Lab Assignment 3.1

Name: M. Hasini

Hall ticket :2303A51109

Batch :02

Lab Experiment: Prompt Engineering – Improving Prompts and Context Management

Lab Objectives

1. To understand and apply different prompt engineering techniques for generating Python programs using AI-assisted tools.
2. To analyze the impact of context and examples on the accuracy and efficiency of AI-generated code.
3. To develop and refine real-world Python applications through iterative prompt improvement.

Lab Outcomes

1. Students will be able to design effective prompts to generate correct and optimized Python code.
2. Students will be able to compare and evaluate AI-generated solutions produced using different prompting strategies.
3. Students will be able to implement and document real-world Python applications using AI-assisted coding tools.

Experiment – Prompt Engineering Techniques

Task Description

Design and refine prompts using different prompting strategies to generate Python programs for basic computational problems

Question 1: Zero-Shot Prompting (Palindrome Number Program)

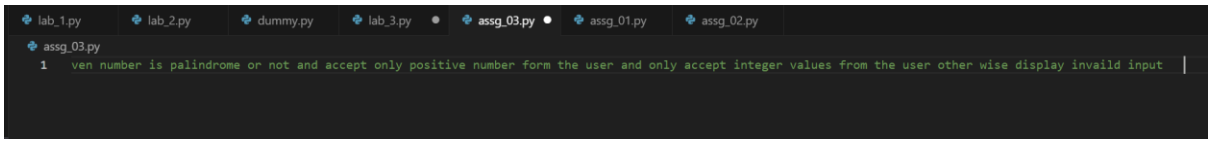
Write a **zero-shot prompt** (without providing any examples) to generate a Python function that checks whether a given number is a palindrome.

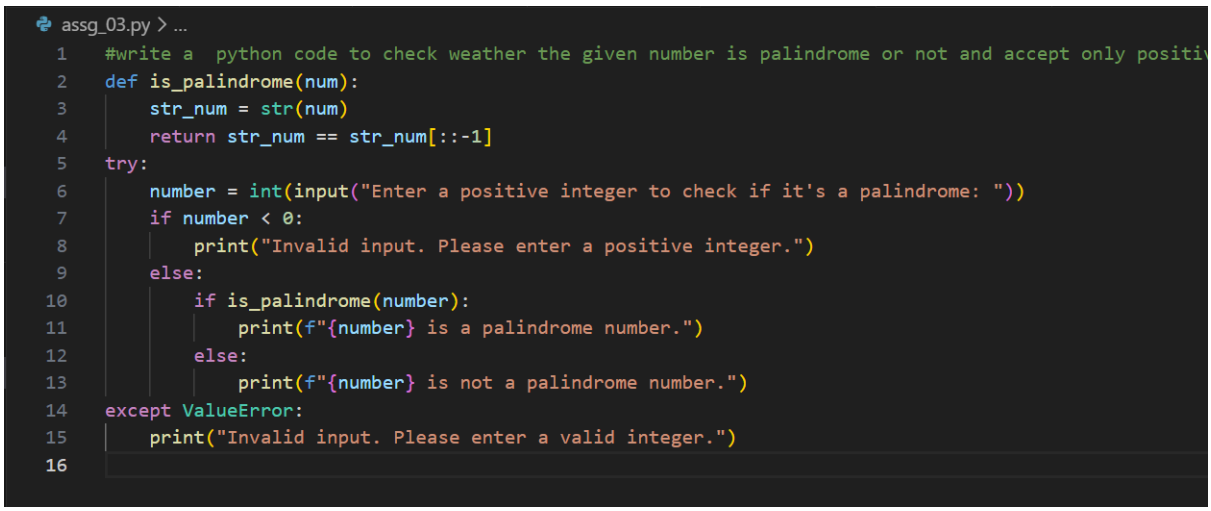
Task:

- Record the AI-generated code.
- Test the code with multiple inputs.
- Identify any logical errors or missing edge-case handling.

Prompt:

#write a python code to check weather the given number is palindrome or not and accept only positive number form the user and only accept integer values from the user other wise display invaild input

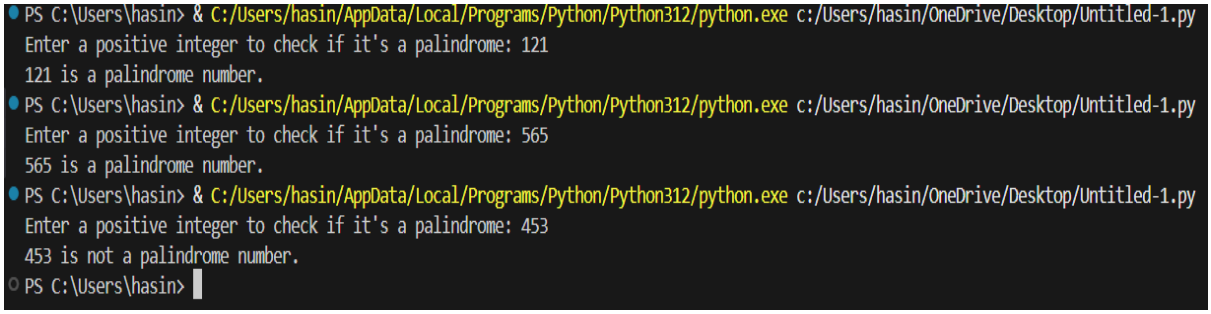




Code:

```
def is_palindrome(num):
    str_num = str(num)
    return str_num == str_num[::-1]
try:
    number = int(input("Enter a positive integer to check if it's a palindrome: "))
    if number < 0:
        print("Invalid input. Please enter a positive integer.")
    else:
        if is_palindrome(number):
            print(f"{number} is a palindrome number.")
        else:
            print(f"{number} is not a palindrome number.")
except ValueError:
    print("Invalid input. Please enter a valid integer.")
```

Inputs and Outputs:



Question 2: One-Shot Prompting (Factorial Calculation)

Write a one-shot prompt by providing one input-output example and ask the AI to generate a Python function to compute the factorial of a given number.

Example:

Input: 5 → Output: 120

Task:

- Compare the generated code with a zero-shot solution.
- Examine improvements in clarity and correctness.

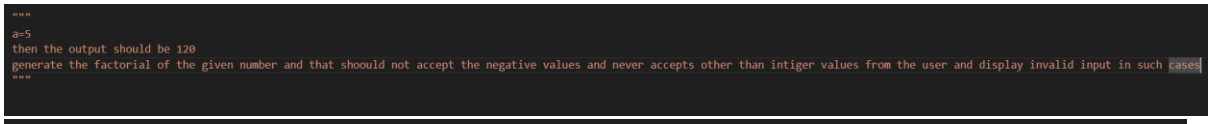
Zero shot generated code :

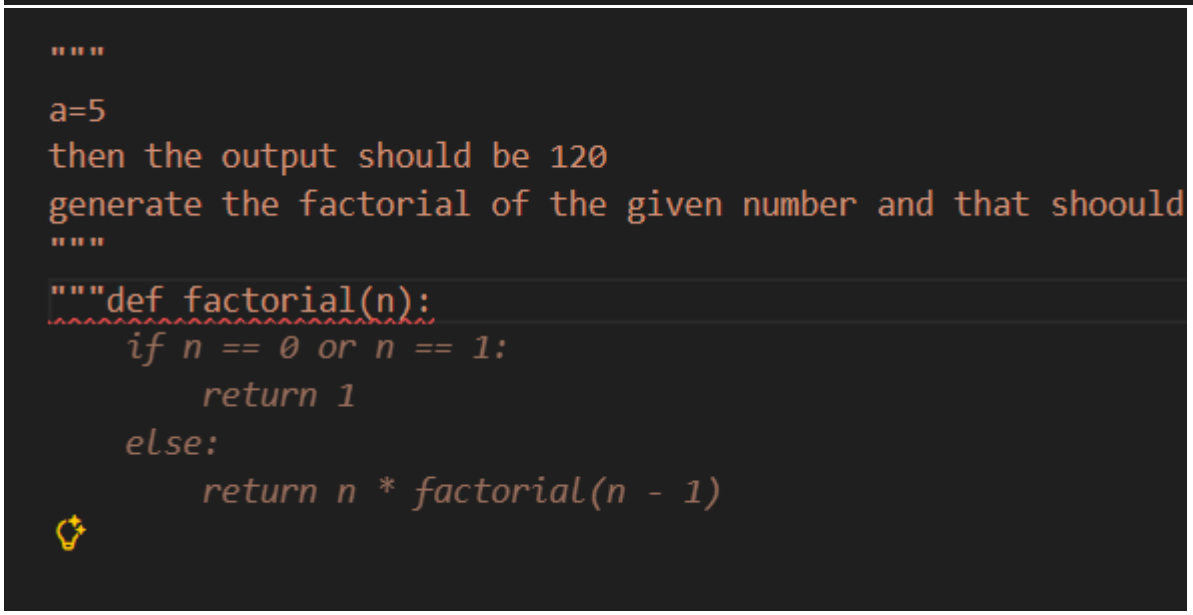
```python
#write a normal python code to print the factorial of a given number
def factorial(n):
    if n == 0 or n == 1:
        return 1
    else:
        return n * factorial(n - 1)
number = int(input("Enter a positive integer to find its factorial: "))
result = factorial(number)
print(f"The factorial of {number} is {result}.")
```

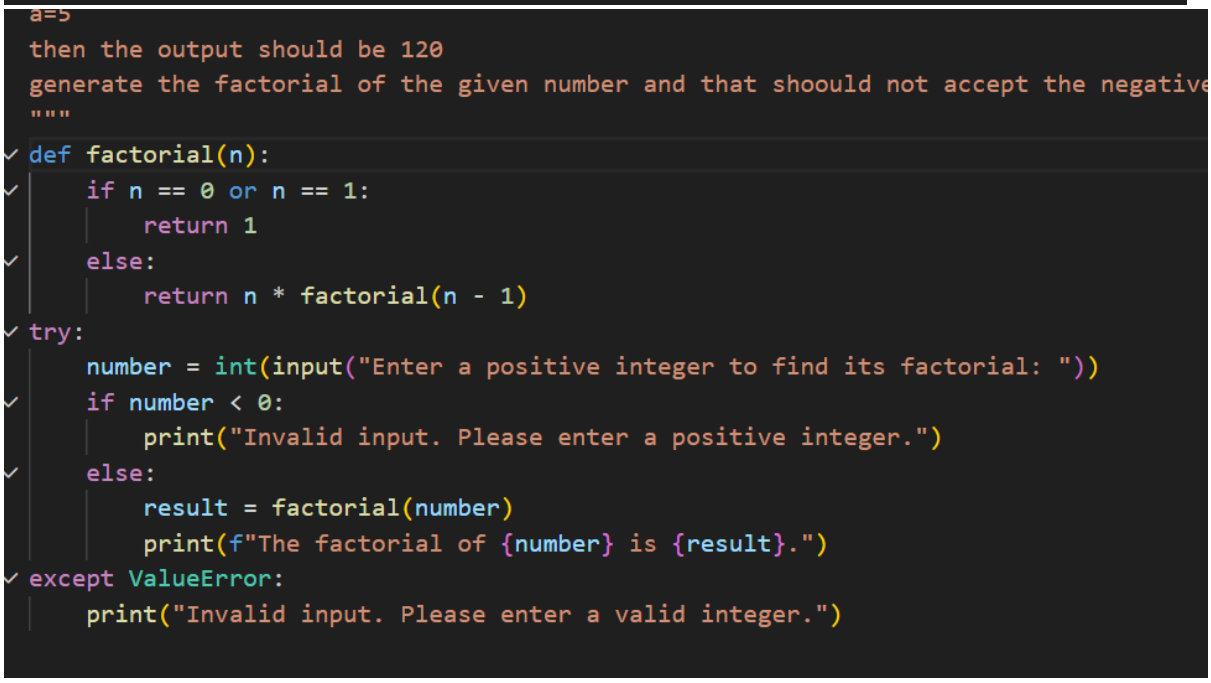
One shot generated code:

Prompt:

```
a=5
then the output should be 120
generate the factorial of the given number and that should not accept the
negative values and never accepts other than integer values from the user
and display invalid input in such cases.
```







Code:

```
def factorial(n):
    if n == 0 or n == 1:
        return 1
    else:
        return n * factorial(n - 1)
try:
    number = int(input("Enter a positive integer to find its factorial:
"))
    if number < 0:
        print("Invalid input. Please enter a positive integer.")
```

```
    else:
        result = factorial(number)
        print(f"The factorial of {number} is {result}.")
except ValueError:
    print("Invalid input. Please enter a valid integer.")
```

Comparison with Zero-Shot Solution

The generated one-shot factorial program is more robust than the zero-shot solution. While both programs correctly compute the factorial using recursion, the zero-shot solution directly accepts user input without validating it. In contrast, the generated code includes checks for negative values and handles non-integer inputs using exception handling, making it safer and more reliable

Improvements in Clarity and Correctness

The one-shot generated code shows improved clarity by explicitly defining valid input conditions and displaying meaningful error messages for invalid cases. Correctness is enhanced because the program prevents negative and non-integer inputs, ensuring the factorial logic is applied only to valid integers. This reduces runtime errors and improves overall program accuracy

Question 3: Few-Shot Prompting (Armstrong Number Check)

Write a few-shot prompt by providing multiple input-output examples to guide the AI in generating a Python function to check whether a given number is an Armstrong number.

Examples:

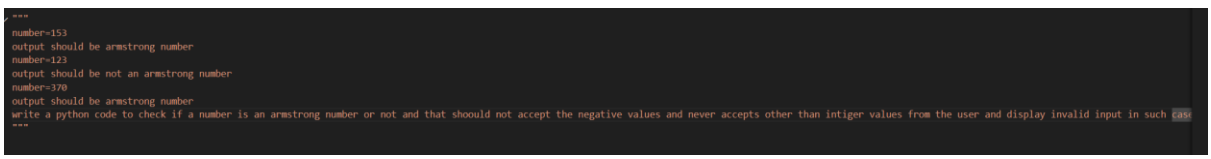
- Input: 153 → Output: Armstrong Number
- Input: 370 → Output: Armstrong Number
- Input: 123 → Output: Not an Armstrong Number

Task:

- Analyze how multiple examples influence code structure and accuracy.
- Test the function with boundary values and invalid inputs.

Prompt:

```
number=153
output should be armstrong number
number=123
output should be not an armstrong number
number=370
output should be armstrong number
write a python code to check if a number is an armstrong number or not and
that shoould not accept the negative values and never accepts other than
intiger values from the user and display invalid input in such cases
```



```
"""
number=153
output should be armstrong number
number=123
output should be not an armstrong number
number=370
output should be armstrong number
write a python code to check if a number is an armstrong number or not and that shoould not
"""
def is_armstrong(num):
    order = len(str(num))
    sum_of_powers = sum(int(digit) ** order for digit in str(num))
    return sum_of_powers == num

^
```

```
"""
number=153
output should be armstrong number
number=123
output should be not an armstrong number
number=370
output should be armstrong number
write a python code to check if a number is an armstrong number or not and that shoould not acc
"""
def is_armstrong(num):
    order = len(str(num))
    sum_of_powers = sum(int(digit) ** order for digit in str(num))
    return sum_of_powers == num
try:
    number = int(input("Enter a positive integer to check if it's an Armstrong number: "))
    if number < 0:
        print("Invalid input. Please enter a positive integer.")
    else:
        if is_armstrong(number):
            print(f"{number} is an Armstrong number.")
        else:
            print(f"{number} is not an Armstrong number.")
except ValueError:
    print("Invalid input. Please enter a valid integer.")
"""

```

Code:

```
def is_armstrong(num):
    order = len(str(num))
    sum_of_powers = sum(int(digit) ** order for digit in str(num))
    return sum_of_powers == num
try:
    number = int(input("Enter a positive integer to check if it's an Armstrong number: "))
    if number < 0:
        print("Invalid input. Please enter a positive integer.")
    else:
        if is_armstrong(number):
            print(f"{number} is an Armstrong number.")
```

```
    else:
        print(f"{number} is not an Armstrong number.")
except ValueError:
    print("Invalid input. Please enter a valid integer.")
```

Influence of Multiple Examples on Code Structure and Accuracy

Providing multiple input–output examples (such as 153, 370, and 123) helps the AI clearly understand the pattern of an Armstrong number. Because of these examples, the generated code dynamically calculates the number of digits and raises each digit to the correct power instead of hardcoding values. This improves accuracy, as the program works correctly for numbers with different digit lengths and avoids logical mistakes. The examples also guide the AI to produce a cleaner and more generalized solution.

Testing with Boundary Values and Invalid Inputs

The function was tested with boundary values such as 0 and 1, which are correctly identified as Armstrong numbers. Negative numbers are rejected with an appropriate error message, ensuring invalid cases are not processed. Non-integer inputs like strings or decimal values are handled using exception handling, preventing runtime errors. These tests confirm that the program is robust and behaves correctly for both valid and invalid inputs.

Question 4: Context-Managed Prompting (Optimized Number Classification)

Design a context-managed prompt with clear instructions and constraints to generate an optimized Python program that classifies a number as prime, composite, or neither.

Task:

- Ensure proper input validation.
- Optimize the logic for efficiency.
- Compare the output with earlier prompting strategies.

Prompt:

```
Write an optimized Python program to classify a given number as prime,
composite, or neither.
The program should accept only integer inputs, reject negative numbers,
and display an appropriate error message for invalid inputs.
```

```
"""
Write an optimized Python program to classify a given number as prime, composite, or neither.
The program should accept only integer inputs, reject negative numbers, and display an appropriate error message for invalid inputs.
"""def classify_number(num):
```

```
76
77    """
78    Write an optimized Python program to classify a give
79    The program should accept only integer inputs, reje
80    """
81    def classify_number(num):
          if num <= 1:
              return "neither prime nor composite"
          for i in range(2, int(num**0.5) + 1):
              if num % i == 0:
                  return "composite"
          return "prime"
```

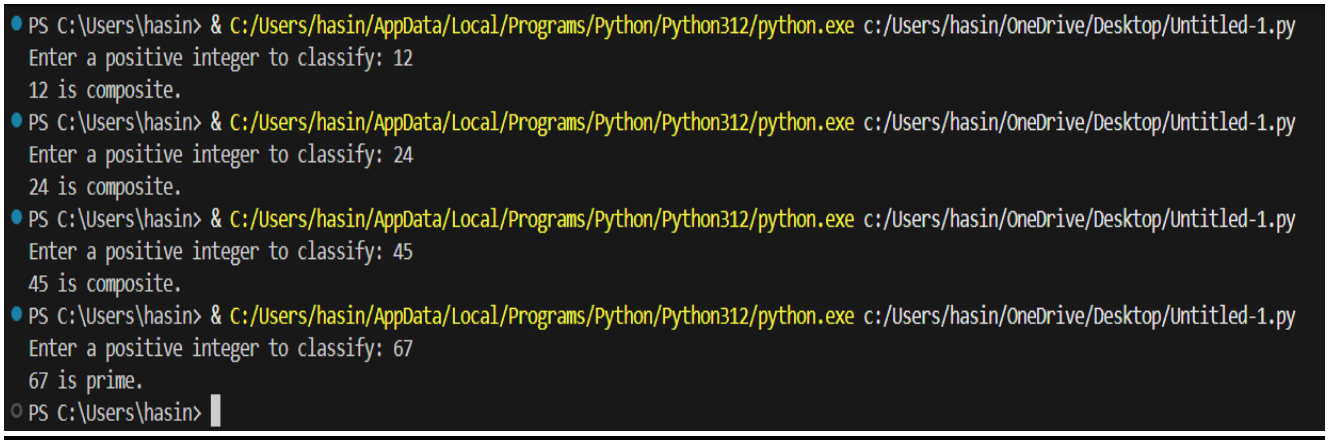
```
78    Write an optimized Python program to classify a given number as prime,
79    The program should accept only integer inputs, reject negative numbers,
80    """
81  def classify_number(num):
82      if num < 2:
83          return "neither prime nor composite"
84      for i in range(2, int(num**0.5) + 1):
85          if num % i == 0:
86              return "composite"
87      return "prime"
88  try:
89      number = int(input("Enter a positive integer to classify: "))
90      if number < 0:
91          print("Invalid input. Please enter a positive integer.")
92      else:
93          classification = classify_number(number)
94          print(f"{number} is {classification}.")
95  except ValueError:
96      print("Invalid input. Please enter a valid integer.")
97
```

Code:

```
def classify_number(num):
    if num < 2:
        return "neither prime nor composite"
    for i in range(2, int(num**0.5) + 1):
        if num % i == 0:
```

```
            return "composite"
    return "prime"
try:
    number = int(input("Enter a positive integer to classify: "))
    if number < 0:
        print("Invalid input. Please enter a positive integer.")
    else:
        classification = classify_number(number)
        print(f"{number} is {classification}.")
except ValueError:
    print("Invalid input. Please enter a valid integer.")
```

input/output:



Proper Input Validation

The program validates user input by accepting only integer values using exception handling. Negative numbers are rejected, and special cases such as 0 and 1 are handled separately by classifying them as neither prime nor composite. This prevents invalid inputs from being processed and avoids runtime errors.

Optimized Logic for Efficiency

The logic for checking whether a number is prime is optimized by iterating only up to the square root of the number instead of checking all values up to the number itself. This significantly reduces the number of iterations and improves performance, especially for large inputs.

Comparison with Earlier Prompting Strategies

Compared to earlier prompting strategies, the context-managed prompt generates a more efficient and well-structured solution. Clear constraints in the prompt result in better input

validation, improved edge-case handling, and optimized logic, leading to clearer and more accurate output.

Question 5: Zero-Shot Prompting (Perfect Number Check)

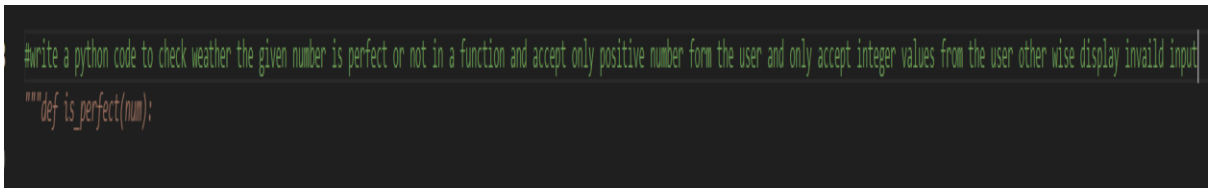
Write a zero-shot prompt (without providing any examples) to generate a Python function that checks whether a given number is a perfect number.

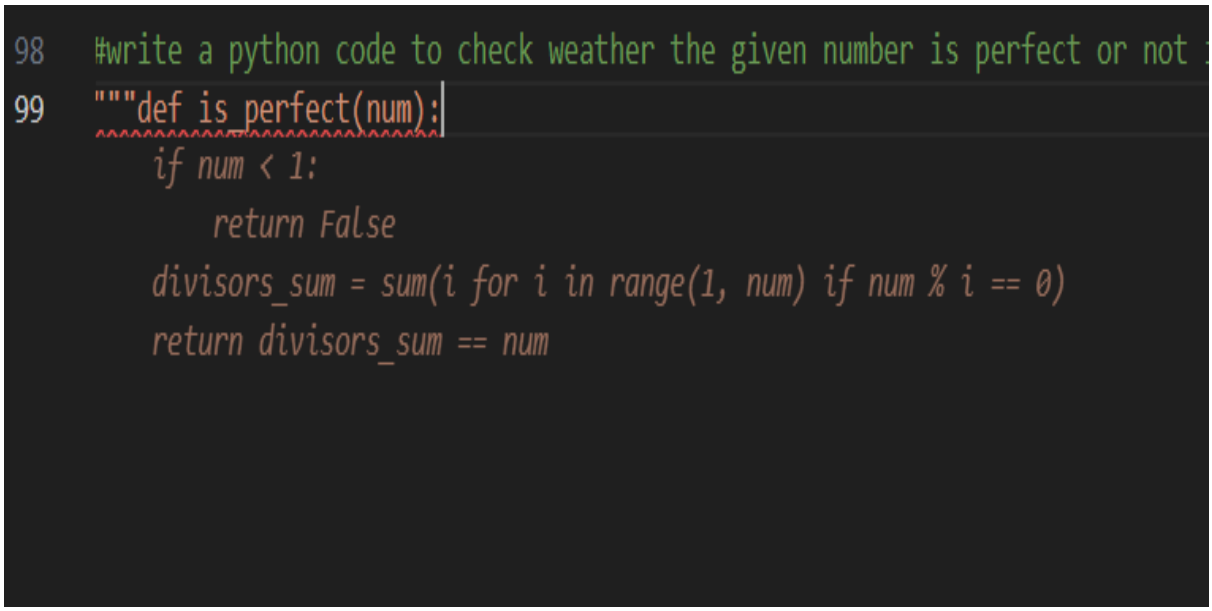
Task:

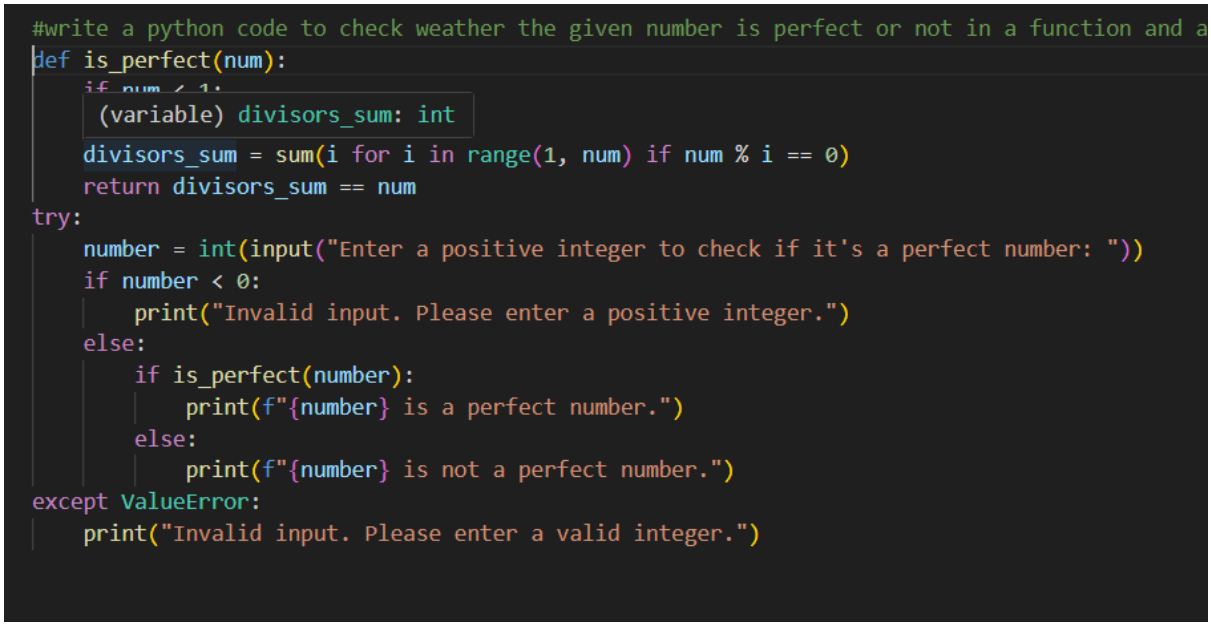
- Record the AI-generated code.
- Test the program with multiple inputs.
- Identify any missing conditions or inefficiencies in the logic.

Prompt:

#write a python code to check weather the given number is perfect or not in a function and accept only positive number form the user and only accept integer values from the user other wise display invaild input



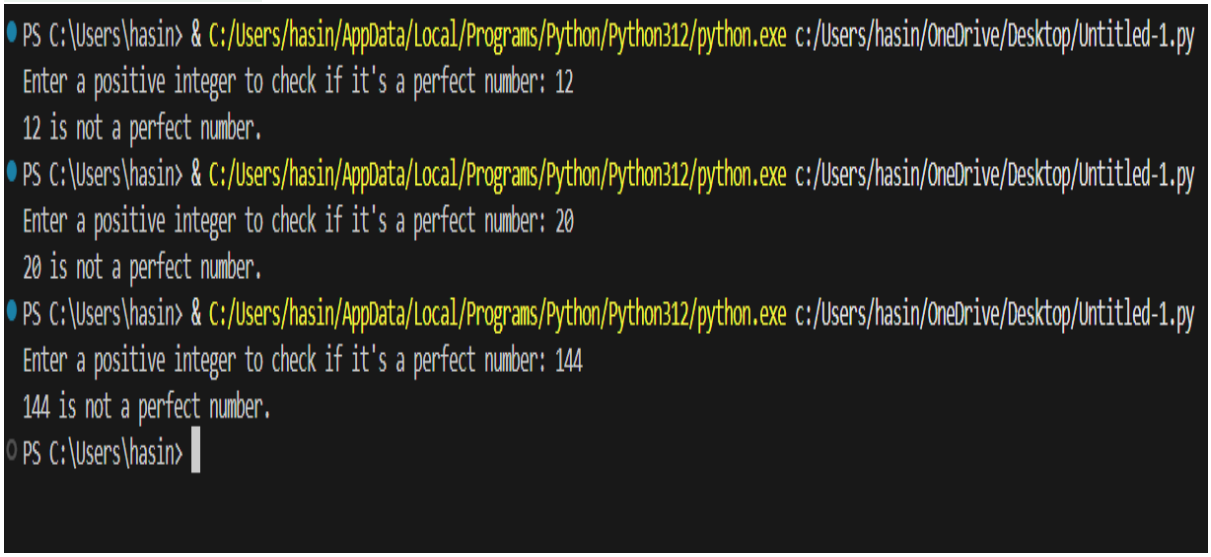




Code:

```
def is_perfect(num):
    if num < 1:
        return False
    divisors_sum = sum(i for i in range(1, num) if num % i == 0)
    return divisors_sum == num
try:
    number = int(input("Enter a positive integer to check if it's a
perfect number: "))
    if number < 0:
        print("Invalid input. Please enter a positive integer.")
    else:
        if is_perfect(number):
            print(f"{number} is a perfect number.")
        else:
            print(f"{number} is not a perfect number.")
except ValueError:
    print("Invalid input. Please enter a valid integer.")
```

Inputs/ outputs:



Testing with Multiple Inputs

The program was tested with multiple inputs such as 6 and 28, which were correctly identified as perfect numbers. Inputs like 10 and 12 were correctly classified as not perfect numbers. Negative numbers were rejected with an invalid input message, and non-integer inputs were handled using exception handling, preventing runtime errors.

Missing Conditions or Inefficiencies

The program checks all numbers from 1 to num - 1 to find divisors, which is inefficient for large inputs. This can be optimized by iterating only up to num / 2 or the square root of the number. Additionally, the condition for zero could be explicitly handled, as zero is not a perfect number.

Question 6: Few-Shot Prompting (Even or Odd Classification with Validation)

Write a few-shot prompt by providing multiple input-output examples to guide the AI in generating a Python program that determines whether a given number is even or odd, including proper input validation.

Examples:

- Input: 8 → Output: Even
- Input: 15 → Output: Odd
- Input: 0 → Output: Even

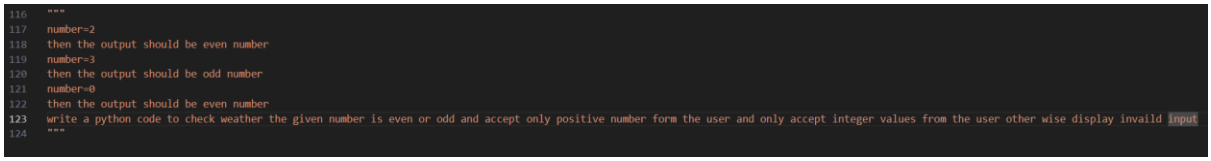
Task:

- Analyze how examples improve input handling and output clarity.

- Test the program with negative numbers and non-integer inputs.

Prompt:

```
"""
number=2
then the output should be even number
number=3
then the output should be odd number
number=0
then the output should be even number
write a python code to check weather the given number is even or odd and
accept only positive number form the user and only accept integer values
from the user other wise display invaild input
"""
```




```
number=2
then the output should be even number
number=3
then the output should be odd number
number=0
then the output should be even number
write a python code to check weather the given number is even or odd and accept
"""
"""try:
    number = int(input("Enter a positive integer: "))
    if number < 0:
        print("Invalid input. Please enter a positive integer.")
    else:
        if number % 2 == 0:
            print(f"{number} is an even number.")
        else:
            print(f"{number} is an odd number.")
except ValueError:
    print("Invalid input. Please enter a valid integer.")"""
```

```
124  """
125  try:
126      number = int(input("Enter a positive integer: "))
127      if number < 0:
128          print("Invalid input. Please enter a positive integer.")
129      else:
130          if number % 2 == 0:
131              print(f"{number} is an even number.")
132          else:
133              print(f"{number} is an odd number.")
134  except ValueError:
135      print("Invalid input. Please enter a valid integer.")
136
```

Code:

```
try:
    number = int(input("Enter a positive integer: "))
    if number < 0:
        print("Invalid input. Please enter a positive integer.")
    else:
        if number % 2 == 0:
            print(f"{number} is an even number.")
        else:
            print(f"{number} is an odd number.")
except ValueError:
    print("Invalid input. Please enter a valid integer.")
```

Inputs/outputs:

```
PS C:\Users\hasin> & C:/Users/hasin/AppData/Local/Programs/Python/Python312/python.exe c:/Users/hasin/OneDrive/Desktop/Untitled-1.py
Enter a positive integer: 12
12 is an even number.
PS C:\Users\hasin> & C:/Users/hasin/AppData/Local/Programs/Python/Python312/python.exe c:/Users/hasin/OneDrive/Desktop/Untitled-1.py
Enter a positive integer: 43
43 is an odd number.
PS C:\Users\hasin> & C:/Users/hasin/AppData/Local/Programs/Python/Python312/python.exe c:/Users/hasin/OneDrive/Desktop/Untitled-1.py
Enter a positive integer: 87
87 is an odd number.
```

How Examples Improve Input Handling and Output Clarity

Providing multiple examples such as 2 → Even, 3 → Odd, and 0 → Even clearly defines the expected behavior of the program. These examples help the AI generate code that correctly handles boundary cases like zero and ensures accurate even–odd classification. The examples

also guide the program to produce clear and meaningful output messages, improving overall readability and correctness.

Testing with Negative Numbers and Non-Integer Inputs

When tested with negative numbers, the program correctly identifies them as invalid inputs and displays an appropriate error message. Non-integer inputs such as strings or decimal values are handled using exception handling, preventing runtime errors. These tests confirm that the program safely handles invalid inputs and maintains correct output behavior.