: 1
Maximum: 5
Mean: 3.375
Median: 3.5
Mode: 5
Variance: 2.5535714285714284
Standard Deviation: 1.5979898086569353
```

suggestion:

Ask the AI to create a Python function called **statistical_operations(tuple_num)** that calculates the minimum, maximum, mean, median, mode, variance, and standard deviation of a tuple of numbers. While writing the function, carefully review the code suggestions given by **GitHub Copilot**. Accept suggestions that are correct and useful, and modify or reject those that are incorrect or unnecessary. Finally, test the function with sample data to confirm all results are accurate.

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:

```
#4
#Write a python program to Create a class Teacher with attributes teacher_id, name, subject, and experience.Class with initializer, method, object creation and 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")
        print("-----")
# Example usage
teacher1 = Teacher(101, "Alice Smith", "Mathematics", 10)
teacher1.display_details()
teacher2 = Teacher(102, "Bob Johnson", "Science", 8)
teacher2.display_details()
teacher3 = Teacher(103, "Charlie Brown", "History", 5)
teacher3.display_details()
teacher4 = Teacher(104, "Diana Prince", "English", 12)
teacher4.display_details()
teacher5 = Teacher(105, "Ethan Hunt", "Physical Education", 7)
teacher5.display_details()
teacher6 = Teacher(106, "Fiona Gallagher", "Art", 4)
teacher6.display_details()
teacher7 = Teacher(107, "George Martin", "Music", 15)
teacher7.display_details()
teacher8 = Teacher(108, "Hannah Baker", "Geography", 6)
teacher8.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
```

```
-----  
Teacher ID: 103
Name: Charlie Brown
Subject: History
Experience: 5 years
```

```
-----  
Teacher ID: 104
Name: Diana Prince
Subject: English
Experience: 12 years
```

```
-----  
Teacher ID: 105
Name: Ethan Hunt
Subject: Physical Education
Experience: 7 years
```

```
-----  
Teacher ID: 106
Name: Fiona Gallagher
Subject: Art
Experience: 4 years
```

```
-----  
Teacher ID: 107
Name: George Martin
Subject: Music
Experience: 15 years
```

```
-----  
Teacher ID: 108
Name: Hannah Baker
Subject: Geography
Experience: 6 years
```

Suggestion:

Ask the AI to write a Python **class named Teacher** with attributes **teacher_id, name, subject, and experience** using an initializer. Add a method that displays all teacher details in a clear format. Then create at least one **object of the class** and call the display method. Finally, check that the attributes are stored and shown correctly.

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:

```
#5  
#Write a python program to generate function that validates Indian phone numbers starting with 6, 7, 8, or 9 and having a total of 10 digits.
```

Code:

```
import re  
def validate_indian_phone_number(phone_number):  
    pattern = r'^[6-9]\d{9}$'  
    if re.match(pattern, phone_number):  
        return True  
    else:  
        return False  
# Example usage  
phone_numbers = ["9876543210", "1234567890", "8765432109", "5678901234", "9123456780"]  
for number in phone_numbers:  
    if validate_indian_phone_number(number):  
        print(f"{number} is a valid Indian phone number.")  
    else:  
        print(f"{number} is not a valid Indian phone number.")
```

Output:

```
9876543210 is a valid Indian phone number.  
1234567890 is not a valid Indian phone number.  
8765432109 is a valid Indian phone number.  
5678901234 is not a valid Indian phone number.  
9123456780 is a valid Indian phone number.
```

Suggestion:

Ask the AI to generate a Python function that validates an **Indian mobile number** using **zero-shot prompting**, without giving any examples. Clearly state the rules that the number must **start with 6, 7, 8, or 9** and must contain **exactly 10 digits**. Review the generated code to ensure all conditions are checked properly. Modify the logic if needed to improve clarity or correctness.

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:

```
#6. write a python program with user input to display armstrong numbers range(1to1000) using a for loop and digit power logic and generate optimized version with list comprehension.
```

Code:

```
def is_armstrong_number(num):
    num_str = str(num)
    num_digits = len(num_str)
    sum_of_powers = sum(int(digit) ** num_digits for digit in num_str)
    return sum_of_powers == num
def display_armstrong_numbers(limit):
    armstrong_numbers = [num for num in range(1, limit + 1) if is_armstrong_number(num)]
    return armstrong_numbers
if __name__ == "__main__":
    limit = int(input("Enter the upper limit to find Armstrong numbers: "))
    armstrong_numbers = display_armstrong_numbers(limit)
    print(f"Armstrong numbers up to {limit}: {armstrong_numbers}")
```

Output:

```
Enter the upper limit to find Armstrong numbers: 500
Armstrong numbers up to 500: [1, 2, 3, 4, 5, 6, 7, 8, 9, 153, 370, 371, 407]
Enter the upper limit to find Armstrong numbers: 850
Armstrong numbers up to 850: [1, 2, 3, 4, 5, 6, 7, 8, 9, 153, 370, 371, 407]
Enter the upper limit to find Armstrong numbers: 700
Armstrong numbers up to 700: [1, 2, 3, 4, 5, 6, 7, 8, 9, 153, 370, 371, 407]
```

Suggestion:

Ask the AI to write a Python function that finds all **Armstrong numbers** within a user-given range using a **for loop** and digit power logic. Verify the correctness by checking known Armstrong numbers like **153** and **370**. After that, request the AI to generate an **optimized version** of the program using **list comprehensions**. Finally, ask for a short explanation comparing the normal and optimized versions.

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:

```
#7. write a python program with user input to display happy numbers range(1to500)using a loop here happy numbers mean repeatedly replace
# a number with the sum of the squares of its digits until the result is either 1 (happy number) or enter a cycle (Not happy) with optimized version.
```

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 display_happy_numbers(limit):
    happy_numbers = [num for num in range(1, limit + 1) if is_happy_number(num)]
    return happy_numbers

if __name__ == "__main__":
    limit = int(input("Enter the upper limit to find happy numbers: "))
    happy_numbers = display_happy_numbers(limit)
    print(f"Happy numbers up to {limit}: {happy_numbers}")
```

Output:

```
Enter the upper limit to find happy numbers: 450
Happy numbers up to 450: [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]
```

```
Happy numbers up to 320: [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]
```

Suggestion:

Ask the AI to write a Python function that prints all **Happy Numbers** within a user-given range such as 1 to 500 using a loop to repeatedly replace a number with the sum of the squares of its digits. Check the correctness of the logic by testing known Happy Numbers like **1, 7, 10, 13, and 19**. Then ask the AI to regenerate an **optimized version** of the program by using a **set to detect cycles** and avoid infinite loops. Finally, request a simple explanation of how the optimized version works and why it is better.

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:

```
#8.write a python program for strong number. A strong number is a number in which the sum of the factorial of each digit is equal to the number itself with optimized version.
```

Code:

```
def factorial(n):
    if n == 0 or n == 1:
        return 1
    result = 1
    for i in range(2, n + 1):
        result *= i
    return result
def is_strong_number(num):
    sum_of_factorials = sum(factorial(int(digit)) for digit in str(num))
    return sum_of_factorials == num
def display_strong_numbers(limit):
    strong_numbers = [num for num in range(1, limit + 1) if is_strong_number(num)]
    return strong_numbers
if __name__ == "__main__":
    limit = int(input("Enter the upper limit to find strong numbers: "))
    strong_numbers = display_strong_numbers(limit)
    print(f"Strong numbers up to {limit}: {strong_numbers}")
```

Output:

```
Enter the upper limit to find strong numbers: 500
Strong numbers up to 500: [1, 2, 145]
Enter the upper limit to find strong numbers: 1000
Strong numbers up to 1000: [1, 2, 145]
```

Suggestion:

Ask the AI to write a Python function that prints all **Happy Numbers** within a user-given range such as 1 to 500 using a loop to repeatedly replace a number with the sum of the squares of its digits. Check the correctness of the logic by testing known Happy Numbers like **1, 7, 10, 13, and 19**. Then ask the AI to regenerate an **optimized version** of the program by using a **set to detect cycles** and avoid infinite loops. Finally, request a simple explanation of how the optimized version works and why it is better.

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 example

prompt:

```
cd /Users/john/Desktop/Downloads/Py/Exercises/2023-08-14/Example2/main
'''Generate a python code that parses a nested dictionary representing student information.
The function should extract and return
Name,Branch,SGPA
Example1:
Full Name: John Doe
Branch: Computer Science
SGPA: 8.5
Example2:
Full Name: Jane Smith
Branch: Mechanical Engineering
SGPA: 9.0
Example3:
Full Name: Alice Johnson
Branch: Electrical Engineering
SGPA: 8.8
Take user input as a nested dictionary and return the formatted string.'''

```

code:

```
def extract_student_info(student_dict):
    try:
        full_name = student_dict['personal_info']['name']
        branch = student_dict['academic_info']['branch']
        sgpa = student_dict['academic_info']['sgpa']

        return f"Full Name: {full_name}\nBranch: {branch}\nSGPA: {sgpa}"
    except KeyError as e:
        return f"Missing key in the dictionary: {e}"
def main():
    # Example nested dictionary input
    student_info = {
        'personal_info': [
            {'name': 'Ram',
             'age': 20,
            },
        ],
        'academic_info': {
            'branch': 'Computer Science',
            'sgpa': 9.5
        }
    }
    print(extract_student_info(student_info))
def extract_student_info(student_dict):
    try:
        full_name = student_dict['personal_info']['name']
        branch = student_dict['academic_info']['branch']
        sgpa = student_dict['academic_info']['sgpa']

        return f"Full Name: {full_name}\nBranch: {branch}\nSGPA: {sgpa}"
    except KeyError as e:
        return f"Missing key in the dictionary: {e}"
if __name__ == "__main__":
    main()
```

output:

```
Full Name: Ram
Branch: Computer Science
SGPA: 9.5
```

suggestion:

Ask the AI to first look at **2–3 example nested dictionaries** that show how student information is stored. Using these examples, instruct the AI to write a

Python function that correctly navigates the nested structure and extracts the **Full Name, Branch, and SGPA**. Make sure the function works even if the data is deeply nested. Finally, check the output using the given examples to confirm the values are extracted correctly.

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 \sqrt{n}).

Expected Output #12:

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

Prompt:

```
#10.write a python program to display all perfect numbers. perfect number means positive integer equal to the sum of its proper divisors excluding the number itself.
```

Code:

```
#10.write a python program to display all perfect numbers. perfect number means positive integer equal to the sum of its proper divisors excluding the number itself.
def is_perfect_number(n):
    if n < 2:
        return False
    divisors_sum = sum(i for i in range(1, n) if n % i == 0)
    return divisors_sum == n
def display_perfect_numbers(limit):
    perfect_numbers = []
    for num in range(2, limit + 1):
        if is_perfect_number(num):
            perfect_numbers.append(num)
    return perfect_numbers
if __name__ == "__main__":
    limit = int(input("Enter the upper limit to find perfect numbers: "))
    perfect_numbers = display_perfect_numbers(limit)
    print(f"Perfect numbers up to {limit}: {perfect_numbers}")
```

Output:

```
Enter the upper limit to find perfect numbers: 7
Perfect numbers up to 7: [6]
```

Suggestion:

Ask the AI to write a Python program that displays all **Perfect Numbers** within a user-given range (such as 1 to 1000) by using a **for loop** to find and sum proper divisors of each number. Verify the correctness of the output using known Perfect Numbers like **6, 28, and 496**. After checking the logic, request

the AI to regenerate an **optimized version** of the program by checking divisors only up to \sqrt{n} . Finally, ask for a short explanation comparing the normal and optimized approaches and why the optimized version is more efficient.