AI ASSISTANT CODING
ASSIGNMENT-3.1

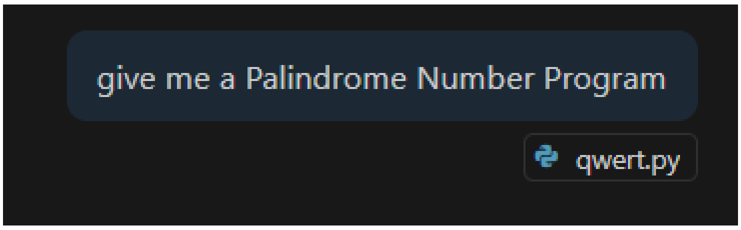
Question 1: Zero-Shot Prompting (Palindrome Number Program)

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

Task:

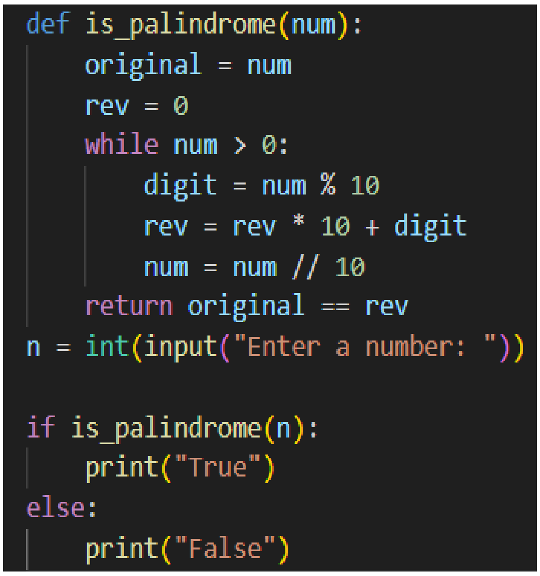
- Record the AI-generated code.
- Test the code with multiple inputs.
- Identify any logical errors or missing edge-case handling.

PROMPT:



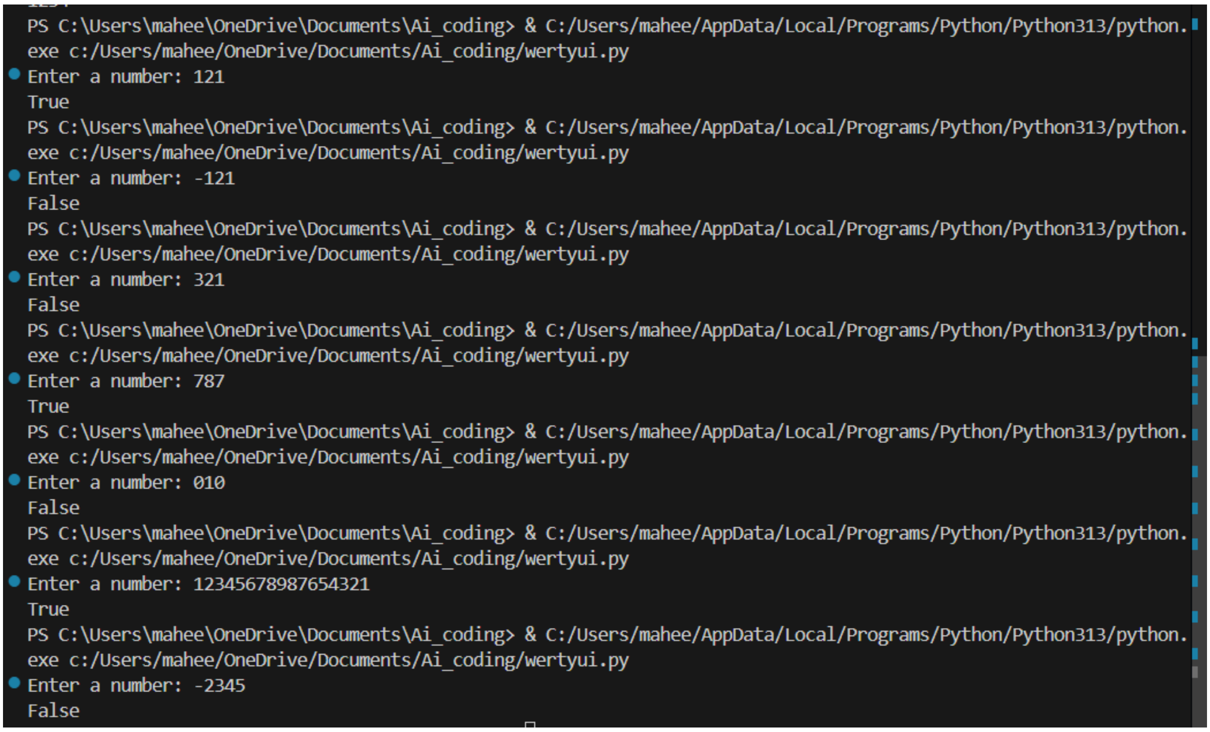
give me a Palindrome Number Program

AI-generated code:



```python
def is_palindrome(num):
    original = num
    rev = 0
    while num > 0:
        digit = num % 10
        rev = rev * 10 + digit
        num = num // 10
    return original == rev
n = int(input("Enter a number: "))

if is_palindrome(n):
    print("True")
else:
    print("False")
```

Test the code with multiple inputs:



```
PS C:\Users\mahee\OneDrive\Documents\Ai_coding> & C:/Users/mahee/AppData/Local/Programs/Python/Python313/python.
exe c:/Users/mahee/OneDrive/Documents/Ai_coding/wertyui.py
Enter a number: 121
True
PS C:\Users\mahee\OneDrive\Documents\Ai_coding> & C:/Users/mahee/AppData/Local/Programs/Python/Python313/python.
exe c:/Users/mahee/OneDrive/Documents/Ai_coding/wertyui.py
Enter a number: -121
False
PS C:\Users\mahee\OneDrive\Documents\Ai_coding> & C:/Users/mahee/AppData/Local/Programs/Python/Python313/python.
exe c:/Users/mahee/OneDrive/Documents/Ai_coding/wertyui.py
Enter a number: 321
False
PS C:\Users\mahee\OneDrive\Documents\Ai_coding> & C:/Users/mahee/AppData/Local/Programs/Python/Python313/python.
exe c:/Users/mahee/OneDrive/Documents/Ai_coding/wertyui.py
Enter a number: 787
True
PS C:\Users\mahee\OneDrive\Documents\Ai_coding> & C:/Users/mahee/AppData/Local/Programs/Python/Python313/python.
exe c:/Users/mahee/OneDrive/Documents/Ai_coding/wertyui.py
Enter a number: 010
False
PS C:\Users\mahee\OneDrive\Documents\Ai_coding> & C:/Users/mahee/AppData/Local/Programs/Python/Python313/python.
exe c:/Users/mahee/OneDrive/Documents/Ai_coding/wertyui.py
Enter a number: 12345678987654321
True
PS C:\Users\mahee\OneDrive\Documents\Ai_coding> & C:/Users/mahee/AppData/Local/Programs/Python/Python313/python.
exe c:/Users/mahee/OneDrive/Documents/Ai_coding/wertyui.py
Enter a number: -2345
False
```

Identify any logical errors or missing edge-case handling:

1. Negative Numbers Not Handled

Logical Error:

Many programs don’t check for negative numbers.

```
if num < 0:
    return False
```

Why?

- -121 becomes 121- when reversed → not equal
- So all negative numbers are NOT palindromes

2. Numbers Ending with Zero (Except 0)

Missing Edge Case:

Numbers like 10, 20, 100 are often wrongly considered.

Example:

10 → reversed = 01 = 1

So:

```
if num % 10 == 0 and num != 0:
    return False
```

3. Zero (0) Case

Sometimes forgotten:

0 → reversed = 0 (Palindrome)

So 0 is a valid palindrome and must be handled properly.

4. Modifying Original Number Without Storing

Logical Error:

If you do:

```
while num > 0:
    ...
```

and then compare:

```
if num == rev:   # WRONG
```

Because num is now 0, you lost the original value.

Correct:

```
original = num
...
```

```
return original == rev
```

5. Not Handling Single-Digit Numbers

Missing Edge Case:

Single digits like:

1, 2, 7, 9

are always palindromes.

But some logic fails if not considered.

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:

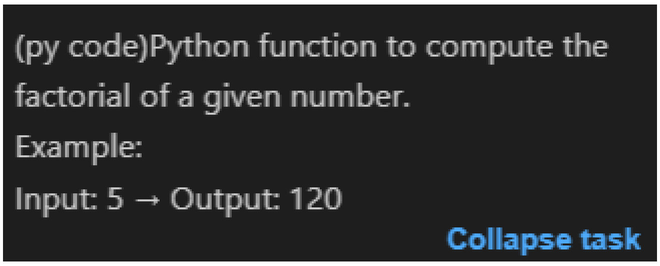
Input: 5 → Output: 120

Task:

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

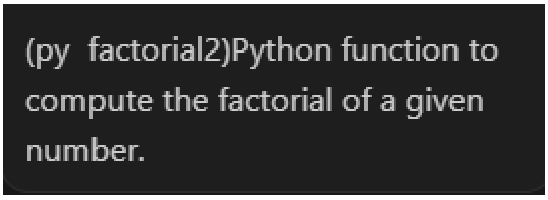
PROMPT:

One-Shot Prompt



(py code)Python function to compute the factorial of a given number.
Example:
Input: 5 → Output: 120

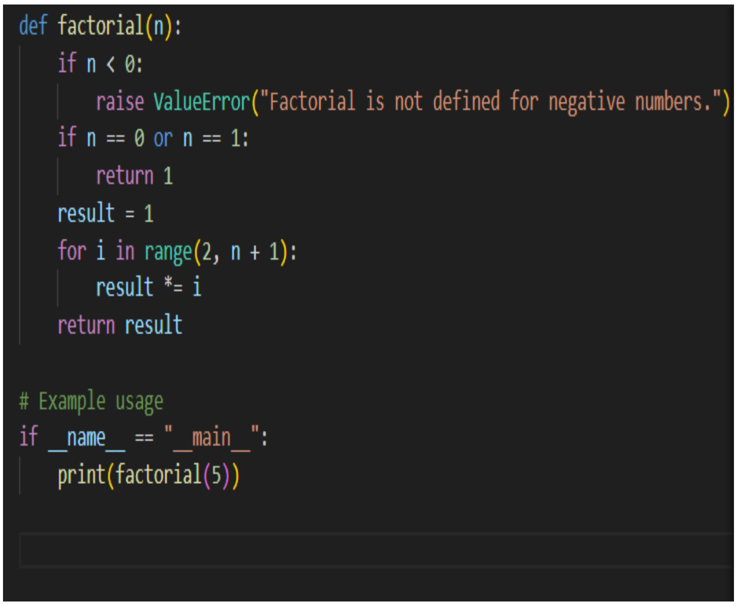
Zero-Shot Prompt:



(py factorial2)Python function to compute the factorial of a given number.

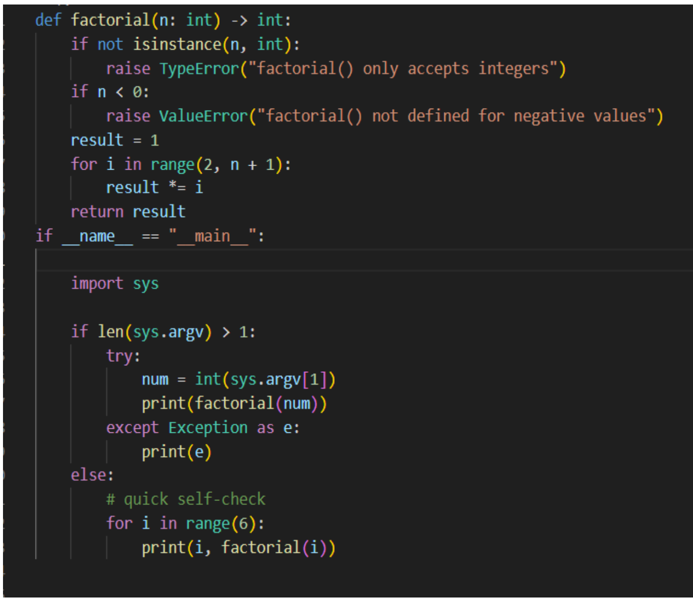
Task:

One-Shot Code:



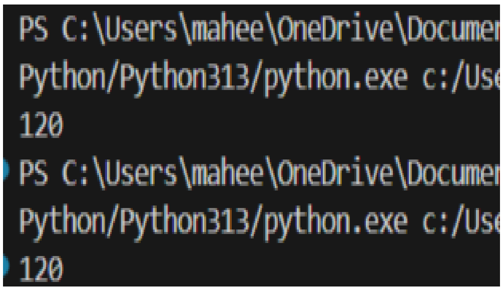
```python
def factorial(n):
    if n < 0:
        raise ValueError("Factorial is not defined for negative numbers.")
    if n == 0 or n == 1:
        return 1
    result = 1
    for i in range(2, n + 1):
        result *= i
    return result


# Example usage
if __name__ == "__main__":
    print(factorial(5))
```

Zero-Shot Code:



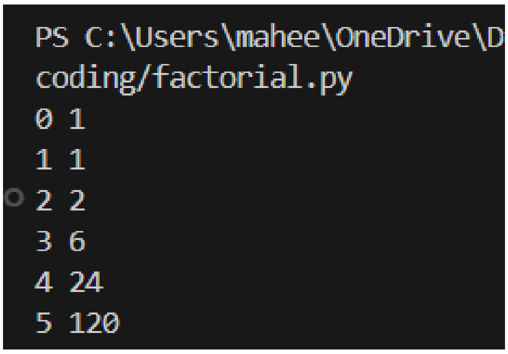
```python
def factorial(n: int) -> int:
    if not isinstance(n, int):
        raise TypeError("factorial() only accepts integers")
    if n < 0:
        raise ValueError("factorial() not defined for negative values")
    result = 1
    for i in range(2, n + 1):
        result *= i
    return result
if __name__ == "__main__":

    import sys

    if len(sys.argv) > 1:
        try:
            num = int(sys.argv[1])
            print(factorial(num))
        except Exception as e:
            print(e)
    else:
        # quick self-check
        for i in range(6):
            print(i, factorial(i))
```

One – Shot Expected Output:



```
PS C:\Users\mahee\OneDrive\Docume
Python/Python313/python.exe c:/Use
120
PS C:\Users\mahee\OneDrive\Docume
Python/Python313/python.exe c:/Use
120
```

Zero – Shot Expected Output:



```
PS C:\Users\mahee\OneDrive\D
coding/factorial.py
0 1
1 1
2 2
3 6
4 24
5 120
```

1. Clarity Improvements

Function Definition

One-Shot Code

- Simple and easy to read
- No type hints, so expected input/output type is implicit

Zero-Shot Code

- Uses type hints:
- def factorial(n: int) -> int:
- This clearly communicates:
  - Input must be an integer
  - Output will be an integer
    Improvement in clarity

Error Messages

One-Shot Code

raise ValueError("Factorial is not defined for negative numbers.")

- Handles negative input clearly
- Does not handle non-integer input

Zero-Shot Code

```
if not isinstance(n, int):
    raise TypeError("factorial() only accepts integers")
if n < 0:
    raise ValueError("factorial() not defined for negative values")
```

- Separates type errors and value errors
- Error messages are precise and descriptive
  Improved clarity and robustness

Code Usage & Testing

One-Shot Code

```
print(factorial(5))
```

- Demonstrates only one example
- Less flexible for testing

Zero-Shot Code

- Accepts command-line input
- Includes a self-check loop

```
for i in range(6):
    print(i, factorial(i))
```

- Helps verify correctness for multiple values
  Much clearer for testing and reuse

---

2. Correctness Improvements

Edge Case Handling

Case	One-Shot Code	Zero-Shot Code
Negative numbers	✔ Handled	✔ Handled
Zero	✔ Returns 1	✔ Returns 1

Case	One-Shot Code	Zero-Shot Code
One	✔ Returns 1	✔ Returns 1
Non-integer input	✕ Not handled	✔ Properly handled

Zero-Shot Code is more correct overall

Defensive Programming

Zero-Shot Code uses:

- Type checking
- Exception handling
- Input validation from command line

This prevents:

- Silent failures
- Unexpected runtime behavior

  Improved correctness and safety

---

3. Readability Comparison

One-Shot Code

✔ Beginner-friendly
✔ Short and clean
✕ Less robust

Zero-Shot Code

✔ Professional-level structure
✔ Better documentation via type hints
✔ Suitable for real-world usage
✕ Slightly more complex for beginners

---

4. Final Conclusion

Overall Assessment:

Aspect	**Better Version**
Simplicity	One-Shot
Clarity	Zero-Shot
Error Handling	Zero-Shot
Correctness	Zero-Shot
Reusability	Zero-Shot

Final Verdict

The Zero-Shot Code shows significant improvements in clarity, correctness, and robustness, making it more suitable for real-world applications, while the One-Shot Code is better for quick demonstrations or beginners.

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.

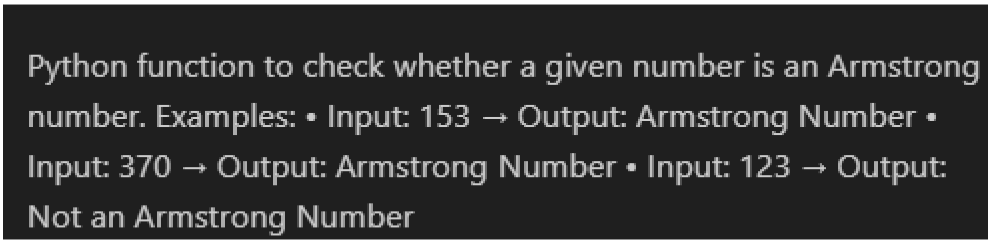
Examples:

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

Task:

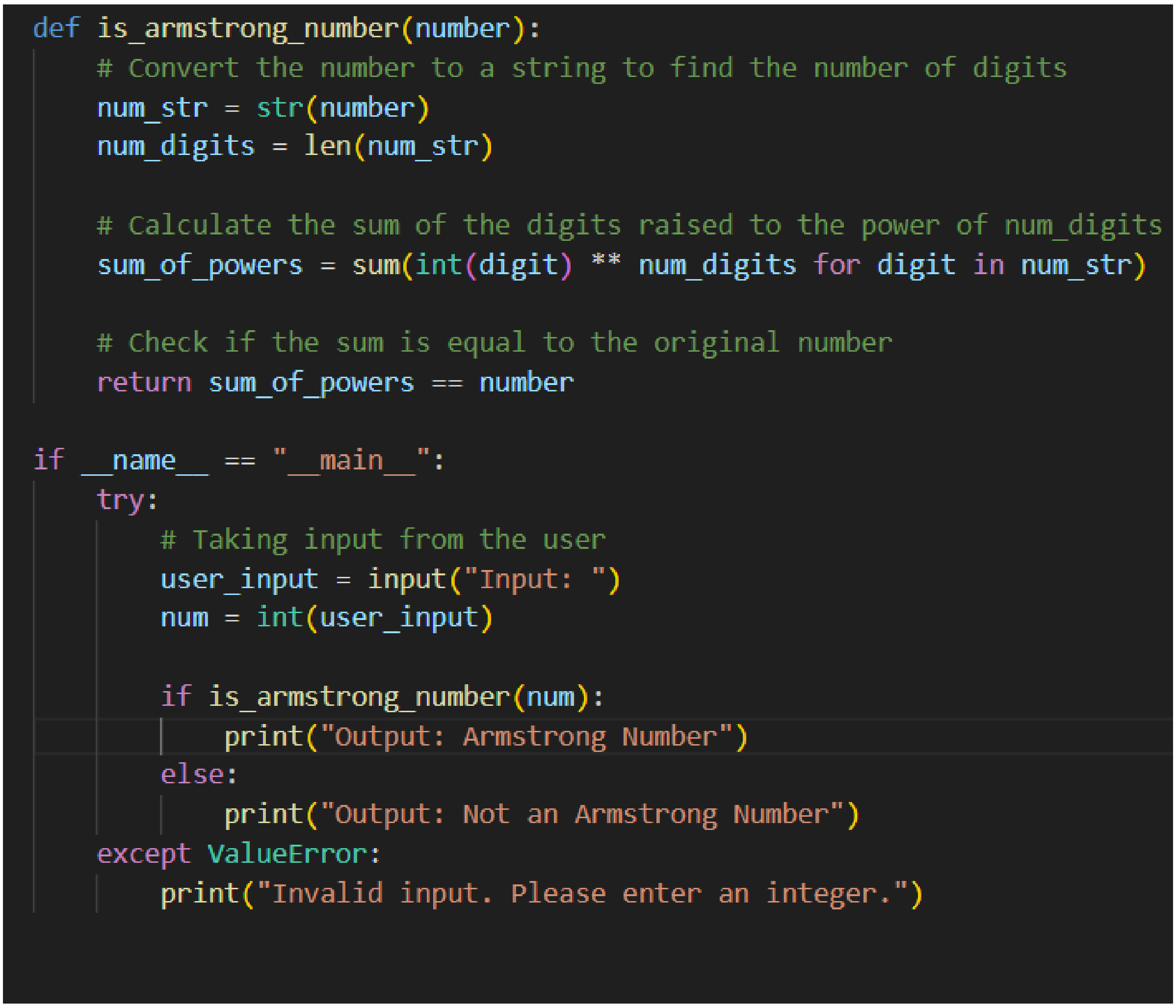
- Analyze how multiple examples influence code structure and accuracy.
- Test the function with boundary values and invalid inputs.

PROMPT:



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

CODE:

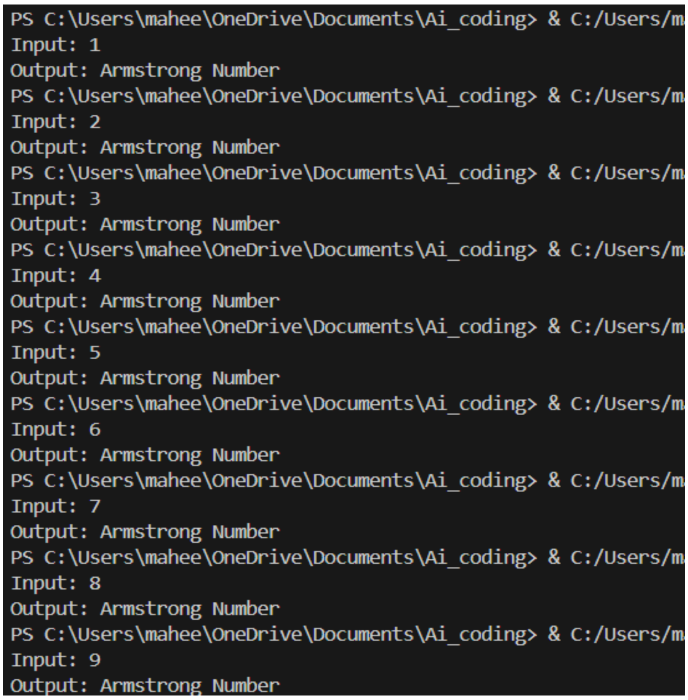


```python
def is_armstrong_number(number):
    # Convert the number to a string to find the number of digits
    num_str = str(number)
    num_digits = len(num_str)

    # Calculate the sum of the digits raised to the power of num_digits
    sum_of_powers = sum(int(digit) ** num_digits for digit in num_str)

    # Check if the sum is equal to the original number
    return sum_of_powers == number

if __name__ == "__main__":
    try:
        # Taking input from the user
        user_input = input("Input: ")
        num = int(user_input)

        if is_armstrong_number(num):
            print("Output: Armstrong Number")
        else:
            print("Output: Not an Armstrong Number")
    except ValueError:
        print("Invalid input. Please enter an integer.")
```

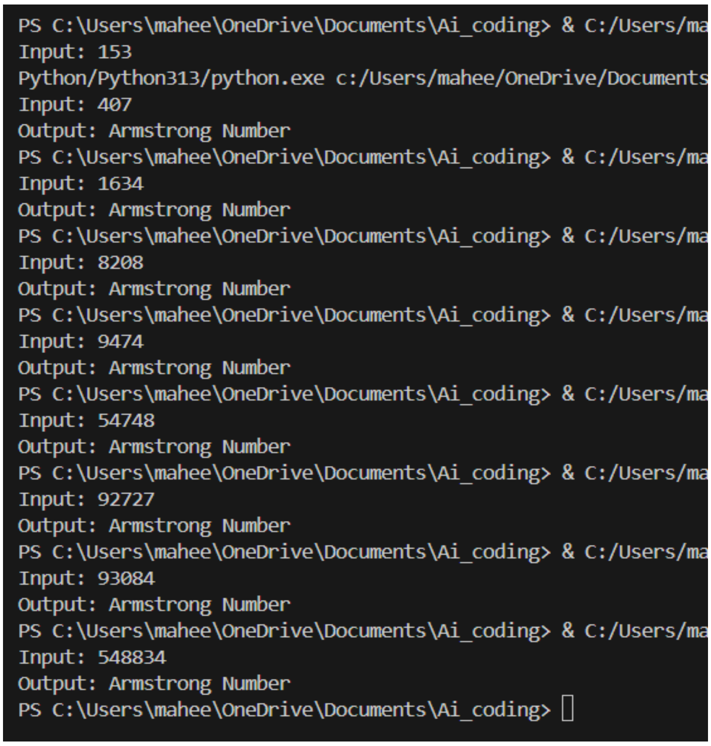
TASK:

Analyze how multiple examples influence code structure and accuracy.

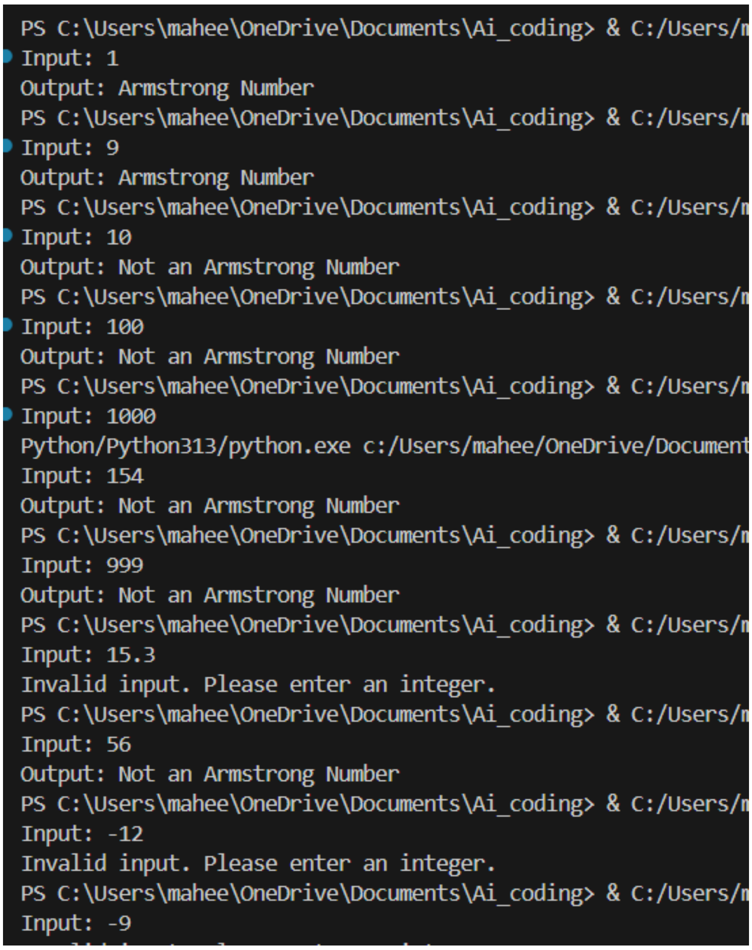
Test the function with boundary values and invalid inputs.



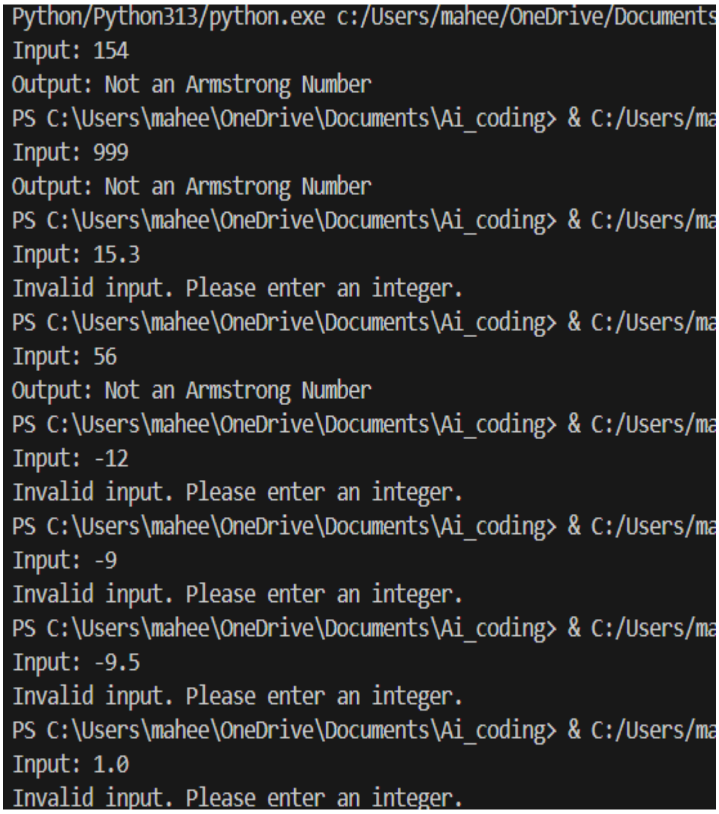
```
PS C:\Users\mahee\OneDrive\Documents\Ai_coding> & C:/Users/m
Input: 1
Output: Armstrong Number
PS C:\Users\mahee\OneDrive\Documents\Ai_coding> & C:/Users/m
Input: 2
Output: Armstrong Number
PS C:\Users\mahee\OneDrive\Documents\Ai_coding> & C:/Users/m
Input: 3
Output: Armstrong Number
PS C:\Users\mahee\OneDrive\Documents\Ai_coding> & C:/Users/m
Input: 4
Output: Armstrong Number
PS C:\Users\mahee\OneDrive\Documents\Ai_coding> & C:/Users/m
Input: 5
Output: Armstrong Number
PS C:\Users\mahee\OneDrive\Documents\Ai_coding> & C:/Users/m
Input: 6
Output: Armstrong Number
PS C:\Users\mahee\OneDrive\Documents\Ai_coding> & C:/Users/m
Input: 7
Output: Armstrong Number
PS C:\Users\mahee\OneDrive\Documents\Ai_coding> & C:/Users/m
Input: 8
Output: Armstrong Number
PS C:\Users\mahee\OneDrive\Documents\Ai_coding> & C:/Users/m
Input: 9
Output: Armstrong Number
```



```
PS C:\Users\mahee\OneDrive\Documents\Ai_coding> & C:/Users/ma
Input: 153
Python/Python313/python.exe c:/Users/mahee/OneDrive/Documents
Input: 407
Output: Armstrong Number
PS C:\Users\mahee\OneDrive\Documents\Ai_coding> & C:/Users/ma
Input: 1634
Output: Armstrong Number
PS C:\Users\mahee\OneDrive\Documents\Ai_coding> & C:/Users/ma
Input: 8208
Output: Armstrong Number
PS C:\Users\mahee\OneDrive\Documents\Ai_coding> & C:/Users/ma
Input: 9474
Output: Armstrong Number
PS C:\Users\mahee\OneDrive\Documents\Ai_coding> & C:/Users/ma
Input: 54748
Output: Armstrong Number
PS C:\Users\mahee\OneDrive\Documents\Ai_coding> & C:/Users/ma
Input: 92727
Output: Armstrong Number
PS C:\Users\mahee\OneDrive\Documents\Ai_coding> & C:/Users/ma
Input: 93084
Output: Armstrong Number
PS C:\Users\mahee\OneDrive\Documents\Ai_coding> & C:/Users/ma
Input: 548834
Output: Armstrong Number
PS C:\Users\mahee\OneDrive\Documents\Ai_coding>
```



```
PS C:\Users\mahee\OneDrive\Documents\Ai_coding> & C:/Users/m
Input: 1
Output: Armstrong Number
PS C:\Users\mahee\OneDrive\Documents\Ai_coding> & C:/Users/m
Input: 9
Output: Armstrong Number
PS C:\Users\mahee\OneDrive\Documents\Ai_coding> & C:/Users/m
Input: 10
Output: Not an Armstrong Number
PS C:\Users\mahee\OneDrive\Documents\Ai_coding> & C:/Users/m
Input: 100
Output: Not an Armstrong Number
PS C:\Users\mahee\OneDrive\Documents\Ai_coding> & C:/Users/m
Input: 1000
Python/Python313/python.exe c:/Users/mahee/OneDrive/Document
Input: 154
Output: Not an Armstrong Number
PS C:\Users\mahee\OneDrive\Documents\Ai_coding> & C:/Users/m
Input: 999
Output: Not an Armstrong Number
PS C:\Users\mahee\OneDrive\Documents\Ai_coding> & C:/Users/m
Input: 15.3
Invalid input. Please enter an integer.
PS C:\Users\mahee\OneDrive\Documents\Ai_coding> & C:/Users/m
Input: 56
Output: Not an Armstrong Number
PS C:\Users\mahee\OneDrive\Documents\Ai_coding> & C:/Users/m
Input: -12
Invalid input. Please enter an integer.
PS C:\Users\mahee\OneDrive\Documents\Ai_coding> & C:/Users/m
Input: -9
```



```
Python/Python313/python.exe c:/Users/mahee/OneDrive/Documents
Input: 154
Output: Not an Armstrong Number
PS C:\Users\mahee\OneDrive\Documents\Ai_coding> & C:/Users/ma
Input: 999
Output: Not an Armstrong Number
PS C:\Users\mahee\OneDrive\Documents\Ai_coding> & C:/Users/ma
Input: 15.3
Invalid input. Please enter an integer.
PS C:\Users\mahee\OneDrive\Documents\Ai_coding> & C:/Users/ma
Input: 56
Output: Not an Armstrong Number
PS C:\Users\mahee\OneDrive\Documents\Ai_coding> & C:/Users/ma
Input: -12
Invalid input. Please enter an integer.
PS C:\Users\mahee\OneDrive\Documents\Ai_coding> & C:/Users/ma
Input: -9
Invalid input. Please enter an integer.
PS C:\Users\mahee\OneDrive\Documents\Ai_coding> & C:/Users/ma
Input: -9.5
Invalid input. Please enter an integer.
PS C:\Users\mahee\OneDrive\Documents\Ai_coding> & C:/Users/ma
Input: 1.0
Invalid input. Please enter an integer.
```

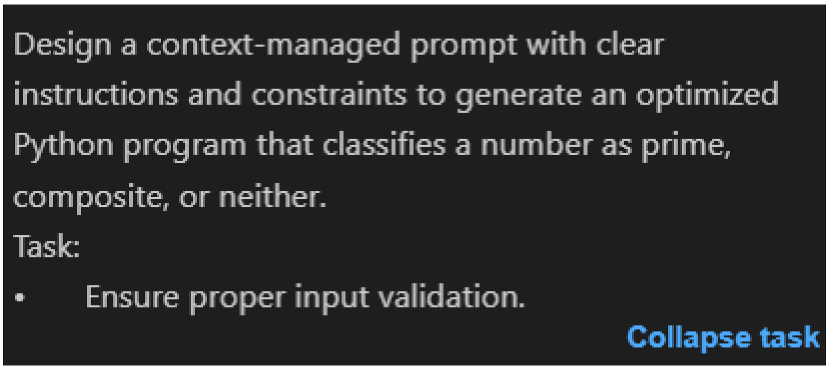
Question 4: Context-Managed Prompting (Optimized Number Classification)

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:



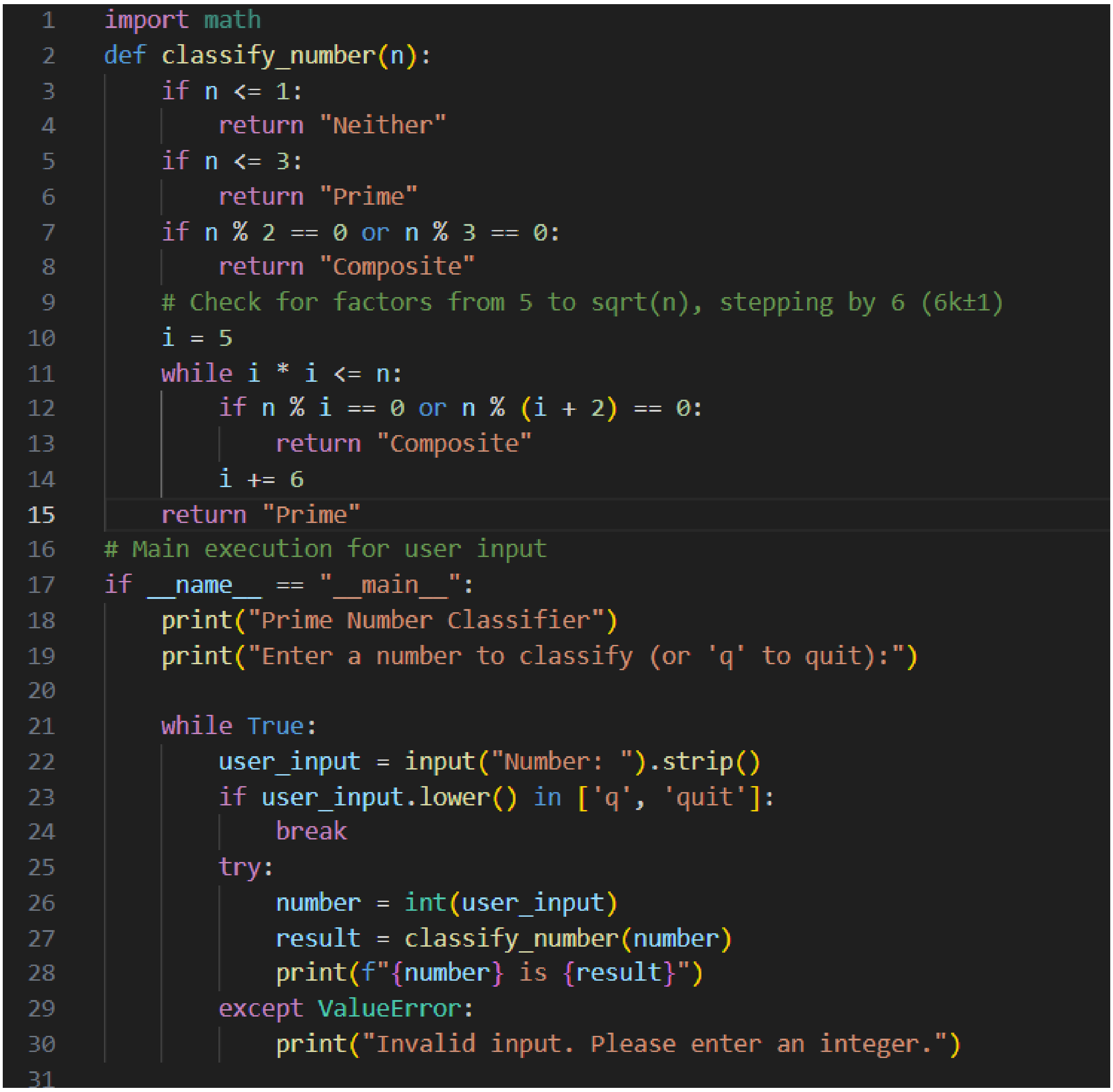
Design a context-managed prompt with clear instructions and constraints to generate an optimized Python program that classifies a number as prime, composite, or neither.

Task:

- Ensure proper input validation.

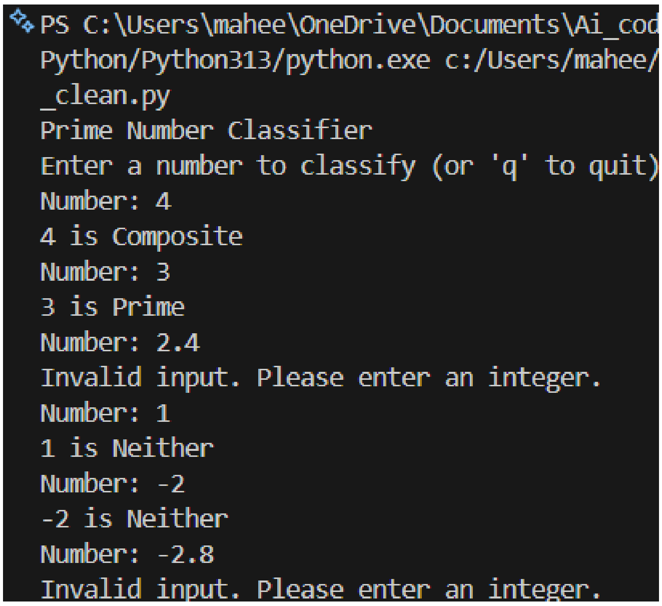
CODE:

- **Ensure proper input validation.**
- **Optimize the logic for efficiency.**



```python
import math
def classify_number(n):
    if n <= 1:
        return "Neither"
    if n <= 3:
        return "Prime"
    if n % 2 == 0 or n % 3 == 0:
        return "Composite"
    # Check for factors from 5 to sqrt(n), stepping by 6 (6k±1)
    i = 5
    while i * i <= n:
        if n % i == 0 or n % (i + 2) == 0:
            return "Composite"
        i += 6
    return "Prime"
# Main execution for user input
if __name__ == "__main__":
    print("Prime Number Classifier")
    print("Enter a number to classify (or 'q' to quit):")

    while True:
        user_input = input("Number: ").strip()
        if user_input.lower() in ['q', 'quit']:
            break
        try:
            number = int(user_input)
            result = classify_number(number)
            print(f"{number} is {result}")
        except ValueError:
            print("Invalid input. Please enter an integer.")

```

Output:



```
PS C:\Users\mahee\OneDrive\Documents\Ai_cod
Python/Python313/python.exe c:/Users/mahee/
_clean.py
Prime Number Classifier
Enter a number to classify (or 'q' to quit)
Number: 4
4 is Composite
Number: 3
3 is Prime
Number: 2.4
Invalid input. Please enter an integer.
Number: 1
1 is Neither
Number: -2
-2 is Neither
Number: -2.8
Invalid input. Please enter an integer.
```

Compare the output with earlier prompting strategies.

Description of Task 4 Program

The Task 4 program classifies a given number into one of the following categories:

- Prime
- Composite
- Neither

The program:

- Uses an optimized prime-checking algorithm (6k ± 1 method)
- Accepts continuous user input
- Allows the user to exit using ‘q’ or ‘quit’
- Handles invalid inputs using exception handling

Comparison with Earlier Prompting Strategies

1. Zero-Shot Prompting Output

- Generated simple logic without examples.
- Output usually limited to:
  - Prime or Not Prime
- Minimal or no handling of:
  - Number 1
  - Negative numbers
  - Repeated user input

Example Output:

17 is Prime

Limitations:

- No classification for special cases
- Less user-friendly
- Limited accuracy for edge cases

---

2. One-Shot Prompting Output

- Provided one example before code generation.
- Output clarity improved compared to zero-shot prompting.
- Still focused on binary classification.

Example Output:

```
1 is not Prime
```

Limitations:

- No separate category for “Neither”
- Less interactive
- Moderate error handling

---

3. Earlier Task Programs (Palindrome, Factorial, Armstrong)

- Output focused on single functionality
- One-time execution
- Simple True/False or numerical output

Examples:

```
True
```

```
120
```

```
Armstrong Number
```

Limitations:

- No classification categories
- Not reusable for continuous input
- Limited user interaction

---

4. Task 4 Output (Context-Managed Prompting)

Example Outputs:

```
1 is Neither

2 is Prime

9 is Composite

17 is Prime
```

Improvements Observed

- Clear and meaningful classification
- Handles edge cases correctly
- Efficient performance
- Better user interaction
- Continuous execution without restarting the program

Comparison Table

Feature	Earlier Prompting	Task 4
Output Type	Simple	Descriptive
Classification	Limited	Prime / Composite / Neither
Input Handling	Single input	Continuous input
Edge Cases	Poor	Well handled
Performance	Basic	Optimized
User Experience	Low	High

Conclusion

The output generated in Task 4 is significantly better than earlier prompting strategies.
It provides accurate classification, handles edge cases, and improves performance and usability.
This shows that context-managed prompting produces higher-quality and more reliable outputs compared to zero-shot and one-shot prompting.

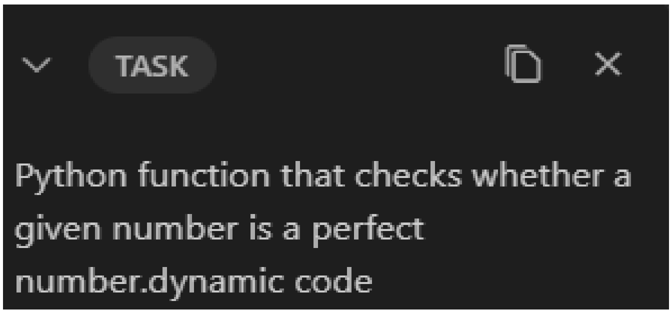
Question 5: Zero-Shot Prompting (Perfect Number Check)

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:

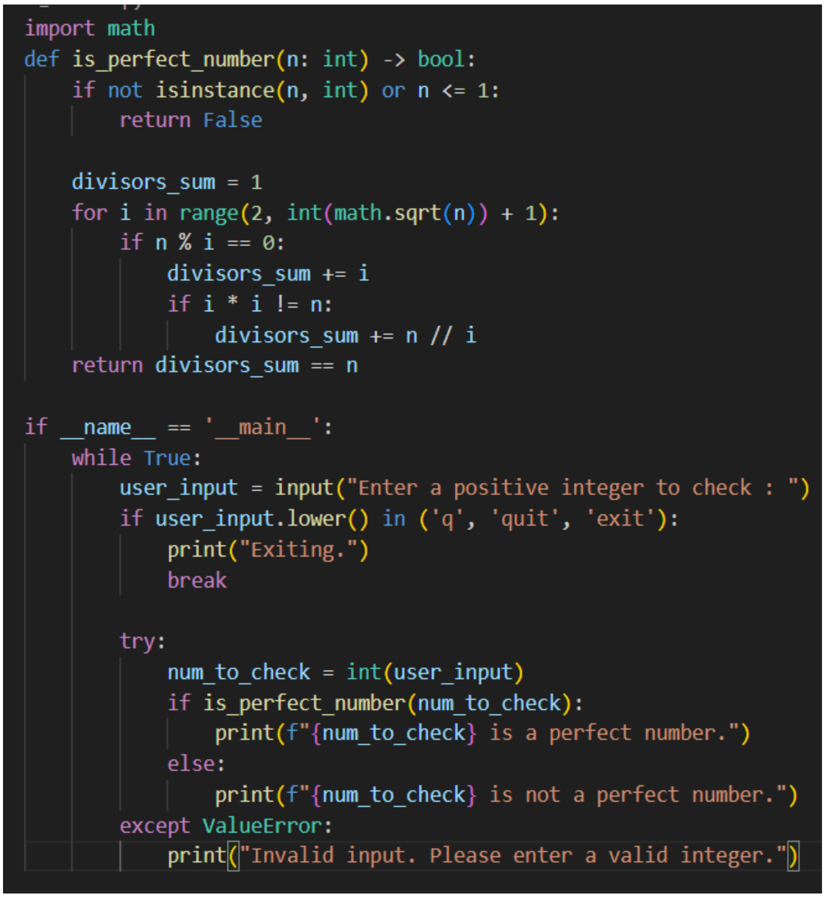
- Record the AI-generated code.
- Test the program with multiple inputs.
- Identify any missing conditions or inefficiencies in the logic.

PROMPT:



Python function that checks whether a given number is a perfect number.dynamic code

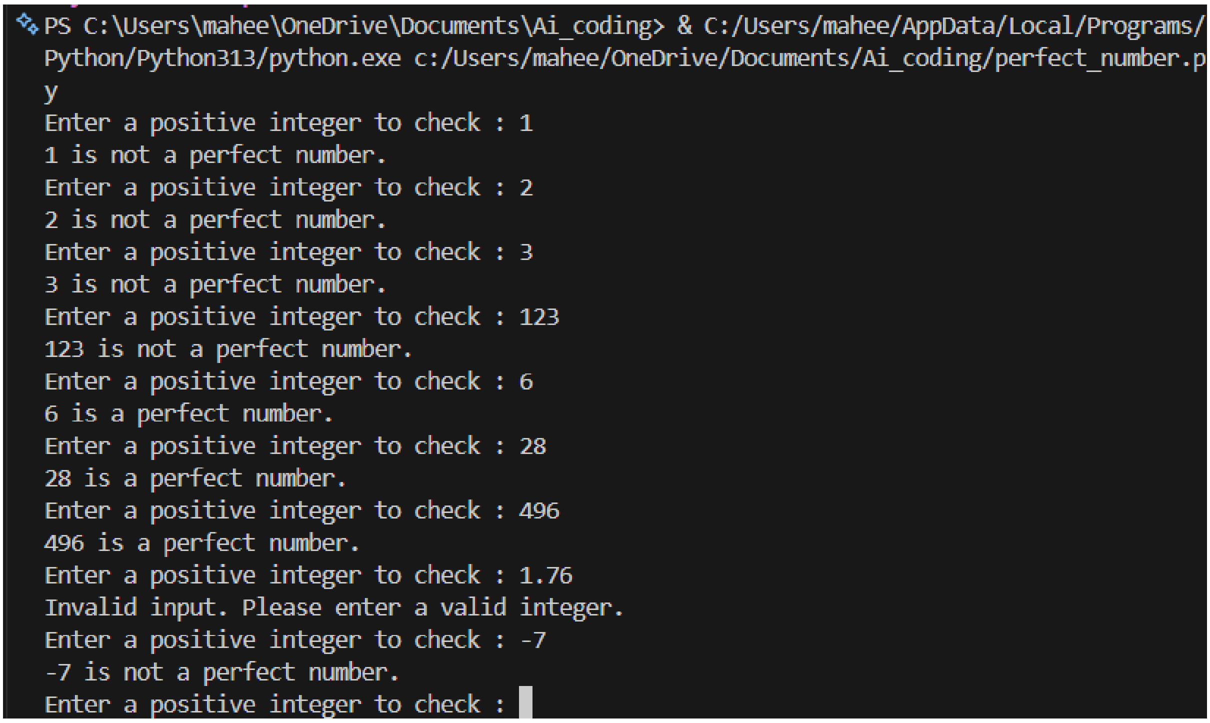
CODE:



```
import math
def is_perfect_number(n: int) -> bool:
    if not isinstance(n, int) or n <= 1:
        return False

    divisors_sum = 1
    for i in range(2, int(math.sqrt(n)) + 1):
        if n % i == 0:
            divisors_sum += i
            if i * i != n:
                divisors_sum += n // i
    return divisors_sum == n

if __name__ == '__main__':
    while True:
        user_input = input("Enter a positive integer to check : ")
        if user_input.lower() in ('q', 'quit', 'exit'):
            print("Exiting.")
            break

        try:
            num_to_check = int(user_input)
            if is_perfect_number(num_to_check):
                print(f"{num_to_check} is a perfect number.")
            else:
                print(f"{num_to_check} is not a perfect number.")
        except ValueError:
            print("Invalid input. Please enter a valid integer.")
```

Output:

Test the program with multiple inputs.



```
PS C:\Users\mahee\OneDrive\Documents\Ai_coding> & C:/Users/mahee/AppData/Local/Programs/
Python/Python313/python.exe c:/Users/mahee/OneDrive/Documents/Ai_coding/perfect_number.p
y
Enter a positive integer to check : 1
1 is not a perfect number.
Enter a positive integer to check : 2
2 is not a perfect number.
Enter a positive integer to check : 3
3 is not a perfect number.
Enter a positive integer to check : 123
123 is not a perfect number.
Enter a positive integer to check : 6
6 is a perfect number.
Enter a positive integer to check : 28
28 is a perfect number.
Enter a positive integer to check : 496
496 is a perfect number.
Enter a positive integer to check : 1.76
Invalid input. Please enter a valid integer.
Enter a positive integer to check : -7
-7 is not a perfect number.
Enter a positive integer to check :
```

Identification of Missing Conditions & Inefficiencies

1. Missing Return Statement (Logical Error) ❌

Problem:

return divisors_sum == n

This line is missing the return keyword in your code image.

Effect:

- The function returns None instead of True or False
- Condition if is_perfect_number(num_to_check): may behave incorrectly

Fix:

return divisors_sum == n

---

2. Unnecessary isinstance Check

Code:

```
if not isinstance(n, int) or n <= 1:
```

Issue:

- n is already converted using int(user_input)
- ValueError is already handled in try-except

Why inefficient:

- Redundant check
- Slight performance overhead

Better approach:

```
if n <= 1:
    return False
```

---

3. Unused Import (math) (Minor)

Code:

```
import math
```

You only use:

```
math.sqrt(n)
```

Improvement (optional):
Use integer square root (faster and cleaner):

```
int(n ** 0.5)
```

This avoids importing math.

---

4. Initializing divisors_sum = 1 (Edge Case Awareness)

Logic:

```
divisors_sum = 1
```

Issue:

- Correct only because you already exclude n <= 1

- If condition is removed later, 1 could be wrongly classified

Recommendation:
Keep as is only if n <= 1 check remains.

---

5. Loop Efficiency (Good, but can be optimized further)

Current loop:

```
for i in range(2, int(math.sqrt(n)) + 1):
```

✓ Efficient compared to checking till n-1
✓ Uses divisor pair optimization

Further optimization (optional):
Stop early if sum exceeds n:

```
if divisors_sum > n:
    return False
```

---

6. Missing Type Hint Consistency (Minor)

You used:

```
def is_perfect_number(n: int) -> bool:
```

But still check isinstance(n, int) — this defeats the purpose of type hints.

Summary Table

Issue Type	Description	Severity
Logical Error	Missing return keyword	High
Redundant Check	isinstance(n, int)	Medium
Optimization	Early loop exit	Medium
Code Style	Unnecessary import	Low
Robustness	Depends on n <= 1 guard	Low

Final Conclusion :

The program efficiently checks for perfect numbers using square-root optimization.
However, it contains a critical logical error due to a missing return statement. Additionally, redundant type checking and minor inefficiencies can be removed to improve performance and code clarity.

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.

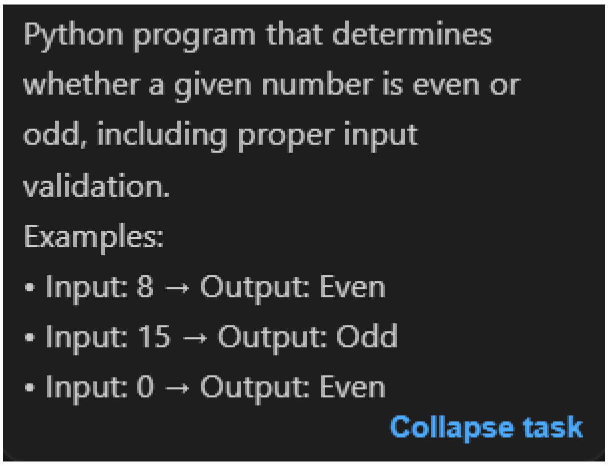
Examples:

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

Task:

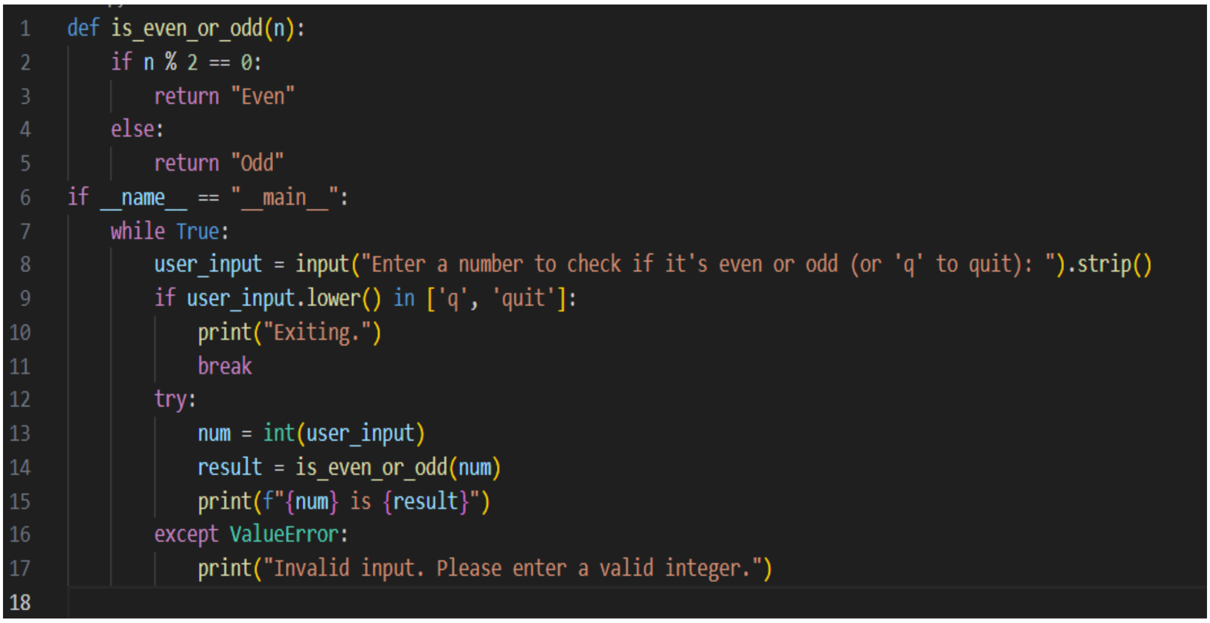
- Analyze how examples improve input handling and output clarity.
- Test the program with negative numbers and non-integer inputs.

PROMPT:



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

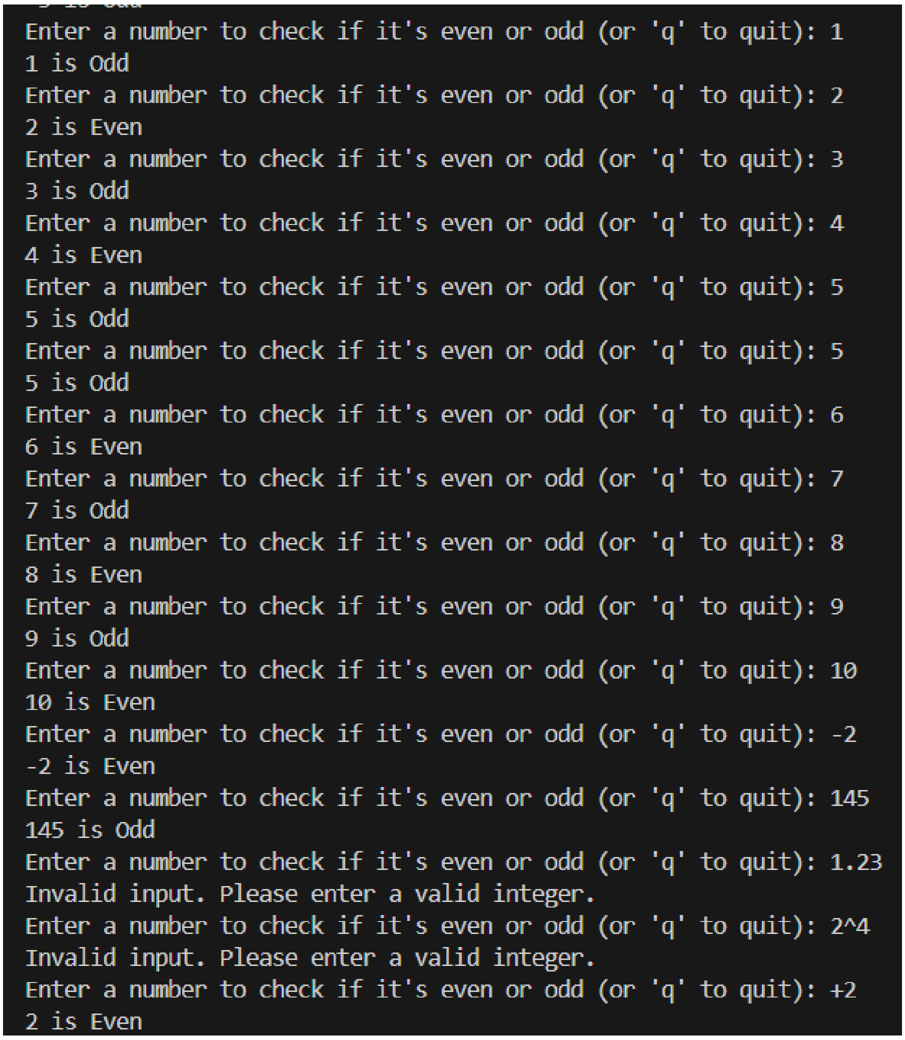
CODE:



```python
def is_even_or_odd(n):
    if n % 2 == 0:
        return "Even"
    else:
        return "Odd"
if __name__ == "__main__":
    while True:
        user_input = input("Enter a number to check if it's even or odd (or 'q' to quit): ").strip()
        if user_input.lower() in ['q', 'quit']:
            print("Exiting.")
            break
        try:
            num = int(user_input)
            result = is_even_or_odd(num)
            print(f"{num} is {result}")
        except ValueError:
            print("Invalid input. Please enter a valid integer.")

```

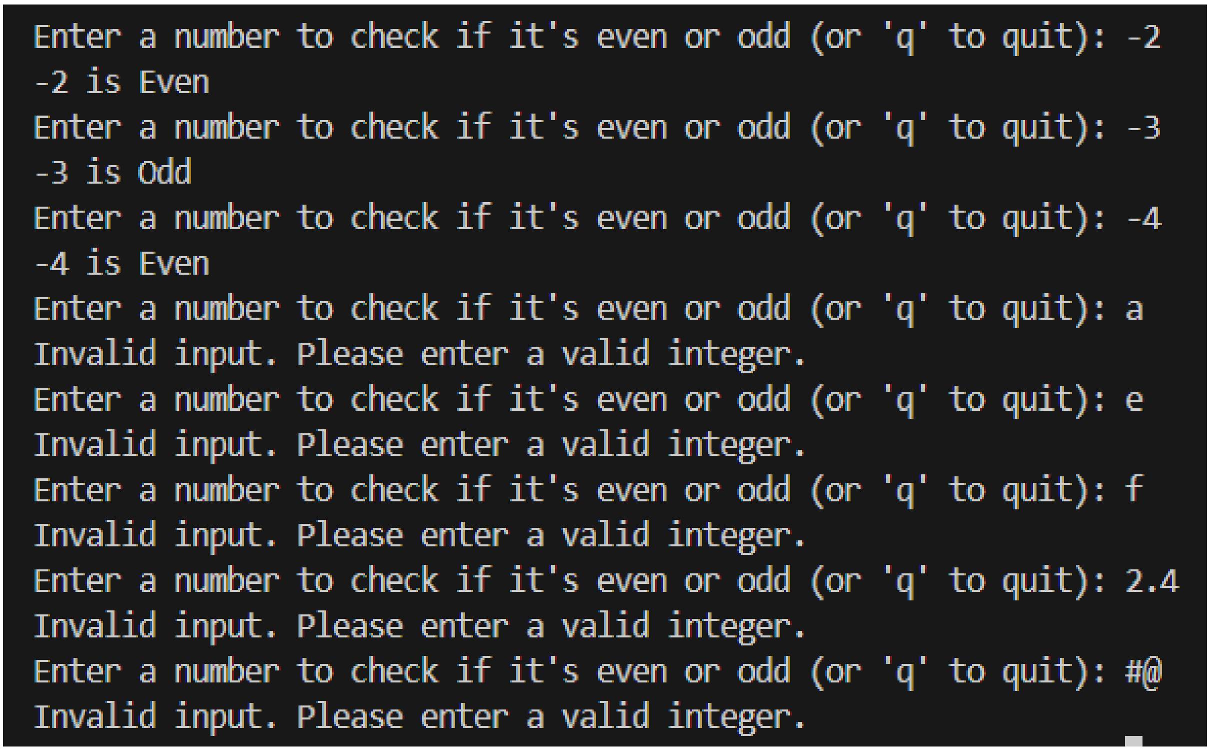
OUTPUT:

Analyze how examples improve input handling and output clarity



```
Enter a number to check if it's even or odd (or 'q' to quit): 1
1 is Odd
Enter a number to check if it's even or odd (or 'q' to quit): 2
2 is Even
Enter a number to check if it's even or odd (or 'q' to quit): 3
3 is Odd
Enter a number to check if it's even or odd (or 'q' to quit): 4
4 is Even
Enter a number to check if it's even or odd (or 'q' to quit): 5
5 is Odd
Enter a number to check if it's even or odd (or 'q' to quit): 5
5 is Odd
Enter a number to check if it's even or odd (or 'q' to quit): 6
6 is Even
Enter a number to check if it's even or odd (or 'q' to quit): 7
7 is Odd
Enter a number to check if it's even or odd (or 'q' to quit): 8
8 is Even
Enter a number to check if it's even or odd (or 'q' to quit): 9
9 is Odd
Enter a number to check if it's even or odd (or 'q' to quit): 10
10 is Even
Enter a number to check if it's even or odd (or 'q' to quit): -2
-2 is Even
Enter a number to check if it's even or odd (or 'q' to quit): 145
145 is Odd
Enter a number to check if it's even or odd (or 'q' to quit): 1.23
Invalid input. Please enter a valid integer.
Enter a number to check if it's even or odd (or 'q' to quit): 2^4
Invalid input. Please enter a valid integer.
Enter a number to check if it's even or odd (or 'q' to quit): +2
2 is Even
```

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



```
Enter a number to check if it's even or odd (or 'q' to quit): -2
-2 is Even
Enter a number to check if it's even or odd (or 'q' to quit): -3
-3 is Odd
Enter a number to check if it's even or odd (or 'q' to quit): -4
-4 is Even
Enter a number to check if it's even or odd (or 'q' to quit): a
Invalid input. Please enter a valid integer.
Enter a number to check if it's even or odd (or 'q' to quit): e
Invalid input. Please enter a valid integer.
Enter a number to check if it's even or odd (or 'q' to quit): f
Invalid input. Please enter a valid integer.
Enter a number to check if it's even or odd (or 'q' to quit): 2.4
Invalid input. Please enter a valid integer.
Enter a number to check if it's even or odd (or 'q' to quit): #@
Invalid input. Please enter a valid integer.
```