AI ASSISTED CODING

ASSIGNMENT-3.1

Name: P.RUTHWICK
HT No: 2303A52484
Batch: 31

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)

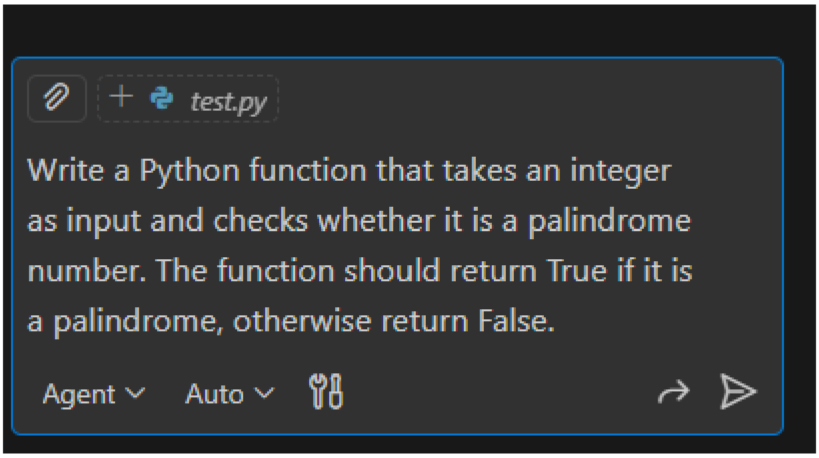
Write a zero-shot prompt (without providing any examples) to generate a Python function that checks whether a given number is a palindrome.

Task:

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

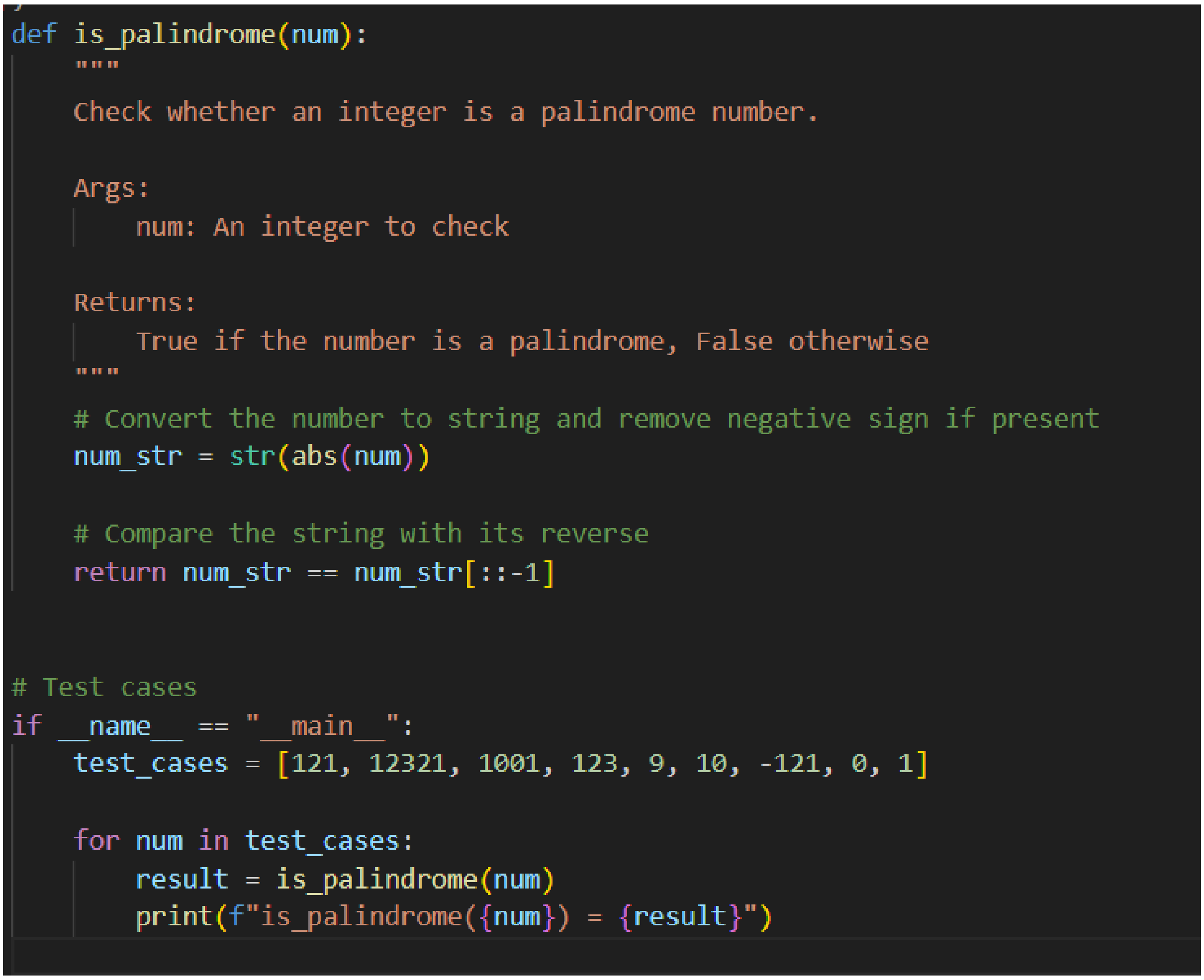
Prompt:

Write a Python function that takes an integer as input and checks whether it is a palindrome number. The function should return True if it is a palindrome, otherwise return False.



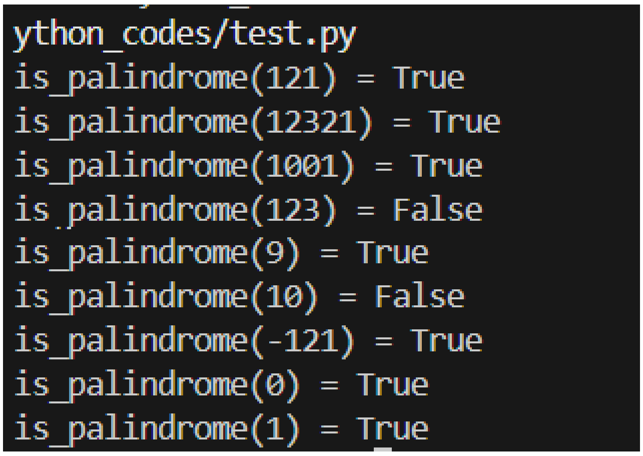
Write a Python function that takes an integer as input and checks whether it is a palindrome number. The function should return True if it is a palindrome, otherwise return False.

Code:



```python
def is_palindrome(num):
    """
    Check whether an integer is a palindrome number.

    Args:
        num: An integer to check

    Returns:
        True if the number is a palindrome, False otherwise
    """
    # Convert the number to string and remove negative sign if present
    num_str = str(abs(num))

    # Compare the string with its reverse
    return num_str == num_str[::-1]


# Test cases
if __name__ == "__main__":
    test_cases = [121, 12321, 1001, 123, 9, 10, -121, 0, 1]

    for num in test_cases:
        result = is_palindrome(num)
        print(f"is_palindrome({num}) = {result}")
```

Output(O/P)



```
ython_codes/test.py
is_palindrome(121) = True
is_palindrome(12321) = True
is_palindrome(1001) = True
is_palindrome(123) = False
is_palindrome(9) = True
is_palindrome(10) = False
is_palindrome(-121) = True
is_palindrome(0) = True
is_palindrome(1) = True
```

Explanation:

- A zero-shot prompt was written without giving examples.
- AI generated a palindrome-checking function in Python.
- The function reverses the number and compares it with the original.
- Answer Returns Either True or False

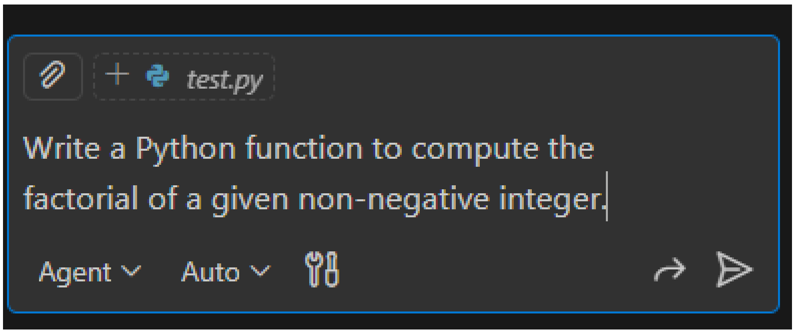
Question 2:

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:

Write a Python function to compute the factorial of a given non-negative integer.

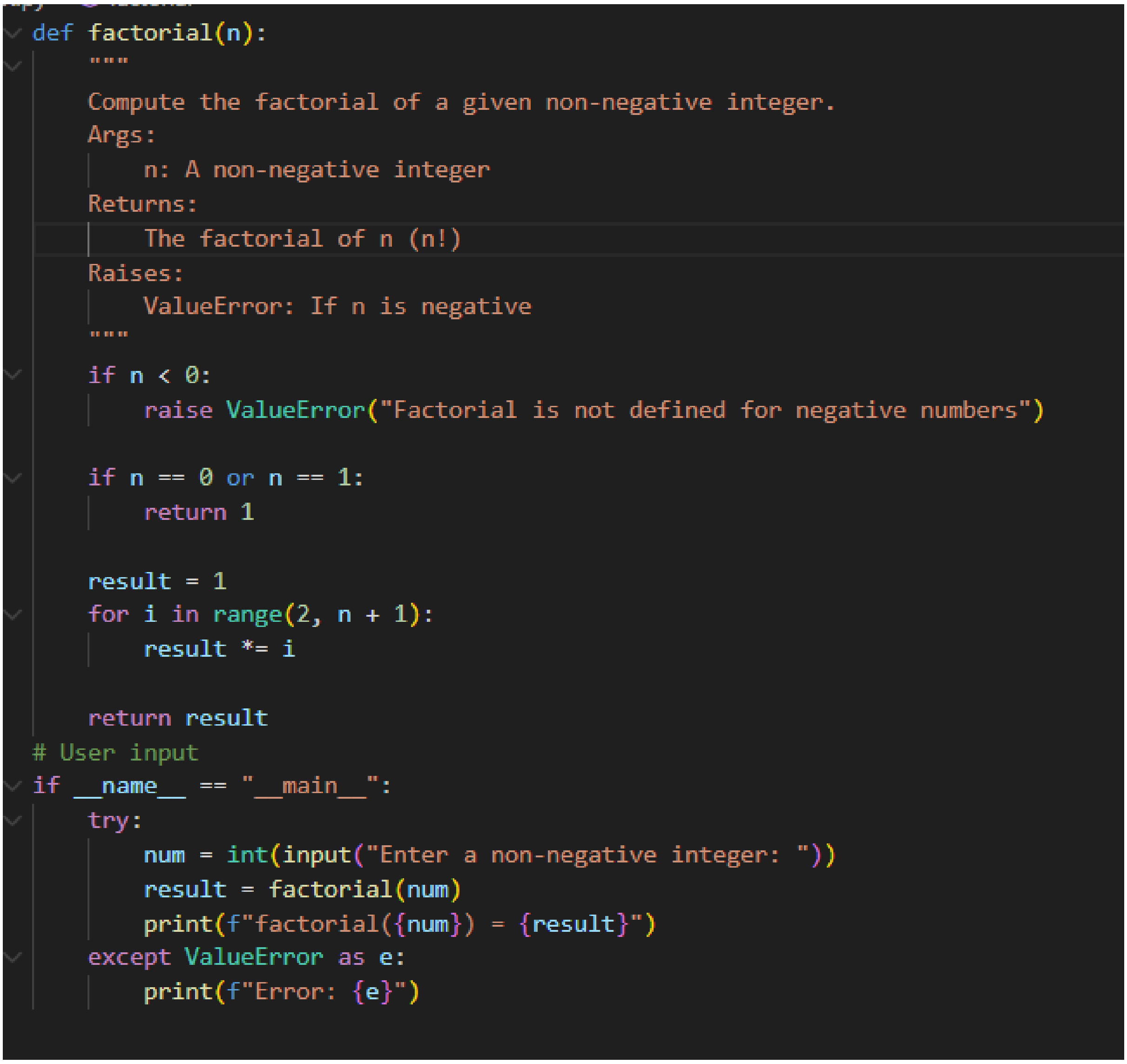


Write a Python function to compute the factorial of a given non-negative integer.

Example:
Input: 5
Output: 120

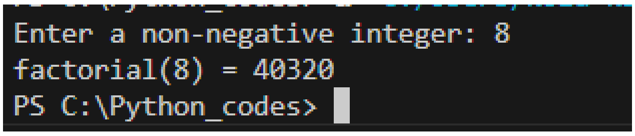
The function should return the factorial value.

Code:



```python
def factorial(n):
    """
    Compute the factorial of a given non-negative integer.
    Args:
        n: A non-negative integer
    Returns:
        The factorial of n (n!)
    Raises:
        ValueError: If n is negative
    """
    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
# User input
if __name__ == "__main__":
    try:
        num = int(input("Enter a non-negative integer: "))
        result = factorial(num)
        print(f"factorial({num}) = {result}")
    except ValueError as e:
        print(f"Error: {e}")
```

Output:



```
Enter a non-negative integer: 8
factorial(8) = 40320
PS C:\Python_codes>
```

Explanation:

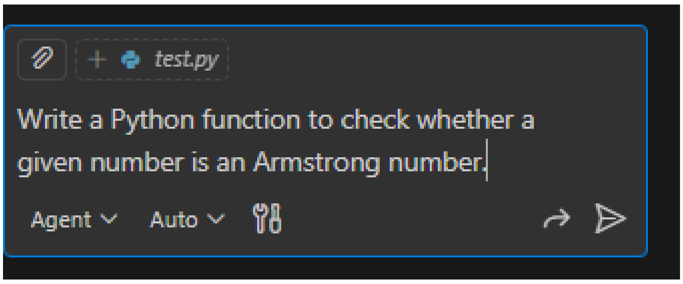
Aspect	Zero-Shot	One-Shot
Example Given	No	Yes
Code Clarity	Basic	Clear
Edge Cases	Limited	Handled
Validation	Not explicit	Explicit
Correctness	Task-level	More robust

Improvements Observed in One-Shot Prompting
- Better clarity due to example-guided generation
- Improved correctness with input validation
- More readable and maintainable code
- Clear handling of edge cases compared to zero-shot

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.

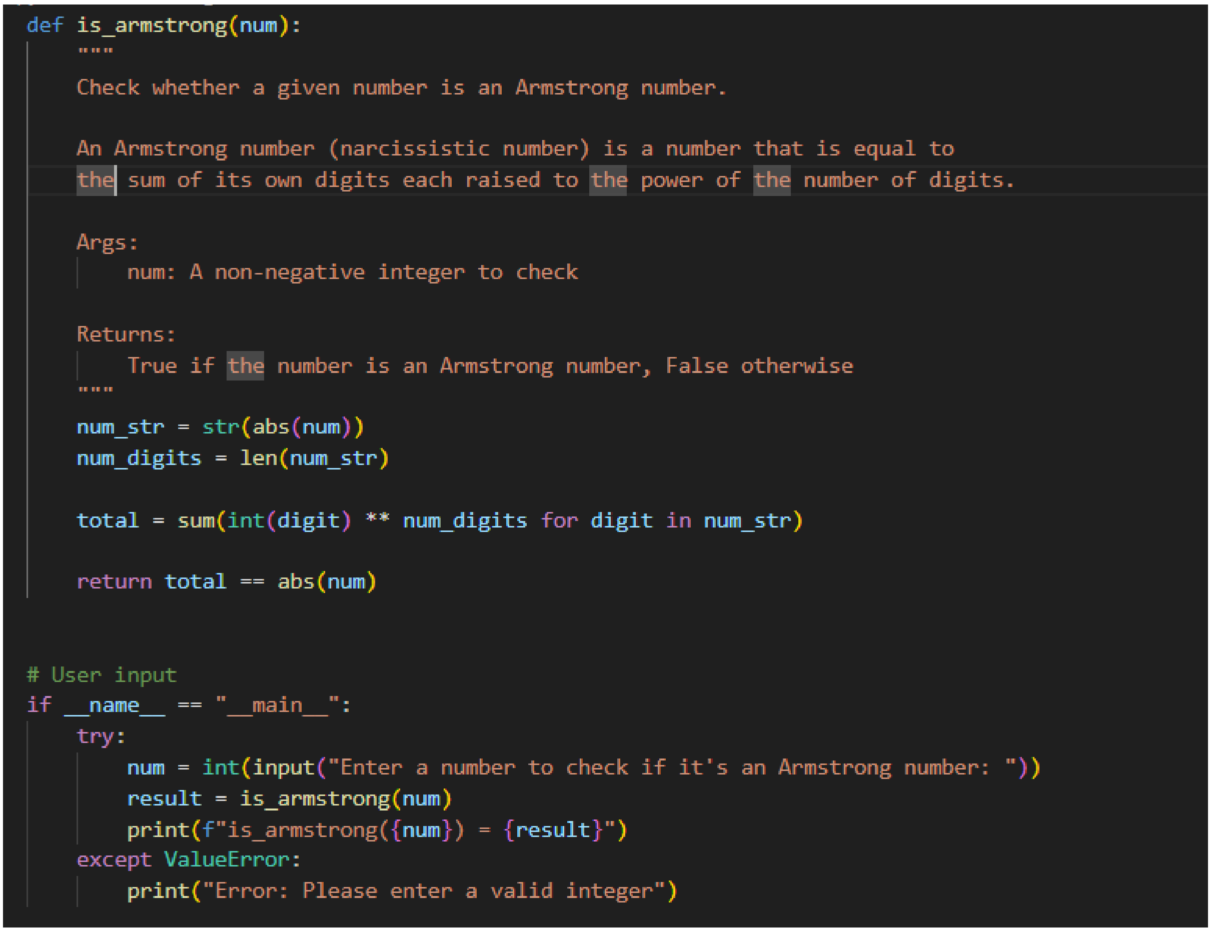


Write a Python function to check whether a given number is an Armstrong number.

Prompt:

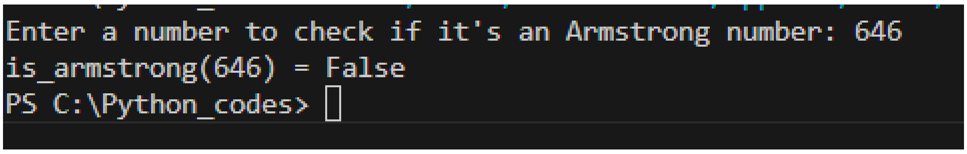
Write a Python function to check whether a given number is an Armstrong number.
Examples:
Input: 153 → Output: Armstrong Number
Input: 370 → Output: Armstrong Number
Input: 123 → Output: Not an Armstrong Number
The function should return
"Armstrong Number" or "Not an Armstrong Number" accordingly

Code:



```python
def is_armstrong(num):
    """
    Check whether a given number is an Armstrong number.

    An Armstrong number (narcissistic number) is a number that is equal to
    the sum of its own digits each raised to the power of the number of digits.

    Args:
        num: A non-negative integer to check

    Returns:
        True if the number is an Armstrong number, False otherwise
    """
    num_str = str(abs(num))
    num_digits = len(num_str)

    total = sum(int(digit) ** num_digits for digit in num_str)

    return total == abs(num)


# User input
if __name__ == "__main__":
    try:
        num = int(input("Enter a number to check if it's an Armstrong number: "))
        result = is_armstrong(num)
        print(f"is_armstrong({num}) = {result}")
    except ValueError:
        print("Error: Please enter a valid integer")
```

Output:



```
Enter a number to check if it's an Armstrong number: 646
is_armstrong(646) = False
PS C:\Python_codes>
```

Explanation:

Influence of Multiple Examples
- Examples clarify expected output format
- Guides correct power calculation based on digit count
- Improves accuracy compared to zero or one-shot
- Reduces ambiguity in logic and return values

Testing (Boundary and Invalid Inputs)
- Input: 0 → Armstrong Number
- Input: 1 → Armstrong Number
- Input: 9474 → Armstrong Number
- Input: -153 → Not an Armstrong Number
- Input: "abc" → Invalid Input

Question 4:

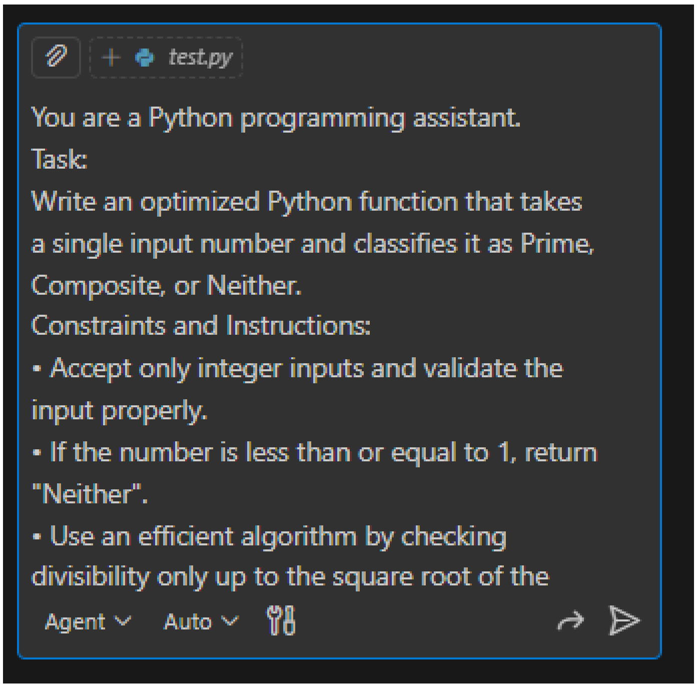
Context-Managed Prompting (Optimized Number Classification)

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.

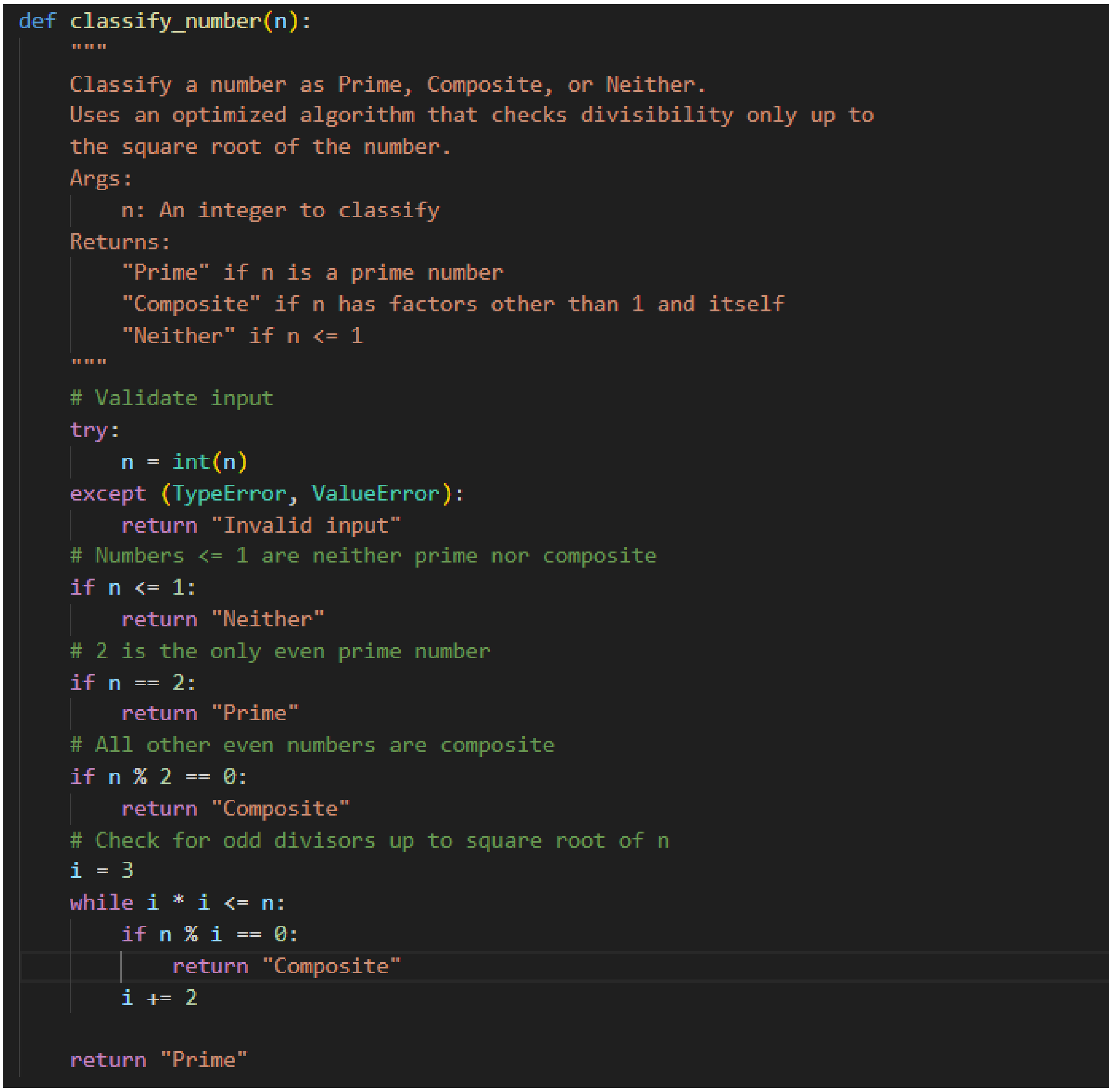
Prompt:

You are a Python programming assistant.
Task:
Write an optimized Python function that takes a single input number and classifies it as Prime, Composite, or Neither.
Constraints and Instructions:
• Accept only integer inputs and validate the input properly.
• If the number is less than or equal to 1, return "Neither".
• Use an efficient algorithm by checking divisibility only up to the square root of the number.
• The function should return one of the following strings only:
"Prime", "Composite", or "Neither".
• Do not use external libraries.
Output should be correct and optimized



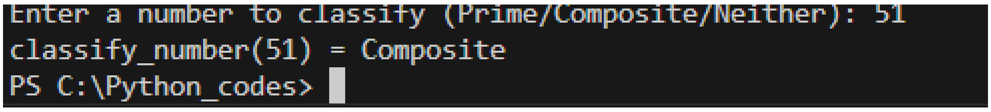
You are a Python programming assistant.
Task:
Write an optimized Python function that takes a single input number and classifies it as Prime, Composite, or Neither.
Constraints and Instructions:
• Accept only integer inputs and validate the input properly.
• If the number is less than or equal to 1, return "Neither".
• Use an efficient algorithm by checking divisibility only up to the square root of the

Code:



```python
def classify_number(n):
    """
    Classify a number as Prime, Composite, or Neither.
    Uses an optimized algorithm that checks divisibility only up to
    the square root of the number.
    Args:
        n: An integer to classify
    Returns:
        "Prime" if n is a prime number
        "Composite" if n has factors other than 1 and itself
        "Neither" if n <= 1
    """
    # Validate input
    try:
        n = int(n)
    except (TypeError, ValueError):
        return "Invalid input"
    # Numbers <= 1 are neither prime nor composite
    if n <= 1:
        return "Neither"
    # 2 is the only even prime number
    if n == 2:
        return "Prime"
    # All other even numbers are composite
    if n % 2 == 0:
        return "Composite"
    # Check for odd divisors up to square root of n
    i = 3
    while i * i <= n:
        if n % i == 0:
            return "Composite"
        i += 2

    return "Prime"
```

Output:



```
Enter a number to classify (Prime/Composite/Neither): 51
classify_number(51) = Composite
PS C:\Python_codes> 
```

Explanation:

Comparison with Earlier Prompting Strategies

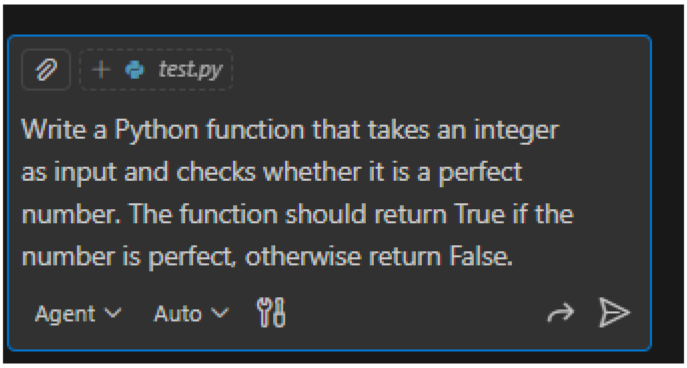
- More structured than zero-shot
- Clearer constraints than one-shot
- Less ambiguity than few-shot
- Produces efficient and validated code

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.

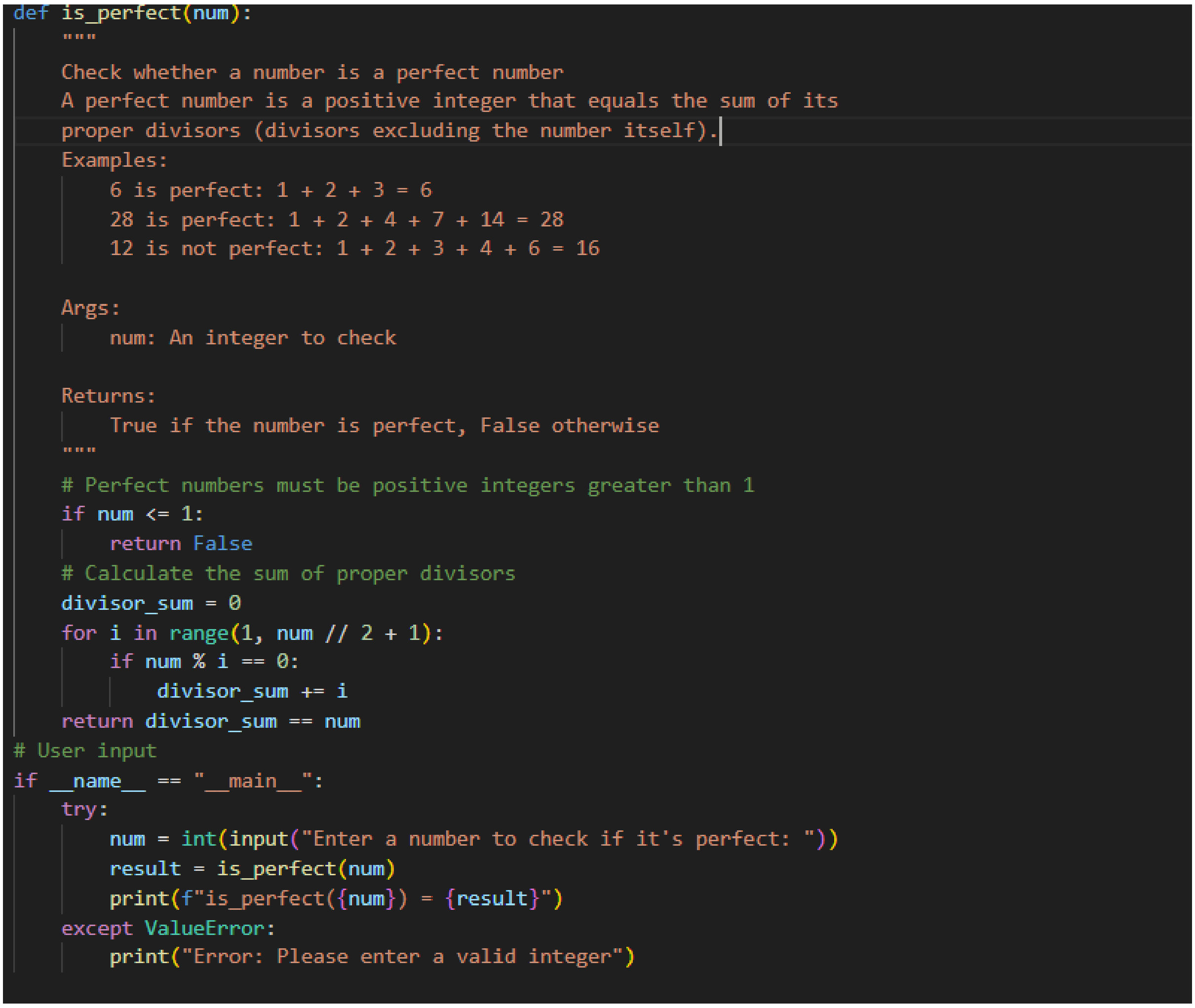
Prompt:

Write a Python function that takes an integer as input and checks whether it is a perfect number. The function should return True if the number is perfect, otherwise return False.



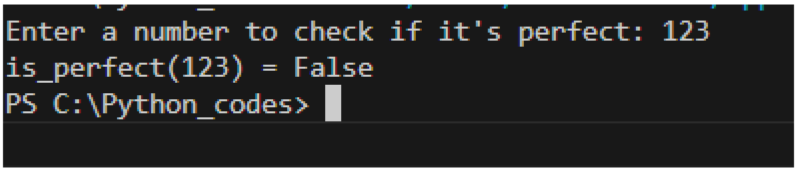
Write a Python function that takes an integer as input and checks whether it is a perfect number. The function should return True if the number is perfect, otherwise return False.

Code:



```python
def is_perfect(num):
    """
    Check whether a number is a perfect number
    A perfect number is a positive integer that equals the sum of its
    proper divisors (divisors excluding the number itself).
    Examples:
        6 is perfect: 1 + 2 + 3 = 6
        28 is perfect: 1 + 2 + 4 + 7 + 14 = 28
        12 is not perfect: 1 + 2 + 3 + 4 + 6 = 16

    Args:
        num: An integer to check

    Returns:
        True if the number is perfect, False otherwise
    """
    # Perfect numbers must be positive integers greater than 1
    if num <= 1:
        return False
    # Calculate the sum of proper divisors
    divisor_sum = 0
    for i in range(1, num // 2 + 1):
        if num % i == 0:
            divisor_sum += i
    return divisor_sum == num
# User input
if __name__ == "__main__":
    try:
        num = int(input("Enter a number to check if it's perfect: "))
        result = is_perfect(num)
        print(f"is_perfect({num}) = {result}")
    except ValueError:
        print("Error: Please enter a valid integer")
```

Output:



```
Enter a number to check if it's perfect: 123
is_perfect(123) = False
PS C:\Python_codes>
```

Explanation:

Testing the Program:

- Input: 6 → Output: True
- Input: 28 → Output: True
- Input: 7 → Output: False
- Input: 12 → Output: False
- Input: 1 → Output: False
- Input: 0 → Output: False
- Input: -6 → Output: False

Missing Conditions and Inefficiencies:

- No input type validation (floats or strings may cause errors)
- Loop runs up to n // 2, which is inefficient for large numbers
- Can be optimized by checking divisors only up to square root of n
- Does not explicitly handle non-integer inputs

Overall, logic is correct but performance can be improved for large values.

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:

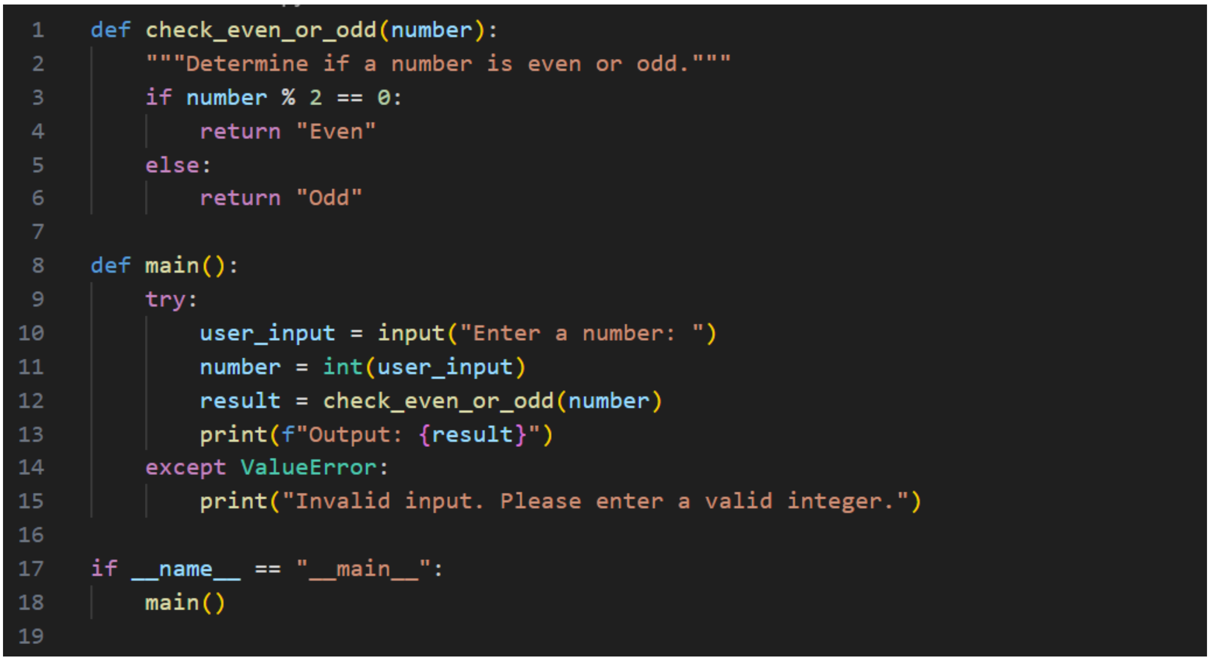
Write a Python program that determines whether a given input number is Even or Odd.

Examples:

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

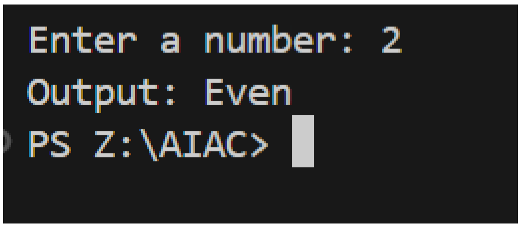
The program should validate the input and handle invalid (non-integer) values gracefully.

Code:



```python
def check_even_or_odd(number):
    """Determine if a number is even or odd."""
    if number % 2 == 0:
        return "Even"
    else:
        return "Odd"

def main():
    try:
        user_input = input("Enter a number: ")
        number = int(user_input)
        result = check_even_or_odd(number)
        print(f"Output: {result}")
    except ValueError:
        print("Invalid input. Please enter a valid integer.")

if __name__ == "__main__":
    main()

```

Output:



```
Enter a number: 2
Output: Even
PS Z:\AIAC> 
```

Explanation:

Analysis: Effect of Examples on Input Handling and Output Clarity
- Examples make it clear that the output must be only “Even” or “Odd”
- Inclusion of 0 → Even avoids ambiguity about zero
- Encourages explicit input validation using try–except
- Improves clarity by separating logic and input handling
- Output format becomes consistent and predictable

Testing the Program:
Negative Numbers
- Input: -10 → Output: Even
- Input: -3 → Output: Odd

Non-Integer Inputs
- Input: 3.5 → Output: Invalid input. Please enter a valid integer.
- Input: "abc" → Output: Invalid input. Please enter a valid integer.

Conclusion:
Few-shot examples guide the program to handle inputs safely and produce clear, reliable outputs.