

Lab Assignment 6.3

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:

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

For Loop:

```
1 """  
2 Write a python program to generate numbers which are automorphic numbers in a given range using for loop.  
3 """  
4  
5 from time import time  
6  
7 start_time = time()  
8  
9 def is_automorphic(num):  
10     square = num * num  
11     num_str = str(num)  
12     square_str = str(square)  
13     return square_str.endswith(num_str)  
14  
15 def automorphic_numbers_in_range(start, end):  
16     automorphic_numbers = []  
17     for num in range(start, end + 1):  
18         if is_automorphic(num):  
19             automorphic_numbers.append(num)  
20     return automorphic_numbers  
21  
22 start_range = 1  
23 end_range = 100000  
24 result = automorphic_numbers_in_range(start_range, end_range)  
25 print(f"Automorphic numbers between {start_range} and {end_range} are: {result}")  
26  
27 end_time = time()  
28 print(f"Execution time for for loop: {end_time - start_time} seconds")
```

While Loop:

```
1 """  
2 Write a python program to generate numbers which are automorphic numbers in a given range using while loop.  
3 """  
4  
5 from time import time  
6  
7 start_time = time()  
8  
9 def is_automorphic(num):  
10     square = num * num  
11     num_str = str(num)  
12     square_str = str(square)  
13     return square_str.endswith(num_str)  
14  
15 def automorphic_numbers_in_range(start, end):  
16     automorphic_numbers = []  
17     num = start  
18     while num <= end:  
19         if is_automorphic(num):  
20             automorphic_numbers.append(num)  
21         num += 1  
22     return automorphic_numbers  
23  
24 start_range = 1 # int(input("Enter the start of the range: "))  
25 end_range = 100000 # int(input("Enter the end of the range: "))  
26 result = automorphic_numbers_in_range(start_range, end_range)  
27 print(f"Automorphic numbers between {start_range} and {end_range} are: {result}")  
28  
29 end_time = time()  
30 print(f"Execution time for while loop: {end_time - start_time} seconds")
```

Execution time difference:

```
PS C:\Users\rhars\Documents\.Dev\Uni > & C:/Python314/python.exe "c:/Users/rhars/Documents/.Dev/Uni/Sem-VI/AI-Assisted/Ass 6.3/automorphic_for.py"
● Automorphic numbers between 1 and 100000 are: [1, 5, 6, 25, 76, 376, 625, 9376, 90625]
Execution time for for loop: 0.03836226463317871 seconds
PS C:\Users\rhars\Documents\.Dev\Uni > & C:/Python314/python.exe "c:/Users/rhars/Documents/.Dev/Uni/Sem-VI/AI-Assisted/Ass 6.3/automorphic_while.py"
Automorphic numbers between 1 and 100000 are: [1, 5, 6, 25, 76, 376, 625, 9376, 90625]
Execution time for while loop: 0.04222440719604492 seconds
```

Observation:

Since the range is given we should use for loop, as while loop is used for unknown ranges, and for this particular situation for loop has a better execution time.

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).

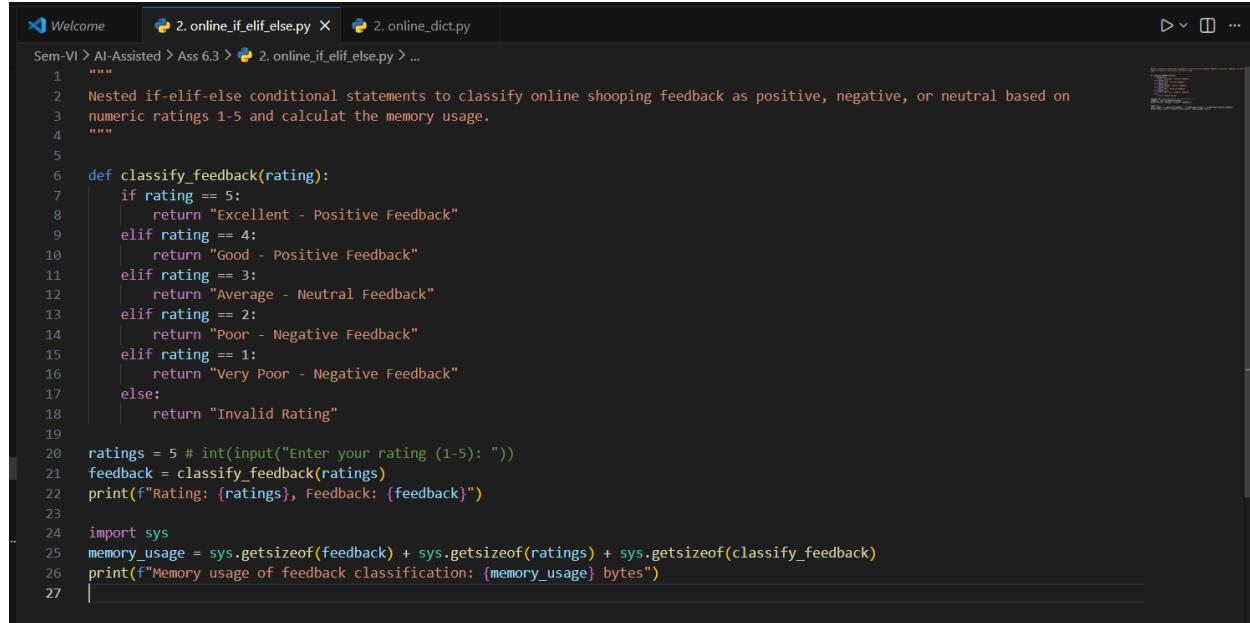
Instructions:

- 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:

Feedback classification function with explanation and an alternative approach.

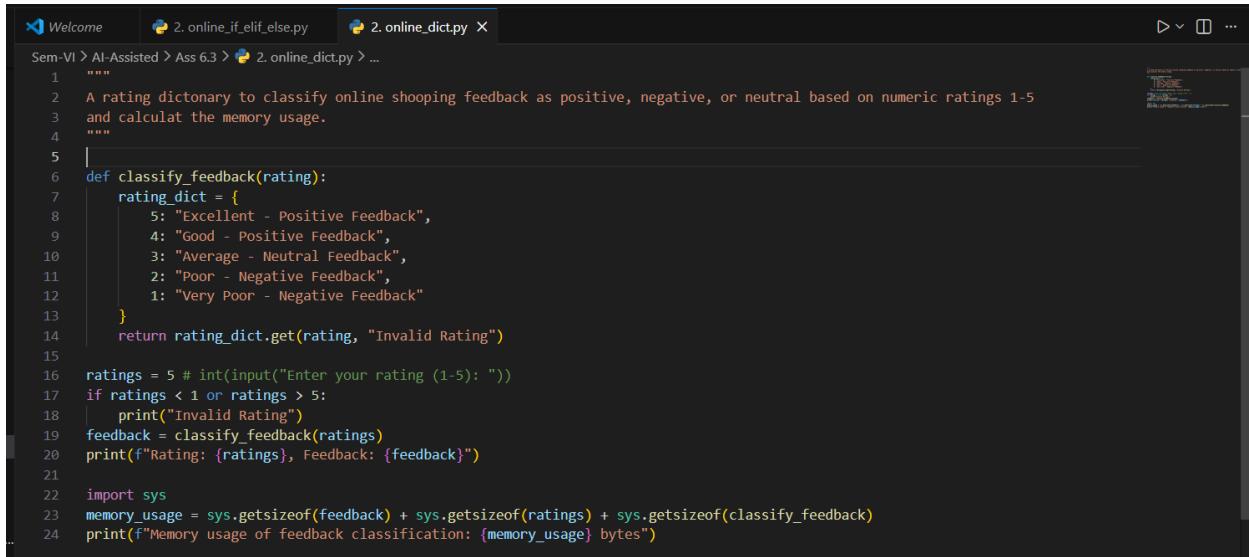
Nested if-elif-else based classification:



The screenshot shows a code editor window with a dark theme. The title bar says "Welcome". There are two tabs open: "2. online_if_elif_else.py" and "2. online_dict.py". The current tab, "2. online_if_elif_else.py", contains the following Python code:

```
1 """
2 Nested if-elif-else conditional statements to classify online shopping feedback as positive, negative, or neutral based on
3 numeric ratings 1-5 and calculate the memory usage.
4 """
5
6 def classify_feedback(rating):
7     if rating == 5:
8         return "Excellent - Positive Feedback"
9     elif rating == 4:
10        return "Good - Positive Feedback"
11    elif rating == 3:
12        return "Average - Neutral Feedback"
13    elif rating == 2:
14        return "Poor - Negative Feedback"
15    elif rating == 1:
16        return "Very Poor - Negative Feedback"
17    else:
18        return "Invalid Rating"
19
20 ratings = 5 # int(input("Enter your rating (1-5): "))
21 feedback = classify_feedback(ratings)
22 print(f"Rating: {ratings}, Feedback: {feedback}")
23
24 import sys
25 memory_usage = sys.getsizeof(feedback) + sys.getsizeof(ratings) + sys.getsizeof(classify_feedback)
26 print(f"Memory usage of feedback classification: {memory_usage} bytes")
27 |
```

Dictionary based classification:



```
1 """
2 A rating dictionary to classify online shopping feedback as positive, negative, or neutral based on numeric ratings 1-5
3 and calculate the memory usage.
4 """
5
6 def classify_feedback(rating):
7     rating_dict = {
8         5: "Excellent - Positive Feedback",
9         4: "Good - Positive Feedback",
10        3: "Average - Neutral Feedback",
11        2: "Poor - Negative Feedback",
12        1: "Very Poor - Negative Feedback"
13    }
14    return rating_dict.get(rating, "Invalid Rating")
15
16 ratings = 5 # int(input("Enter your rating (1-5): "))
17 if ratings < 1 or ratings > 5:
18     print("Invalid Rating")
19 feedback = classify_feedback(ratings)
20 print(f"Rating: {ratings}, Feedback: {feedback}")
21
22 import sys
23 memory_usage = sys.getsizeof(feedback) + sys.getsizeof(ratings) + sys.getsizeof(classify_feedback)
24 print(f"Memory usage of feedback classification: {memory_usage} bytes")
```

Memory usage comparison:

```
• PS C:\Users\rhars\Documents\.Dev\Uni> & C:/Python314/python.exe "c:/Users/rhars/Documents/.Dev/Uni/Sem-VI/AI-Assisted/Ass 6.3/2. online_if_elif_else.py"
Rating: 5, Feedback: Excellent - Positive Feedback
Memory usage of feedback classification: 266 bytes
• PS C:\Users\rhars\Documents\.Dev\Uni> & C:/Python314/python.exe "c:/Users/rhars/Documents/.Dev/Uni/Sem-VI/AI-Assisted/Ass 6.3/2. online_dict.py"
Rating: 5, Feedback: Excellent - Positive Feedback
Memory usage of feedback classification: 266 bytes
```

Observation:

- Both Dictionary and Nested if-elif-else based comparison has the same memory usage
- The major difference comes from the readability and the preference, where dict is always preferred as its lookup time is O(1)

3. Statistical Operations

Task:

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

```

Welcome  x  2. online_if_else.py  2. online_dict.py  3. statistics_import.py  ...
Sem-VI > AI-Assisted > Ass 6.3 > 3. statistics_import.py > ...
1 """
2 Perform statistical operation on a tuple of numbers by defining a function statistical_operations(tuple_num)
3 The statistical operations to be performed are:
4 Maximum, Minimum
5 Mean, Median, Mode
6 Variance, Standard Deviation
7 """
8
9 import statistics as stats
10
11 def statistical_operations(tuple_num):
12     results = {}
13
14     results['Maximum'] = max(tuple_num)
15     results['Minimum'] = min(tuple_num)
16     results['Mean'] = stats.mean(tuple_num)
17     results['Median'] = stats.median(tuple_num)
18     try:
19         results['Mode'] = stats.mode(tuple_num)
20     except stats.StatisticsError:
21         results['Mode'] = "No unique mode found"
22     results['Variance'] = stats.variance(tuple_num)
23     results['Standard Deviation'] = stats.stdev(tuple_num)
24
25     return results
26
27 # Example usage
28 numbers = (10, 20, 20, 30, 40, 50, 50, 50)
29 stats_results = statistical_operations(numbers)
30 print("Statistical Operations Results:")
31 for key, value in stats_results.items():
32     print(f"{key}: {value}")

```

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.

```

notes.txt  x  4.py  ...
Sem-VI > AI-Assisted > Ass 6.3 > 4.py > ...
1 class Teacher:
2     def __init__(self, teacher_id, name, subject, experience):
3         self.teacher_id = teacher_id
4         self.name = name
5         self.subject = subject
6         self.experience = experience
7
8     def display(self):
9         print(f"Teacher ID: {self.teacher_id}")
10        print(f"Name: {self.name}")
11        print(f"Subject: {self.subject}")
12        print(f"Experience: {self.experience} years")
13
14 teacher1 = Teacher(101, "Anand", "Biology", 10)
15 teacher1.display()

```

5. Zero-Shot Prompting with Conditional Validation

Task:

Use zero-shot prompting to instruct an AI tool to generate a function that validates an Indian mobile number.

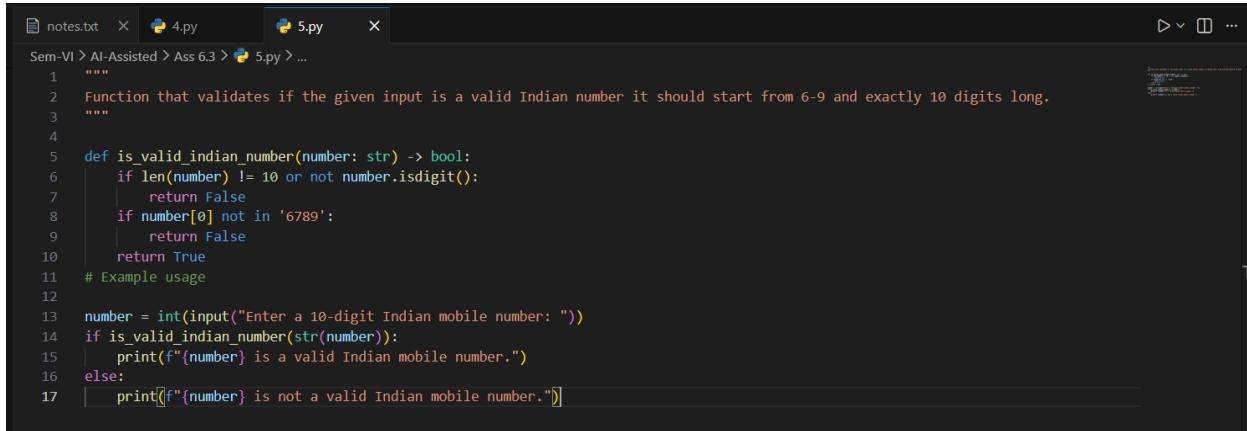
Requirements:

- 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.



```

notes.txt 4.py 5.py ...
Sem-VI > AI-Assisted > Ass 6.3 > 5.py > ...
1 """
2 Function that validates if the given input is a valid Indian number it should start from 6-9 and exactly 10 digits long.
3 """
4
5 def is_valid_indian_number(number: str) -> bool:
6     if len(number) != 10 or not number.isdigit():
7         return False
8     if number[0] not in '6789':
9         return False
10    return True
11 # Example usage
12
13 number = int(input("Enter a 10-digit Indian mobile number: "))
14 if is_valid_indian_number(str(number)):
15     print(f"{number} is a valid Indian mobile number.")
16 else:
17     print(f"{number} is not a valid Indian mobile number.")

```

6. (Loops – Armstrong Numbers in a Range)

Task:

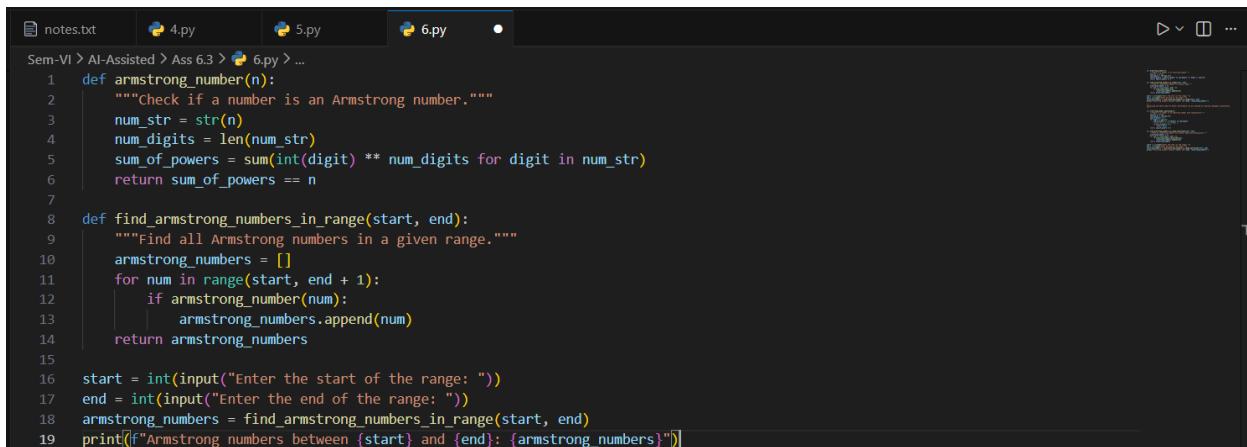
Write a function using AI that finds all Armstrong numbers in a user-specified range (e.g., 1 to 1000).

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:

- Python program listing Armstrong numbers in the range.
- Optimized version with explanation.

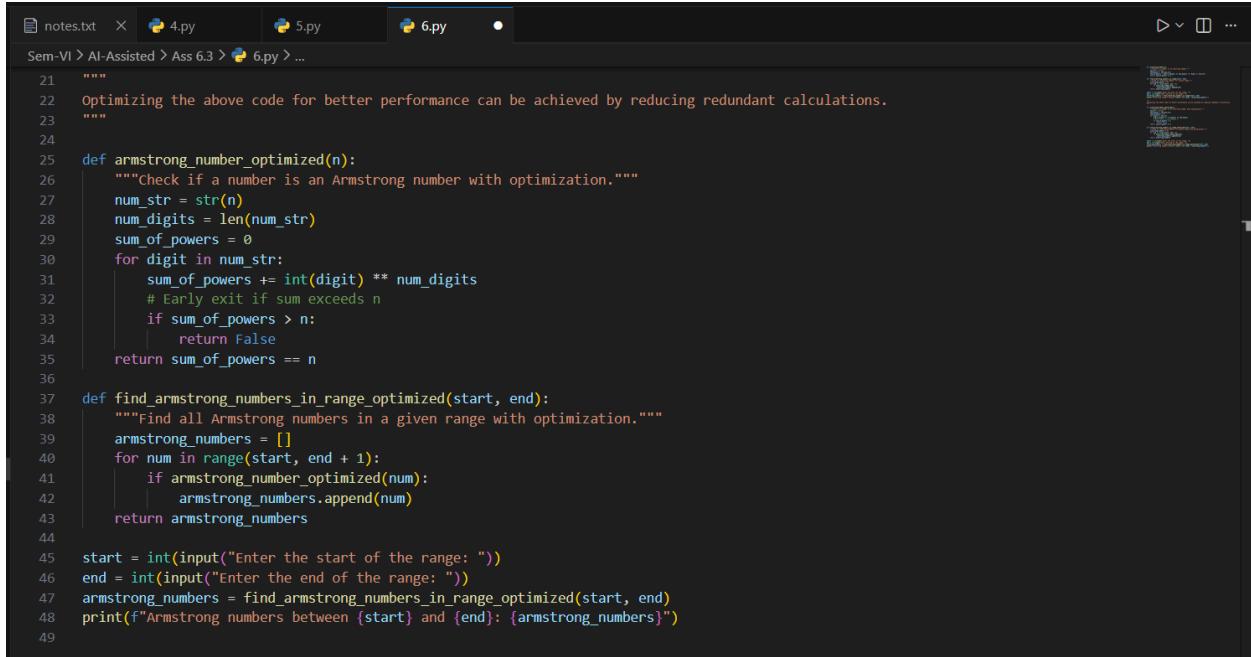


```

notes.txt 4.py 5.py 6.py ...
Sem-VI > AI-Assisted > Ass 6.3 > 6.py > ...
1 def armstrong_number(n):
2     """Check if a number is an Armstrong number."""
3     num_str = str(n)
4     num_digits = len(num_str)
5     sum_of_powers = sum(int(digit) ** num_digits for digit in num_str)
6     return sum_of_powers == n
7
8 def find_armstrong_numbers_in_range(start, end):
9     """Find all Armstrong numbers in a given range."""
10    armstrong_numbers = []
11    for num in range(start, end + 1):
12        if armstrong_number(num):
13            armstrong_numbers.append(num)
14    return armstrong_numbers
15
16 start = int(input("Enter the start of the range: "))
17 end = int(input("Enter the end of the range: "))
18 armstrong_numbers = find_armstrong_numbers_in_range(start, end)
19 print(f"Armstrong numbers between {start} and {end}: {armstrong_numbers}")

```

Optimized Version:



The screenshot shows a code editor window with several tabs at the top: notes.txt, 4.py, 5.py, and 6.py. The 6.py tab is active, displaying the following Python code:

```
21 """
22 Optimizing the above code for better performance can be achieved by reducing redundant calculations.
23 """
24
25 def armstrong_number_optimized(n):
26     """Check if a number is an Armstrong number with optimization."""
27     num_str = str(n)
28     num_digits = len(num_str)
29     sum_of_powers = 0
30     for digit in num_str:
31         sum_of_powers += int(digit) ** num_digits
32         # Early exit if sum exceeds n
33         if sum_of_powers > n:
34             return False
35     return sum_of_powers == n
36
37 def find_armstrong_numbers_in_range_optimized(start, end):
38     """Find all Armstrong numbers in a given range with optimization."""
39     armstrong_numbers = []
40     for num in range(start, end + 1):
41         if armstrong_number_optimized(num):
42             armstrong_numbers.append(num)
43     return armstrong_numbers
44
45 start = int(input("Enter the start of the range: "))
46 end = int(input("Enter the end of the range: "))
47 armstrong_numbers = find_armstrong_numbers_in_range_optimized(start, end)
48 print(f"Armstrong numbers between {start} and {end}: {armstrong_numbers}")
```

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:

- Python program that prints all Happy Numbers within a range.
- Optimized version using cycle detection with explanation.

```

notes.txt 4.py 5.py 6.py 7.py x
Sem-VI > AI-Assisted > Ass 6.3 > 7.py > ...
1
2 def happy_number(n):
3     seen = set()
4     while n != 1 and n not in seen:
5         seen.add(n)
6         n = sum(int(digit) ** 2 for digit in str(n))
7     return n == 1
8
9 def find_happy_numbers_in_range(start, end):
10    happy_numbers = []
11    for num in range(start, end + 1):
12        if happy_number(num):
13            happy_numbers.append(num)
14    return happy_numbers
15
16 start = int(input("Enter the start of the range: "))
17 end = int(input("Enter the end of the range: "))
18 happy_numbers = find_happy_numbers_in_range(start, end)
19 print(f"Happy numbers between {start} and {end}: {happy_numbers}")

```

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:

- Python program that lists Strong Numbers.
- Optimized version with explanation.

```

notes.txt 4.py 5.py 6.py 7.py 8.py x
Sem-VI > AI-Assisted > Ass 6.3 > 8.py > ...
1
2 def strong_number(n):
3     """Check if a number is a Strong number."""
4     from math import factorial
5
6     num_str = str(n)
7     sum_of_factorials = sum(factorial(int(digit)) for digit in num_str)
8     return sum_of_factorials == n
9
10 def find_strong_numbers_in_range(start, end):
11     """Find all Strong numbers in a given range."""
12     strong_numbers = []
13     for num in range(start, end + 1):
14         if strong_number(num):
15             strong_numbers.append(num)
16     return strong_numbers
17
18 start = int(input("Enter the start of the range: "))
19 end = int(input("Enter the end of the range: "))
20 strong_numbers = find_strong_numbers_in_range(start, end)
21 print(f"Strong numbers between {start} and {end}: {strong_numbers}")

```

Optimized Version:

```
notes.txt 4.py 5.py 6.py 7.py 8.py ...  
Sem-VI > AI-Assisted > Ass 6.3 > 8.py > ...  
21 """  
22     """Optimize the above code for better performance by precomputing factorials of digits 0-9.  
23     """  
24  
25  
26 def strong_number_optimized(n, factorials):  
27     """Check if a number is a Strong number with optimization."""  
28     num_str = str(n)  
29     sum_of_factorials = 0  
30     for digit in num_str:  
31         sum_of_factorials += factorials[int(digit)]  
32         # Early exit if sum exceeds n  
33         if sum_of_factorials > n:  
34             return False  
35     return sum_of_factorials == n  
36  
37 def find_strong_numbers_in_range_optimized(start, end):  
38     """Find all Strong numbers in a given range with optimization."""  
39     from math import factorial  
40  
41     # Precompute factorials of digits 0-9  
42     factorials = {i: factorial(i) for i in range(10)}  
43  
44     strong_numbers = []  
45     for num in range(start, end + 1):  
46         if strong_number_optimized(num, factorials):  
47             strong_numbers.append(num)  
48     return strong_numbers  
49  
50 start = int(input("Enter the start of the range: "))  
51 end = int(input("Enter the end of the range: "))  
52 strong_numbers = find_strong_numbers_in_range_optimized(start, end)  
53 print(f"Strong numbers between {start} and {end}: {strong_numbers}")
```

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

```

notes.txt X 4.py 5.py 6.py 7.py 8.py 9.py X
Sem-VI > AI-Assisted > Ass 6.3 > 9.py > ...
1 """
2 A function that parses a nested dictionary representing student information it should return Full Name, Branch, SGPA.
3 Example:
4 Input:
5 ex1 = {
6     "student_1" : {
7         "Full Name": "Alice Johnson",
8         "Branch": "Computer Science",
9         "SGPA": 9.1
10    }
11 }
12 ex2 = {
13     "student_2" : {
14         "Full Name": "Bob Smith",
15         "Branch": "Mechanical Engineering",
16         "SGPA": 8.5
17    }
18 }
19 ex3 = {
20     "student_3" : {
21         "Full Name": "Charlie Brown",
22         "Branch": "Electrical Engineering",
23         "SGPA": 8.9
24    },
25     "student_4" : {
26         "Full Name": "Diana Prince",
27         "Branch": "Civil Engineering",
28         "SGPA": 9.3
29    }
30 }
31 """
32
33 def parse_student_info(student_dict):
34     """Parse a nested dictionary to extract student information."""
35     for student_key, info in student_dict.items():
36         full_name = info.get("Full Name", "N/A")
37         branch = info.get("Branch", "N/A")
38         sgpa = info.get("SGPA", "N/A")
39         return full_name, branch, sgpa
40 # Example usage
41 ex1 = {
42     "student_1" : {
43         "Full Name": "Alice Johnson",
44         "Branch": "Computer Science",
45         "SGPA": 9.1
46    }
47 }
48
49 full_name, branch, sgpa = parse_student_info(ex1)
50 print(f"Full Name: {full_name}, Branch: {branch}, SGPA: {sgpa}")

```

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).

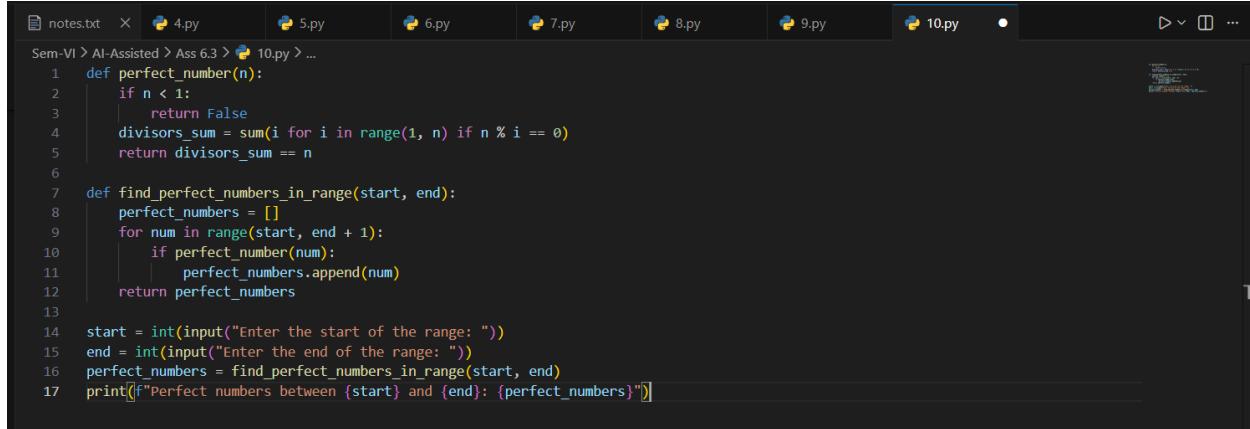
Instructions:

- 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:

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



The screenshot shows a code editor window with multiple tabs at the top, including "notes.txt", "4.py", "5.py", "6.py", "7.py", "8.py", "9.py", "10.py" (which is the active tab), and "...". The main pane displays the following Python code:

```
1 def perfect_number(n):
2     if n < 1:
3         return False
4     divisors_sum = sum(i for i in range(1, n) if n % i == 0)
5     return divisors_sum == n
6
7 def find_perfect_numbers_in_range(start, end):
8     perfect_numbers = []
9     for num in range(start, end + 1):
10        if perfect_number(num):
11            perfect_numbers.append(num)
12    return perfect_numbers
13
14 start = int(input("Enter the start of the range: "))
15 end = int(input("Enter the end of the range: "))
16 perfect_numbers = find_perfect_numbers_in_range(start, end)
17 print(f"Perfect numbers between {start} and {end}: {perfect_numbers}")
```