# AI Assisted Coding

## ASSIGNMENT 3.1

Name: Madadi Uday
HT No:  2303A52038
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.

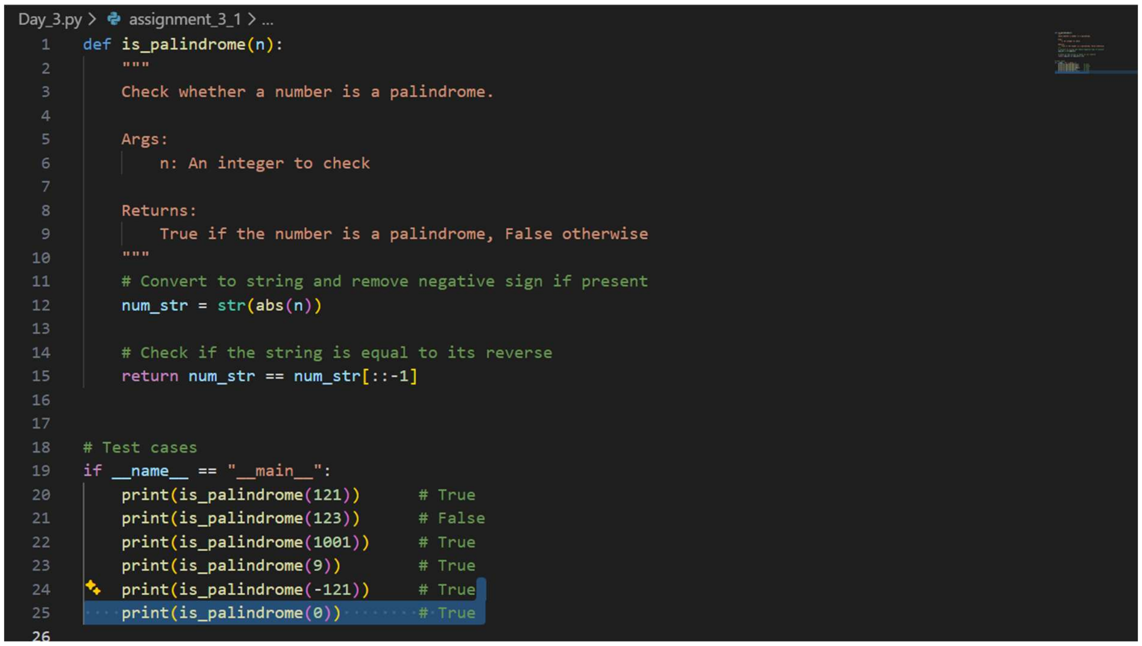
• Test the code with multiple inputs.

• 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.

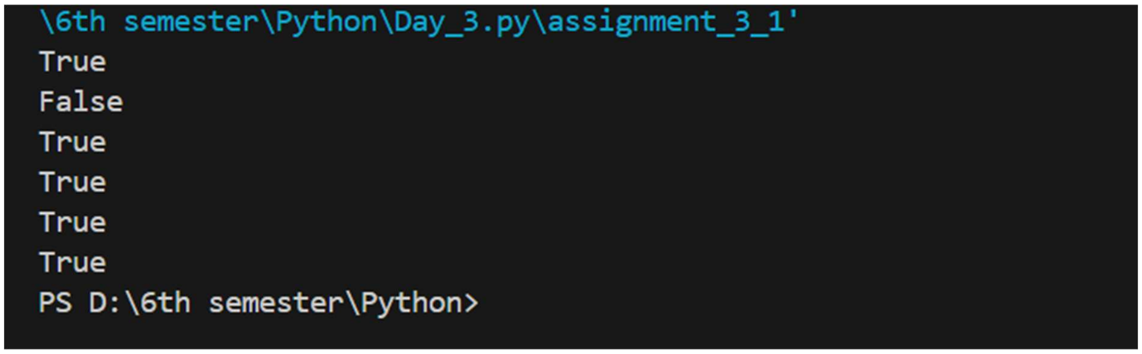
## Code:



Day_3.py > assignment_3_1 > ...

```python
def is_palindrome(n):
    """
    Check whether a number is a palindrome.

    Args:
        n: An integer to check

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

    # Check if the string is equal to its reverse
    return num_str == num_str[::-1]


# Test cases
if __name__ == "__main__":
    print(is_palindrome(121))      # True
    print(is_palindrome(123))      # False
    print(is_palindrome(1001))     # True
    print(is_palindrome(9))        # True
    print(is_palindrome(-121))     # True
    print(is_palindrome(0))        # True
```

## Output(O/P)



```
\6th semester\Python\Day_3.py\assignment_3_1'
True
False
True
True
True
True
PS D:\6th semester\Python>
```

## 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.

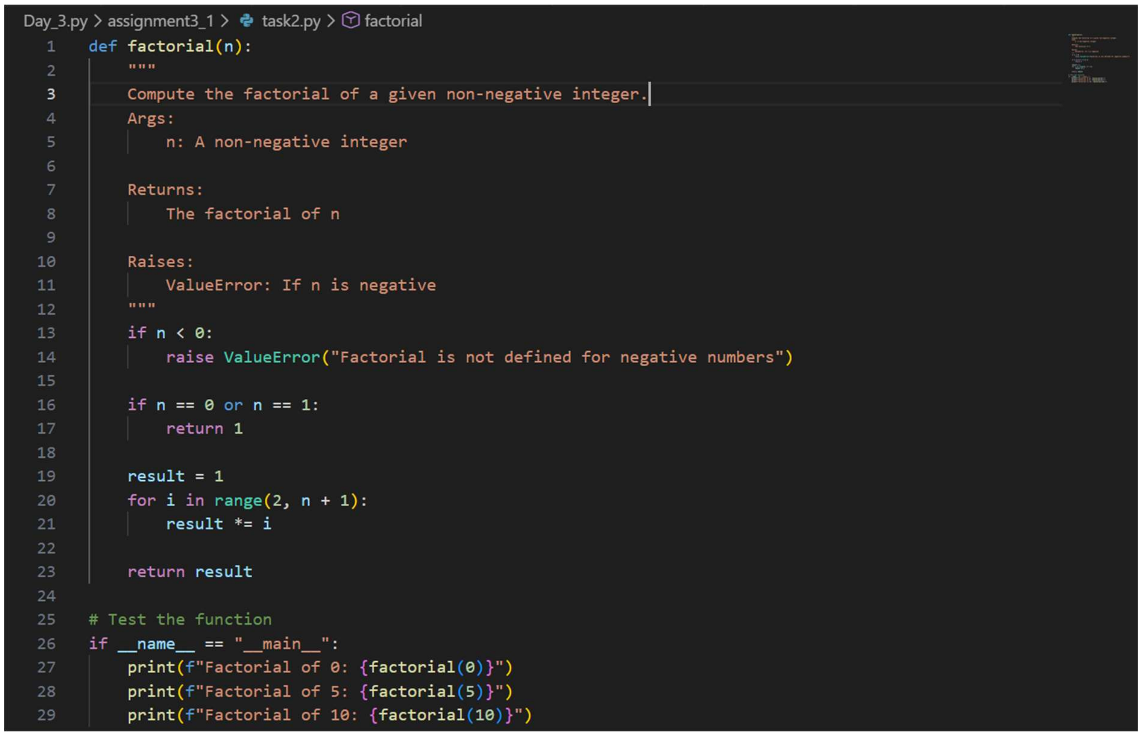
## Prompt:

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

Example:
Input: 5
Output: 120

The function should return the factorial value.

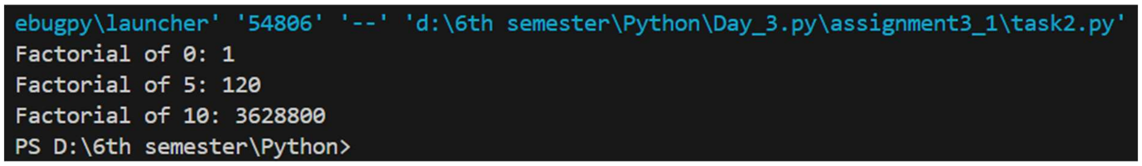
Code:



Day_3.py > assignment3_1 > task2.py > factorial

```python
def factorial(n):
    """
    Compute the factorial of a given non-negative integer.
    Args:
        n: A non-negative integer

    Returns:
        The factorial of 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

# Test the function
if __name__ == "__main__":
    print(f"Factorial of 0: {factorial(0)}")
    print(f"Factorial of 5: {factorial(5)}")
    print(f"Factorial of 10: {factorial(10)}")
```

Output:



```
ebugpy\launcher' '54806' '--' 'd:\6th semester\Python\Day_3.py\assignment3_1\task2.py'
Factorial of 0: 1
Factorial of 5: 120
Factorial of 10: 3628800
PS D:\6th semester\Python>
```

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:

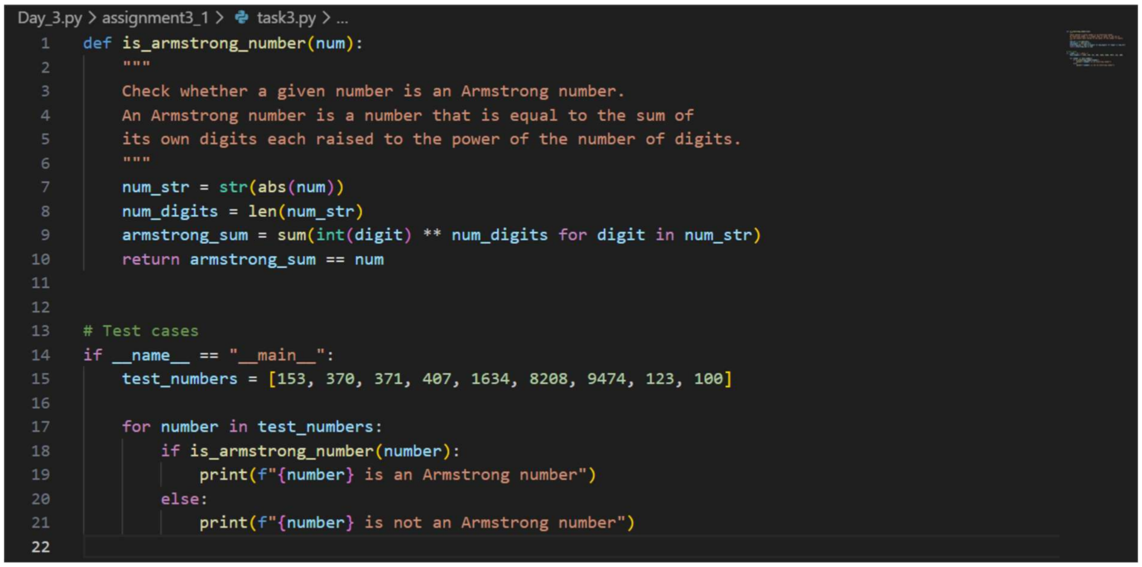
### Few-Shot Prompting (Armstrong Number Check)

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.

### 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

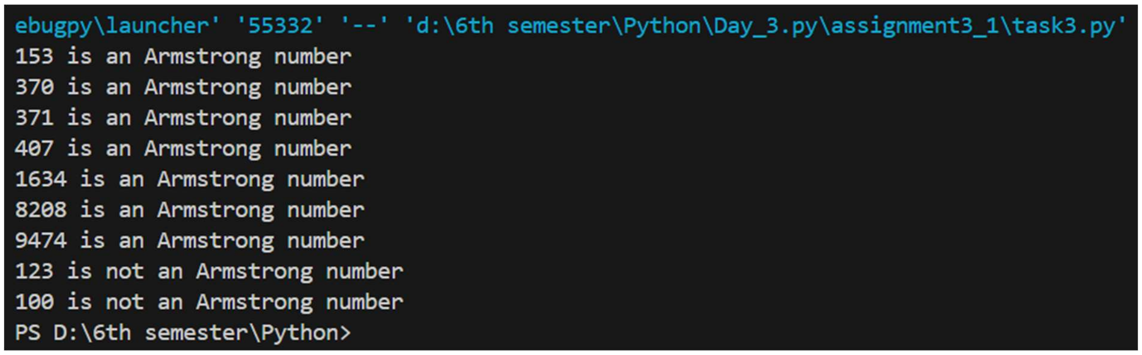
## Code:



Day_3.py > assignment3_1 > task3.py > ...

```python
def is_armstrong_number(num):
    """
    Check whether a given number is an Armstrong number.
    An Armstrong number is a number that is equal to the sum of
    its own digits each raised to the power of the number of digits.
    """
    num_str = str(abs(num))
    num_digits = len(num_str)
    armstrong_sum = sum(int(digit) ** num_digits for digit in num_str)
    return armstrong_sum == num


# Test cases
if __name__ == "__main__":
    test_numbers = [153, 370, 371, 407, 1634, 8208, 9474, 123, 100]

    for number in test_numbers:
        if is_armstrong_number(number):
            print(f"{number} is an Armstrong number")
        else:
            print(f"{number} is not an Armstrong number")

```

## Output:



```
ebugpy\launcher' '55332' '--' 'd:\6th semester\Python\Day_3.py\assignment3_1\task3.py'
153 is an Armstrong number
370 is an Armstrong number
371 is an Armstrong number
407 is an Armstrong number
1634 is an Armstrong number
8208 is an Armstrong number
9474 is an Armstrong number
123 is not an Armstrong number
100 is not an Armstrong number
PS D:\6th semester\Python>
```

## 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:

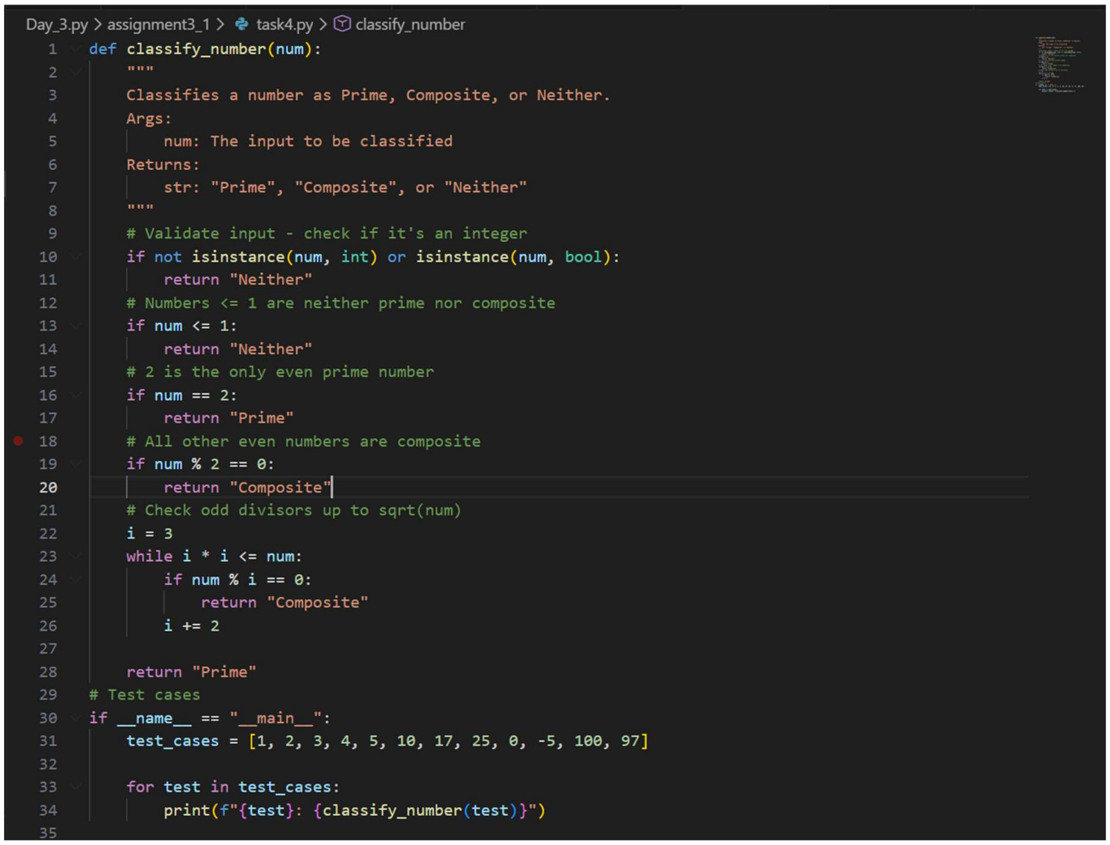
### Context-Managed Prompting (Optimized Number Classification)

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

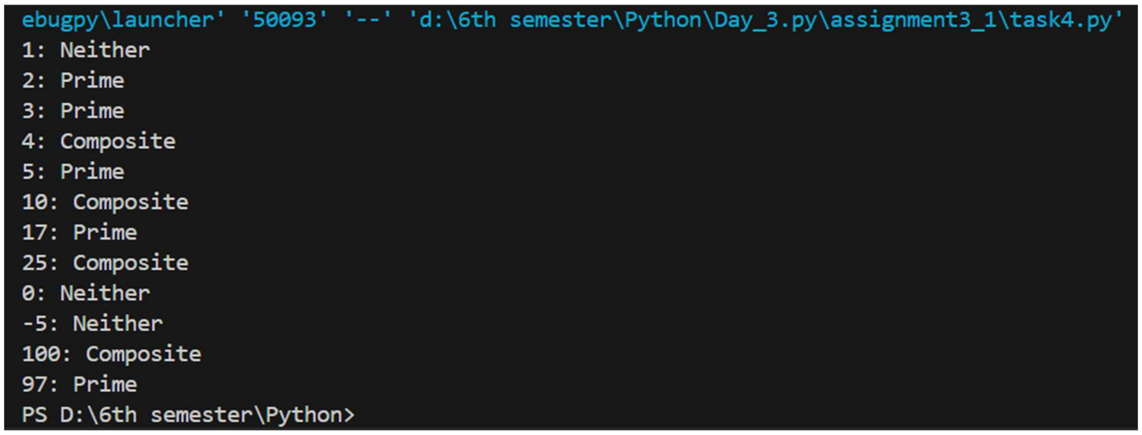
## Code:



Day_3.py > assignment3_1 > task4.py > classify_number

```python
def classify_number(num):
    """
    Classifies a number as Prime, Composite, or Neither.
    Args:
        num: The input to be classified
    Returns:
        str: "Prime", "Composite", or "Neither"
    """
    # Validate input - check if it's an integer
    if not isinstance(num, int) or isinstance(num, bool):
        return "Neither"
    # Numbers <= 1 are neither prime nor composite
    if num <= 1:
        return "Neither"
    # 2 is the only even prime number
    if num == 2:
        return "Prime"
    # All other even numbers are composite
    if num % 2 == 0:
        return "Composite"
    # Check odd divisors up to sqrt(num)
    i = 3
    while i * i <= num:
        if num % i == 0:
            return "Composite"
        i += 2

    return "Prime"
# Test cases
if __name__ == "__main__":
    test_cases = [1, 2, 3, 4, 5, 10, 17, 25, 0, -5, 100, 97]

    for test in test_cases:
        print(f"{test}: {classify_number(test)}")
```

## Output:



```
ebugpy\launcher' '50093' '--' 'd:\6th semester\Python\Day_3.py\assignment3_1\task4.py'
1: Neither
2: Prime
3: Prime
4: Composite
5: Prime
10: Composite
17: Prime
25: Composite
0: Neither
-5: Neither
100: Composite
97: Prime
PS D:\6th semester\Python>
```

## 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.

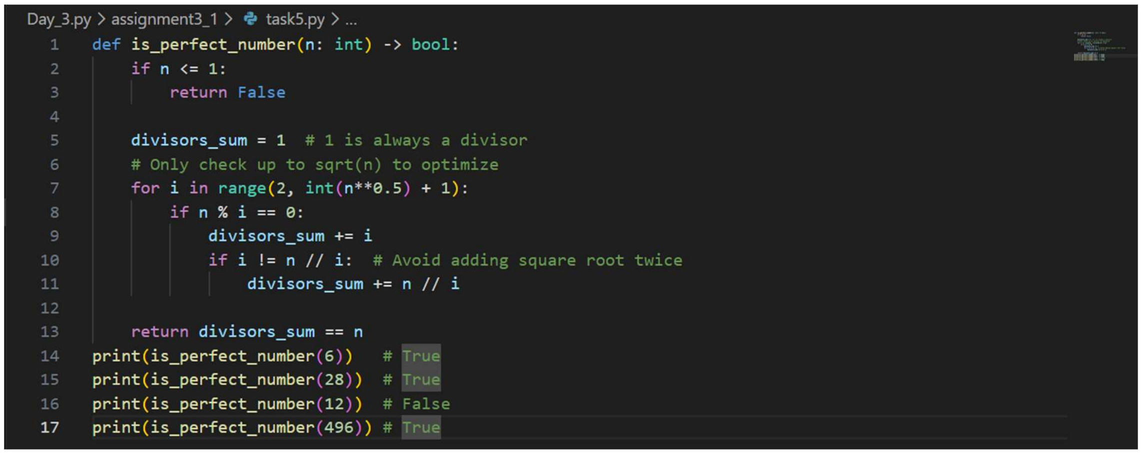
• Test the program with multiple inputs.

• 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.

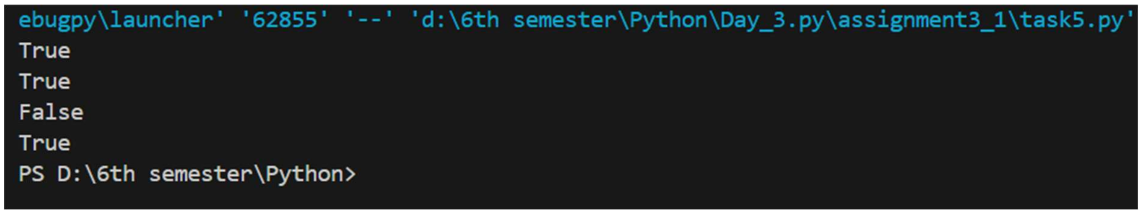
Code:



Day_3.py > assignment3_1 > task5.py > ...

```python
def is_perfect_number(n: int) -> bool:
    if n <= 1:
        return False

    divisors_sum = 1  # 1 is always a divisor
    # Only check up to sqrt(n) to optimize
    for i in range(2, int(n**0.5) + 1):
        if n % i == 0:
            divisors_sum += i
            if i != n // i:  # Avoid adding square root twice
                divisors_sum += n // i

    return divisors_sum == n
print(is_perfect_number(6))   # True
print(is_perfect_number(28))  # True
print(is_perfect_number(12))  # False
print(is_perfect_number(496)) # True
```

Output:



```
ebugpy\launcher' '62855' '--' 'd:\6th semester\Python\Day_3.py\assignment3_1\task5.py'
True
True
False
True
PS D:\6th semester\Python>
```

## 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:

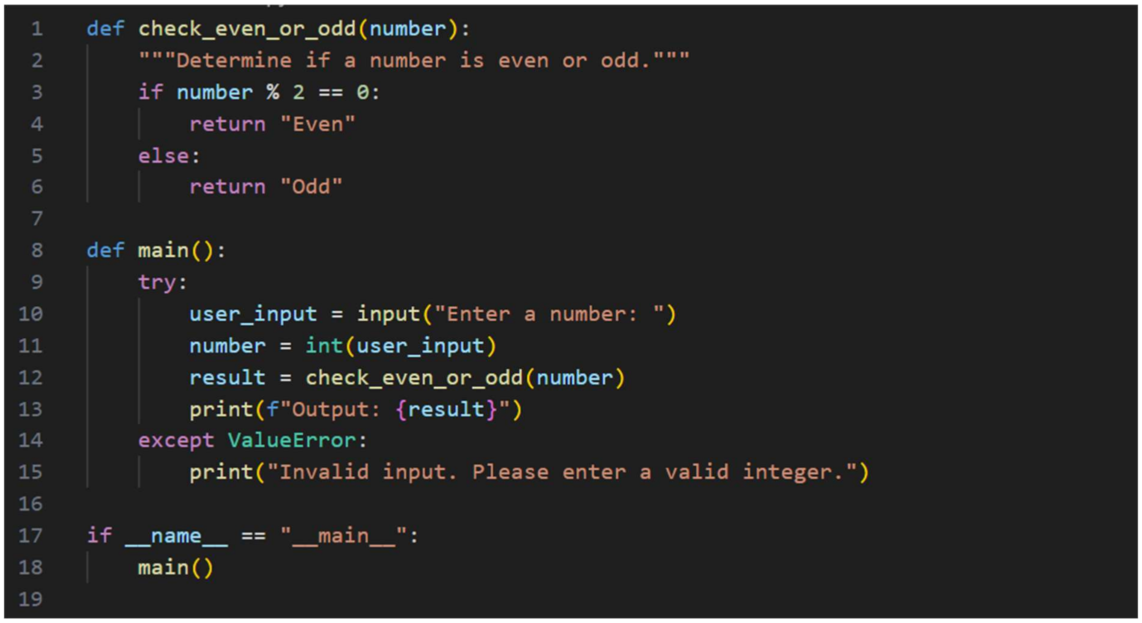
### Few-Shot Prompting (Even or Odd Classification with Validation)

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:

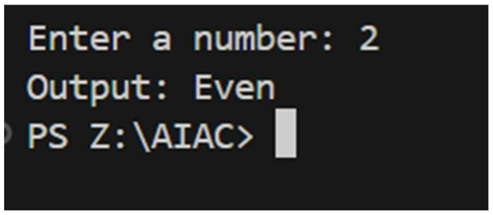
Write a Python program that determines whether a given input number is Even or Odd.
Examples:
Input: 8 → Output: Even
Input: 15 → Output: Odd
Input: 0 → Output: Even
The program should validate the input and handle invalid (non-integer) values gracefully.

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.