AIAC - Lab Assignment : 6.3

Name : Syed Zaid Sami Khizar

Enrollment No : 2303A52016

Batch : 31

Lab 6: AI-Based Code Completion – Classes, Loops, and Conditionals
Lab Objectives:
• To explore AI-powered auto-completion features for core Python constructs.
• To analyze how AI suggests logic for class definitions, loops, and conditionals.
• To evaluate the completeness and correctness of code generated by AI assistants.
Lab Outcomes (LOs):
After completing this lab, students will be able to:
• Use AI tools to generate and complete class definitions and methods.
• Understand and assess AI-suggested loops for iterative tasks.
• Generate conditional statements through prompt-driven suggestions.
• Critically evaluate AI-assisted code for correctness and clarity.

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.

Solution :

Prompt : Generate a function that displays all Automorphic numbers between 1 and 1000 using a for loop.

• Instructions:

o Generated code to list Automorphic numbers using a for loop.

o Analyze the correctness and efficiency of the generated logic.

o To regenerate using a while loop and compare both implementations.

Code Given By AI :

```
# Function using FOR LOOP
def automorphic_numbers_for():
    """
    Displays all automorphic numbers between 1 and 1000.
    An automorphic number is a number whose square ends with
the number itself.
    Example: 5^2 = 25 (ends with 5), 6^2 = 36 (ends with 6)
    """
    print("Automorphic numbers (FOR LOOP):")
    automorphic_list = []

    for num in range(1, 1001):
        square = num * num
        if square % 10 == num % 10:  # Check if last digit of
square matches last digit of num
            automorphic_list.append(num)

    print(automorphic_list)
    return automorphic_list
```

```python
# Function using WHILE LOOP
def automorphic_numbers_while():
    """
    Same functionality using while loop for comparison.
    """
    print("Automorphic numbers (WHILE LOOP):")
    automorphic_list = []
    num = 1

    while num <= 1000:
        square = num * num
        if square % 10 == num % 10:
            automorphic_list.append(num)
        num += 1

    print(automorphic_list)
    return automorphic_list


# Main execution
if __name__ == "__main__":
    automorphic_numbers_for()
    print()
    automorphic_numbers_while()

    # Comparison
    print("\nComparison:")
    print("- FOR LOOP: More Pythonic, cleaner, preferred for
iterating fixed ranges")
    print("- WHILE LOOP: More control, better for conditional
iterations")
    print("- Both have O(n) time complexity")
```

Output:

```
Automorphic numbers (FOR LOOP):
[1, 5, 6, 10, 11, 15, 16, 20, 21, 25, 26, 30, 31, 35, 36, 40, 41, 45, 46, 50, 51, 55, 56, 60, 61, 65, 66, 70, 71,
 75, 76, 80, 81, 85, 86, 90, 91, 95, 96, 100, 101, 105, 106, 110, 111, 115, 116, 120, 121, 125, 126, 130, 131, 13
5, 136, 140, 141, 145, 146, 150, 151, 155, 156, 160, 161, 165, 166, 170, 171, 175, 176, 180, 181, 185, 186, 190,
191, 195, 196, 200, 201, 205, 206, 210, 211, 215, 216, 220, 221, 225, 226, 230, 231, 235, 236, 240, 241, 245, 246
, 250, 251, 255, 256, 260, 261, 265, 266, 270, 271, 275, 276, 280, 281, 285, 286, 290, 291, 295, 296, 300, 301, 3
05, 306, 310, 311, 315, 316, 320, 321, 325, 326, 330, 331, 335, 336, 340, 341, 345, 346, 350, 351, 355, 356, 360,
 361, 365, 366, 370, 371, 375, 376, 380, 381, 385, 386, 390, 391, 395, 396, 400, 401, 405, 406, 410, 411, 415, 41
6, 420, 421, 425, 426, 430, 431, 435, 436, 440, 441, 445, 446, 450, 451, 455, 456, 460, 461, 465, 466, 470, 471,
475, 476, 480, 481, 485, 486, 490, 491, 495, 496, 500, 501, 505, 506, 510, 511, 515, 516, 520, 521, 525, 526, 530
, 531, 535, 536, 540, 541, 545, 546, 550, 551, 555, 556, 560, 561, 565, 566, 570, 571, 575, 576, 580, 581, 585, 5
86, 590, 591, 595, 596, 600, 601, 605, 606, 610, 611, 615, 616, 620, 621, 625, 626, 630, 631, 635, 636, 640, 641,
 645, 646, 650, 651, 655, 656, 660, 661, 665, 666, 670, 671, 675, 676, 680, 681, 685, 686, 690, 691, 695, 696, 70
0, 701, 705, 706, 710, 711, 715, 716, 720, 721, 725, 726, 730, 731, 735, 736, 740, 741, 745, 746, 750, 751, 755,
756, 760, 761, 765, 766, 770, 771, 775, 776, 780, 781, 785, 786, 790, 791, 795, 796, 800, 801, 805, 806, 810, 811
, 815, 816, 820, 821, 825, 826, 830, 831, 835, 836, 840, 841, 845, 846, 850, 851, 855, 856, 860, 861, 865, 866, 8
70, 871, 875, 876, 880, 881, 885, 886, 890, 891, 895, 896, 900, 901, 905, 906, 910, 911, 915, 916, 920, 921, 925,
 926, 930, 931, 935, 936, 940, 941, 945, 946, 950, 951, 955, 956, 960, 961, 965, 966, 970, 971, 975, 976, 980, 98
1, 985, 986, 990, 991, 995, 996, 1000]

Automorphic numbers (WHILE LOOP):
[1, 5, 6, 10, 11, 15, 16, 20, 21, 25, 26, 30, 31, 35, 36, 40, 41, 45, 46, 50, 51, 55, 56, 60, 61, 65, 66, 70, 71,
 75, 76, 80, 81, 85, 86, 90, 91, 95, 96, 100, 101, 105, 106, 110, 111, 115, 116, 120, 121, 125, 126, 130, 131, 13
5, 136, 140, 141, 145, 146, 150, 151, 155, 156, 160, 161, 165, 166, 170, 171, 175, 176, 180, 181, 185, 186, 190,
191, 195, 196, 200, 201, 205, 206, 210, 211, 215, 216, 220, 221, 225, 226, 230, 231, 235, 236, 240, 241, 245, 246
, 250, 251, 255, 256, 260, 261, 265, 266, 270, 271, 275, 276, 280, 281, 285, 286, 290, 291, 295, 296, 300, 301, 3
05, 306, 310, 311, 315, 316, 320, 321, 325, 326, 330, 331, 335, 336, 340, 341, 345, 346, 350, 351, 355, 356, 360,
 361, 365, 366, 370, 371, 375, 376, 380, 381, 385, 386, 390, 391, 395, 396, 400, 401, 405, 406, 410, 411, 415, 41
6, 420, 421, 425, 426, 430, 431, 435, 436, 440, 441, 445, 446, 450, 451, 455, 456, 460, 461, 465, 466, 470, 471,
475, 476, 480, 481, 485, 486, 490, 491, 495, 496, 500, 501, 505, 506, 510, 511, 515, 516, 520, 521, 525, 526, 530
, 531, 535, 536, 540, 541, 545, 546, 550, 551, 555, 556, 560, 561, 565, 566, 570, 571, 575, 576, 580, 581, 585, 5
86, 590, 591, 595, 596, 600, 601, 605, 606, 610, 611, 615, 616, 620, 621, 625, 626, 630, 631, 635, 636, 640, 641,
 645, 646, 650, 651, 655, 656, 660, 661, 665, 666, 670, 671, 675, 676, 680, 681, 685, 686, 690, 691, 695, 696, 70
0, 701, 705, 706, 710, 711, 715, 716, 720, 721, 725, 726, 730, 731, 735, 736, 740, 741, 745, 746, 750, 751, 755,
756, 760, 761, 765, 766, 770, 771, 775, 776, 780, 781, 785, 786, 790, 791, 795, 796, 800, 801, 805, 806, 810, 811
, 815, 816, 820, 821, 825, 826, 830, 831, 835, 836, 840, 841, 845, 846, 850, 851, 855, 856, 860, 861, 865, 866, 8
70, 871, 875, 876, 880, 881, 885, 886, 890, 891, 895, 896, 900, 901, 905, 906, 910, 911, 915, 916, 920, 921, 925,
 926, 930, 931, 935, 936, 940, 941, 945, 946, 950, 951, 955, 956, 960, 961, 965, 966, 970, 971, 975, 976, 980, 98
1, 985, 986, 990, 991, 995, 996, 1000]

Comparison:
- FOR LOOP: More Pythonic, cleaner, preferred for iterating fixed ranges
- WHILE LOOP: More control, better for conditional iterations
- Both have O(n) time complexity
```

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:

Task: 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 To rewrite using dictionary-based or match-case structure

Code Given By AI:

```
# Version 1: Nested if-elif-else
def classify_feedback_nested(rating):
    """Classify feedback based on numerical rating (1-5)"""
    if rating >= 1:
        if rating <= 2:
            return "Negative"
        elif rating == 3:
            return "Neutral"
        elif rating <= 5:
            return "Positive"
    return "Invalid rating"


# Version 2: Dictionary-based (cleaner & more maintainable)
def classify_feedback_dict(rating):
    """Classify feedback using dictionary mapping"""
    feedback_map = {
        1: "Negative",
        2: "Negative",
        3: "Neutral",
        4: "Positive",
        5: "Positive"
    }
    return feedback_map.get(rating, "Invalid rating")
```

```python
# Version 3: Match-case (Python 3.10+)
def classify_feedback_match(rating):
    """Classify feedback using match-case"""
    match rating:
        case 1 | 2:
            return "Negative"
        case 3:
            return "Neutral"
        case 4 | 5:
            return "Positive"
        case _:
            return "Invalid rating"


# Test all implementations
if __name__ == "__main__":
    test_ratings = [1, 2, 3, 4, 5, 0, 6]

    for rating in test_ratings:
        print(f"Rating {rating}: {classify_feedback_dict(rating)}")
```

Output :

```
Rating 1: Negative
Rating 2: Negative
Rating 3: Neutral
Rating 4: Positive
Rating 5: Positive
Rating 0: Invalid rating
Rating 6: Invalid rating
```

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 :

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

Code Given By AI :

```
from statistics import mean, median, mode, stdev, variance

def statistical_operations(tuple_num):
    """
    Performs statistical operations on a tuple of numbers.

    Args:
        tuple_num: A tuple of numbers

    Returns:
        A dictionary containing min, max, mean, median, mode,
variance, and stdev
    """
    results = {
        'minimum': min(tuple_num),
        'maximum': max(tuple_num),
        'mean': mean(tuple_num),
        'median': median(tuple_num),
```

```
        'mode': mode(tuple_num),
        'variance': variance(tuple_num),
        'standard_deviation': stdev(tuple_num)
    }
    return results


# Example usage
if __name__ == "__main__":
    numbers = (10, 20, 30, 40, 50, 50)
    stats = statistical_operations(numbers)

    for key, value in stats.items():
        print(f"{key}: {value}")
```

Output :

```
minimum: 10
maximum: 50
mean: 33.333333333333336
median: 35.0
mode: 50
variance: 266.6666666666667
standard_deviation: 16.32993161855452
```

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 Given By AI :

```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

    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")

    @classmethod
    def take_input(cls):
        teacher_id = int(input("Enter Teacher ID: "))
        name = input("Enter Teacher Name: ")
        subject = input("Enter Subject: ")
        experience = int(input("Enter Years of Experience: "))
        return cls(teacher_id, name, subject, experience)


# Creating object using class input method
teacher = Teacher.take_input()

# Displaying details
teacher.display_details()
```

Output :

```
Enter Teacher ID: 32
Enter Teacher Name: Ram
Enter Subject: MATHS
Enter Years of Experience: 4
Teacher ID: 32
Name: Ram
Subject: MATHS
Experience: 4 years
```

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

Prompt :

Use zero-shot prompting t 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 and take input from user

Code Given By AI :

```
def validate_indian_mobile(mobile_number):
    """
    Validates an Indian mobile number using zero-shot prompti
ng logic.

    Requirements:
```

```
    - Must start with 6, 7, 8, or 9
    - Must contain exactly 10 digits

    Args:
        mobile_number (str): The mobile number to validate

    Returns:
        bool: True if valid, False otherwise
    """
    # Check if the input is exactly 10 digits
    if not mobile_number.isdigit() or len(mobile_number) != 1
0:
        return False

    # Check if it starts with 6, 7, 8, or 9
    if mobile_number[0] not in ['6', '7', '8', '9']:
        return False

    return True


# Take input from user
user_input = input("Enter an Indian mobile number: ")
if validate_indian_mobile(user_input):
    print(f"✓ {user_input} is a valid Indian mobile number")
else:
    print(f"✗ {user_input} is not a valid Indian mobile numbe
r")
```

Output :

```
Enter an Indian mobile number: 9999999999
✓ 9999999999 is a valid Indian mobile number
```

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 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.).
- Regenerate an optimized version (using list comprehensions).

Code Given By AI :

```
# Function to find Armstrong numbers in a range

def find_armstrong_numbers(start, end):
    """
    Find all Armstrong numbers in the given range.
    An Armstrong number is a number that equals the sum of it
s digits
    each raised to the power of the number of digits.
    """
    armstrong_numbers = []

    for num in range(start, end + 1):
        # Convert number to string to count digits
```

```
        digits = str(num)
        num_digits = len(digits)

        # Calculate sum of digits raised to power of num_digi
ts
        sum_of_powers = sum(int(digit) ** num_digits for digi
t in digits)

        # Check if it's an Armstrong number
        if sum_of_powers == num:
            armstrong_numbers.append(num)

    return armstrong_numbers


# Optimized version using list comprehension
def find_armstrong_numbers_optimized(start, end):
    """Optimized version using list comprehension."""
    return [
        num for num in range(start, end + 1)
        if sum(int(digit) ** len(str(num)) for digit in str(n
um)) == num
    ]


# Test cases
if __name__ == "__main__":
    # Test with range 1 to 1000
    result = find_armstrong_numbers(1, 1000)
    print(f"Armstrong numbers (1-1000): {result}")

    # Verify known Armstrong numbers
    print(f"Is 153 Armstrong? {153 in result}")
    print(f"Is 370 Armstrong? {370 in result}")
    print(f"Is 371 Armstrong? {371 in result}")
    print(f"Is 407 Armstrong? {407 in result}")
```

```python
    # Test optimized version
    result_optimized = find_armstrong_numbers_optimized(1, 10000)
    print(f"\nArmstrong numbers (1-10000): {result_optimized}")
```

Output :

```
Armstrong numbers (1-1000): [1, 2, 3, 4, 5, 6, 7, 8, 9, 153, 370, 371, 407]
Is 153 Armstrong? True
Is 370 Armstrong? True
Is 371 Armstrong? True
Is 407 Armstrong? True

Armstrong numbers (1-10000): [1, 2, 3, 4, 5, 6, 7, 8, 9, 153, 370, 371, 407, 1634, 8208, 9474]
```

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 #7:

• Python program that prints all Happy Numbers within a range.

• Optimized version using cycle detection with explanation.

Code Given By AI :

```
# Task Description #7 (Loops – Happy Numbers in a Range) Tas
k: Generate a function using  displays all Happy Numbers with
in a user-specified range (e.g., 1 to 500).
# take input from the user for the range
from sqlite3 import Time


start_range = int(input("Enter the start of the range: "))
end_range = int(input("Enter the end of the range: "))

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 happy_numbers_in_range(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 = 500
happy_numbers = happy_numbers_in_range(start_range, end_rang
e)
print(f"Happy numbers between {start_range} and {end_range}:
{happy_numbers}")
```

Output :

```
Enter the start of the range: 1
Enter the end of the range: 500
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]
```

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 #8:
• Python program that lists Strong Numbers.
• Optimized version with explanation.

Code Given By AI :

```
# Generate a Python function using loops to find all Strong Numbers in a given range, extract digits and compute factorials, validate with examples (1, 2, 145), produce an optimized version by precomputing digit factorials, and explain the optimization.
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):
    original_num = num
    sum_of_factorials = 0
    while num > 0:
        digit = num % 10
        sum_of_factorials += factorial(digit)
        num //= 10
    return sum_of_factorials == original_num
def find_strong_numbers_in_range(start, end):
```

```
    strong_numbers = []
    for num in range(start, end + 1):
        if is_strong_number(num):
            strong_numbers.append(num)
    return strong_numbers
# Validate with examples
print(find_strong_numbers_in_range(1, 200))  # Should include 1, 2, 145
```

Output :

```
[1, 2, 145]
```

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 Given By AI :

```
# Function to extract student information from a nested dictionary
```

```python
def extract_student_info(data):
    first_name = data["student"]["name"]["first"]
    last_name = data["student"]["name"]["last"]
    branch = data["student"]["academic"]["branch"]
    sgpa = data["student"]["academic"]["sgpa"]

    full_name = f"{first_name} {last_name}"

    return {
        "Full Name": full_name,
        "Branch": branch,
        "SGPA": sgpa
    }


# Sample input
student_data = {
    "student": {
        "name": {
            "first": "Aarav",
            "last": "Sharma"
        },
        "academic": {
            "branch": "CSE",
            "sgpa": 8.7
        }
    }
}

# Function call
result = extract_student_info(student_data)

# Output
print(result)
```

Output :

```
{'Full Name': 'Aarav Sharma', 'Branch': 'CSE', 'SGPA': 8.7}
```

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 #10:

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

• Optimized version with explanation.

Code Given By AI :

```
def find_perfect_numbers(start, end):
    """
    Find all perfect numbers in the given range.
    A perfect number equals the sum of its proper divisors (e
xcluding itself).
    """
    perfect_numbers = []

    for num in range(start, end + 1):
        # Calculate sum of proper divisors
        divisor_sum = 0

        # Optimized: check divisors only up to sqrt(n)
```

```
        for i in range(1, int(num ** 0.5) + 1):
            if num % i == 0:
                divisor_sum += i  # Add the divisor
                # Add the complementary divisor (if different
and not the number itself)
                if i != num // i and num // i != num:
                    divisor_sum += num // i

        # Check if perfect number
        if divisor_sum == num:
            perfect_numbers.append(num)

    return perfect_numbers


# Main program
if __name__ == "__main__":
    # Get user input
    try:
        start = int(input("Enter the start of range: "))
        end = int(input("Enter the end of range: "))

        if start < 1 or end < start:
            print("Invalid range. Please enter positive integ
ers with start <= end.")
        else:
            result = find_perfect_numbers(start, end)

            if result:
                print(f"\nPerfect Numbers between {start} and
{end}:")
                for num in result:
                    print(num)
            else:
                print(f"\nNo perfect numbers found between {s
tart} and {end}.")
```

```python
    except ValueError:
        print("Please enter valid integers.")
```

Output :

```
Enter the start of range: 1
Enter the end of range: 1000

Perfect Numbers between 1 and 1000:
1
6
28
496
```