

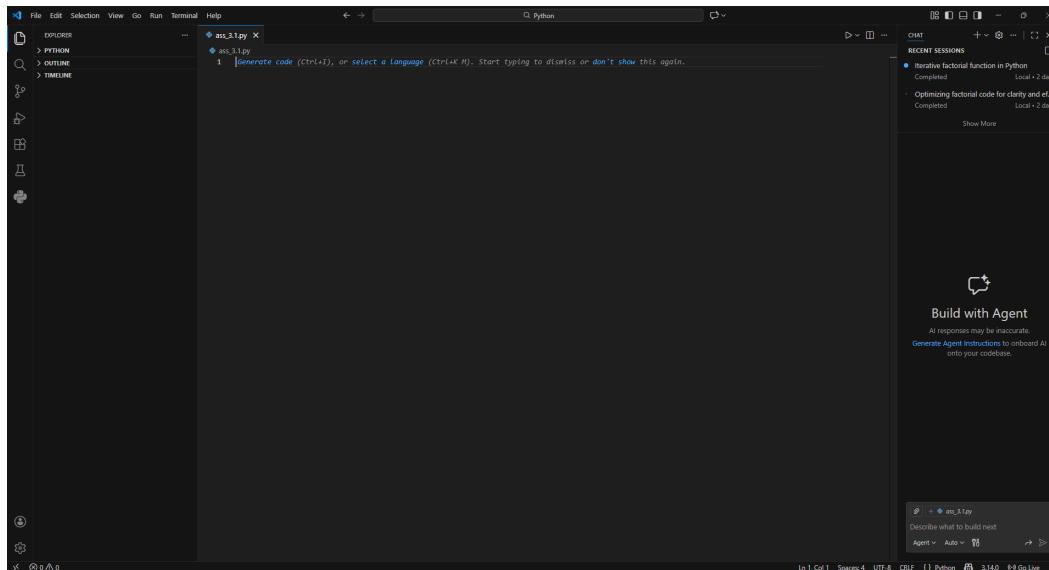
# Lab Assignment 3.1

## 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.

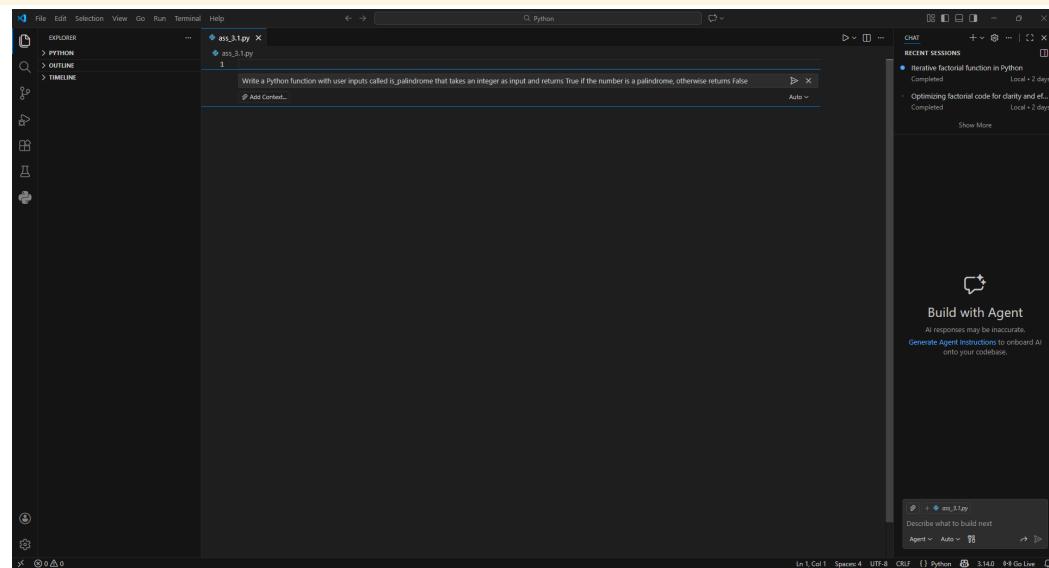
Task:

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



Prompt Instructed:

Write a Python function with user inputs called `is_palindrome` that takes an integer as input and returns True if the number is a palindrome, otherwise returns False



The screenshot shows two instances of the Visual Studio Code (VS Code) interface, both displaying the same Python file: `ass_3.1.py`. The file contains the following code:

```

1 def is_palindrome(num):
2     """
3         Check if an integer is a palindrome.
4     """
5     Args:
6         num: An integer to check
7
8     Returns:
9         True if the number is a palindrome, False otherwise
10    """
11    # Convert to string and remove the negative sign if present
12    num_str = str(abs(num))
13
14    # Check if the string is equal to its reverse
15    return num_str == num_str[::-1]
16
17
18    # User input
19    if __name__ == "__main__":
20        user_input = int(input("Enter an integer: "))
21        result = is_palindrome(user_input)
22        print(f"{user_input} is a palindrome: {result}")

```

The interface includes the following elements:

- Explorer View:** Shows the project structure with files `ass_3.1.py` and `ass_3.1.py ~`.
- Code Editor:** Displays the Python code.
- Terminal:** Shows the command `Ln 22, Col 53`.
- Status Bar:** Shows the file path `ass_3.1.py`, encoding `UTF-8`, line count `314.0`, and column count `4`.
- Chat Panel:** Shows a "Build with Agent" section with a message: "All responses may be inaccurate. Generate Agent Instructions to onboard AI onto your codebase."
- Recent Sessions:** Lists completed sessions: "Iterative factorial function in Python" and "Optimizing factorial code for clarity and ef...".

AI-generated code:

```

def is_palindrome(num):
    """
    Check if an integer is a palindrome.

```

```

Args:
    num: An integer to check

Returns:
    True if the number is a palindrome, False otherwise
"""

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

# User input
if __name__ == "__main__":
    user_input = input("Enter an integer: ")
    result = is_palindrome(user_input)
    print(f"{user_input} is a palindrome: {result}")

```

Testing Code with multiple inputs:

```

Enter an integer: 0
0 is a palindrome: True

Enter an integer: 12
12 is a palindrome: False

Enter an integer: 11
11 is a palindrome: True

Enter an integer: -12
-12 is a palindrome: False

Enter an integer: -121
-121 is a palindrome: False

```

Identify any logical errors or missing edge-case handling:

- **Negative Numbers:** The function treats -121 as a palindrome by using `abs(num)`. Mathematically, negative numbers are generally not considered palindromes because the negative sign breaks the symmetry. The prompt didn't specify this, so the AI made an arbitrary choice
- **Non-Integer Inputs:** No validation for non-integer inputs
- **Leading Zeros:** Numbers with leading zeros (like 00100) would be problematic, but in Python integers don't preserve leading zeros anyway
- **Single Digit Numbers:** The function correctly handles these, but it's worth noting as an edge case.

## Question 2: One-Shot Prompting (Factorial Calculation)

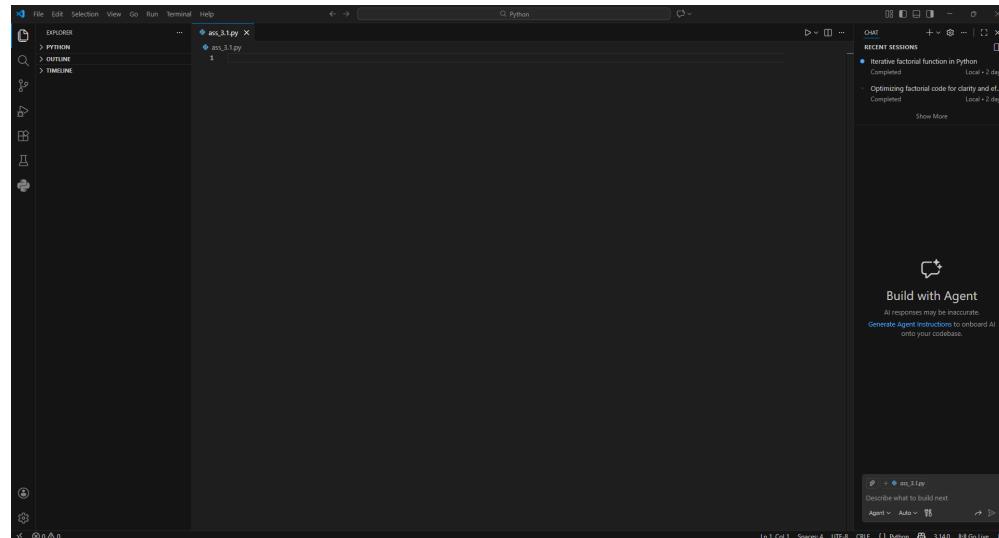
Write a one-shot prompt by providing one input-output example and ask the AI to generate a Python function to compute the factorial of a given number.

Example:

- Input: 5 → Output: 120

Task:

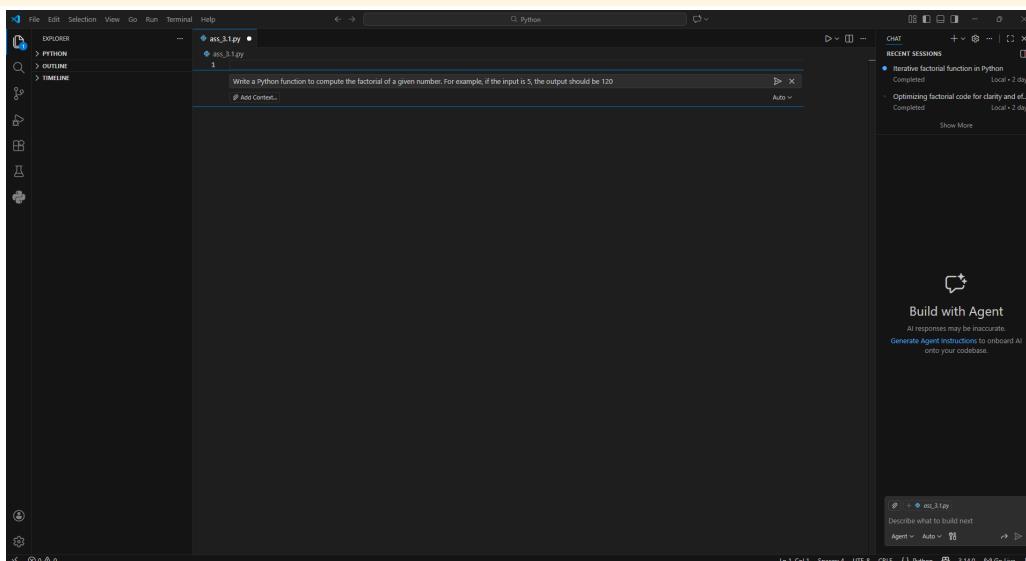
- Compare the generated code with a zero-shot solution.
- Examine improvements in clarity and correctness.



One shot:

Prompt Instructed:

Write a Python function to compute the factorial of a given number. For example, if the input is 5, the output should be 120



A screenshot of the Visual Studio Code interface. The main area shows a Python file named `ass_3.py` with the following code:

```
def factorial(n):
    """Compute the factorial of a given number."""
    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
print(factorial(5)) # Output: 120
```

The status bar at the bottom indicates "In 14, Col 35". On the right side, there is a "CHAT" panel titled "RECENT SESSIONS" with two entries: "Iterative factorial function in Python" and "Optimizing factorial code for clarity and ef...". Below the chat is a "Build with Agent" section with a message: "AI responses may be inaccurate. Generate Agent Instructions to onboard AI onto your codebase." A small input field says "Describe what to build next".

A second screenshot of the Visual Studio Code interface, identical to the first one, showing the same Python file `ass_3.py` and the same "Build with Agent" panel.

```
def factorial(n):
    """Compute the factorial of a given number."""
    if n < 0:
        raise ValueError("Factorial is not defined for negative numbers")
    if n == 0 or n == 1:
        return 1

    result = 1
```

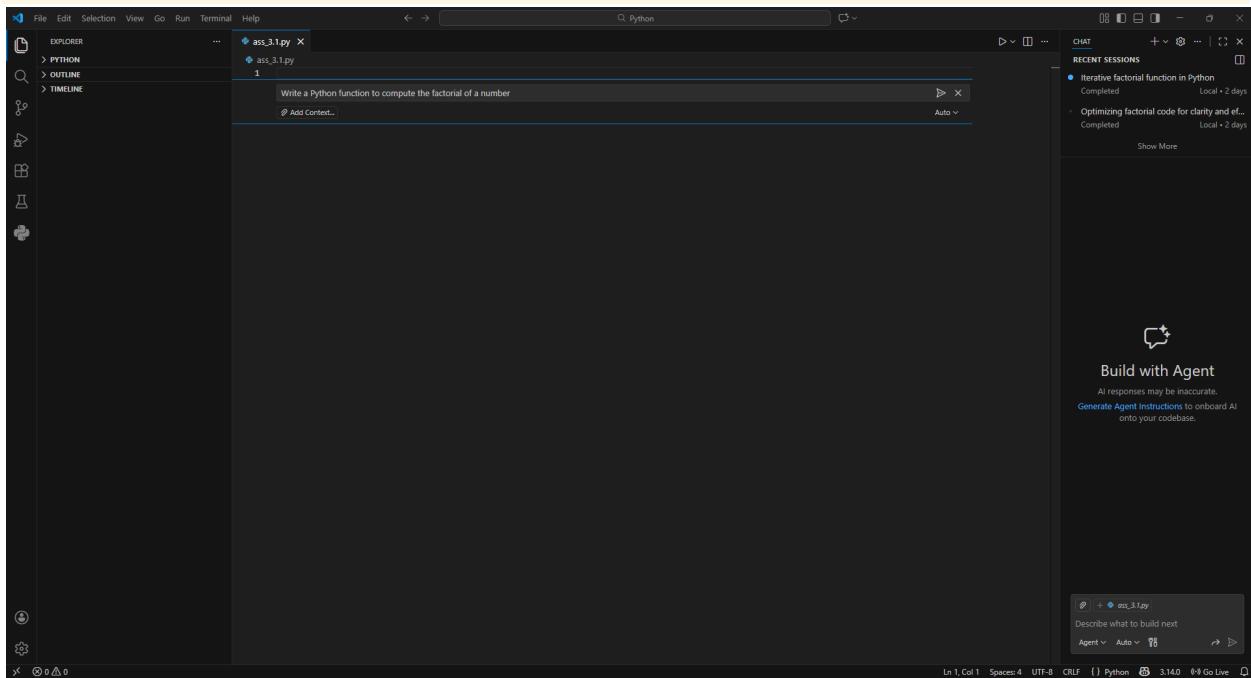
```
for i in range(2, n + 1):
    result *= i
return result

# Example usage
print(factorial(5)) # Output: 120
```

## Zero-Shot:

### Prompt Instructed:

Write a Python function to compute the factorial of a number



A screenshot of the Visual Studio Code (VS Code) interface. The main area shows a Python file named `ass_3.1.py` with the following code:

```
def factorial(n):
    """Compute the factorial of a number n."""
    if n < 0:
        raise ValueError("Factorial is not defined for negative numbers")
    if n == 0 or n == 1:
        return 1
    return n * factorial(n - 1)
```

The interface includes a sidebar with "EXPLORER", "PYTHON", "OUTLINE", and "TIMELINE" sections. On the right, there's a "CHAT" panel titled "RECENT SESSIONS" with two entries: "Iterative factorial function in Python" (Completed, Local + 2 days) and "Optimizing factorial code for clarity and ef..." (Completed, Local + 2 days). Below the chat is a "Build with Agent" section with a message about AI responses being inaccurate and a link to "Generate Agent Instructions". At the bottom, status bar information shows "Ln 7, Col 32, Spaces: 4, UTF-8, 3.14.0, 8:9 Go Live".

A second screenshot of the VS Code interface, showing the same Python file `ass_3.1.py` with the same code. The background is a different shade of dark gray compared to the first screenshot. The "CHAT", "RECENT SESSIONS", and "Build with Agent" sections are identical to the first screenshot. The status bar at the bottom also remains the same.

```
def factorial(n):
    """Compute the factorial of a number n."""
    if n < 0:
        raise ValueError("Factorial is not defined for negative numbers")
    if n == 0 or n == 1:
        return 1
    return n * factorial(n - 1)
```

Comparison: One-Shot vs Zero-Shot Factorial Implementation, examining improvements in clarity and correctness

Evaluation Criteria	One-Shot Implementation (Iterative)	Zero-Shot Implementation (Recursive)
<b>Algorithm Approach</b>	Iterative (for-loop)	Recursive (self-referential)
<b>Execution Transparency</b>	Linear, visible step-by-step computation	Abstract, distributed across stack frames
<b>Debugging Complexity</b>	Low - single execution path	High - multiple stack frames
<b>Memory Efficiency</b>	$O(1)$ - constant space	$O(n)$ - linear space (call stack)
<b>Scalability</b>	Unlimited by recursion depth	Limited by system recursion depth (~1000)
<b>Performance</b>	Lower overhead, faster execution	Higher overhead, slower for large n
<b>Error Handling</b>	Explicit validation for negatives	Explicit validation for negatives
<b>Production Readiness</b>	Industry-standard, robust	Theoretically elegant but impractical
<b>Maintainability</b>	High - linear logic flow	Medium - requires recursion understanding
<b>Edge Case Handling</b>	Explicit 0/1 base cases	Explicit 0/1 base cases
<b>Input Validation</b>	Negative number validation	Negative number validation
<b>System Resource Impact</b>	Minimal, predictable	Potentially high (stack overflow risk)
<b>Time Complexity</b>	$O(n)$	$O(n)$
<b>Space Complexity</b>	$O(1)$	$O(n)$

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

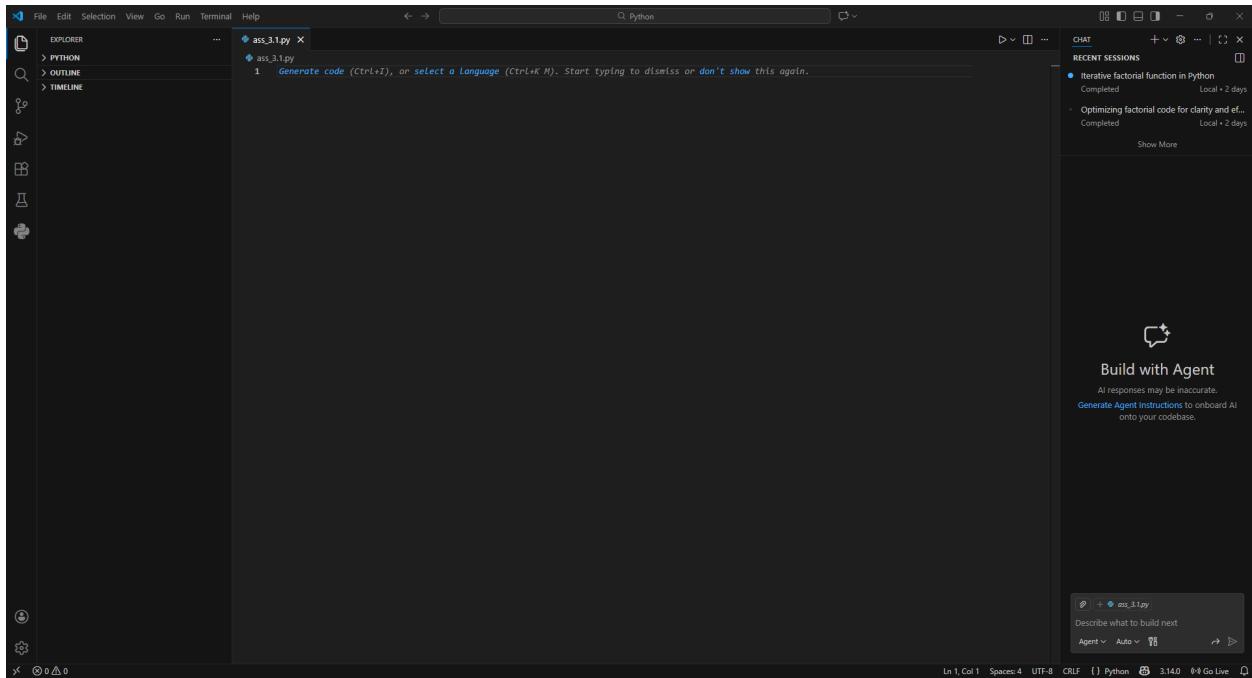
Write a few-shot prompt by providing multiple input-output examples to guide the AI in generating a Python function to check whether a given number is an Armstrong number.

Examples:

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

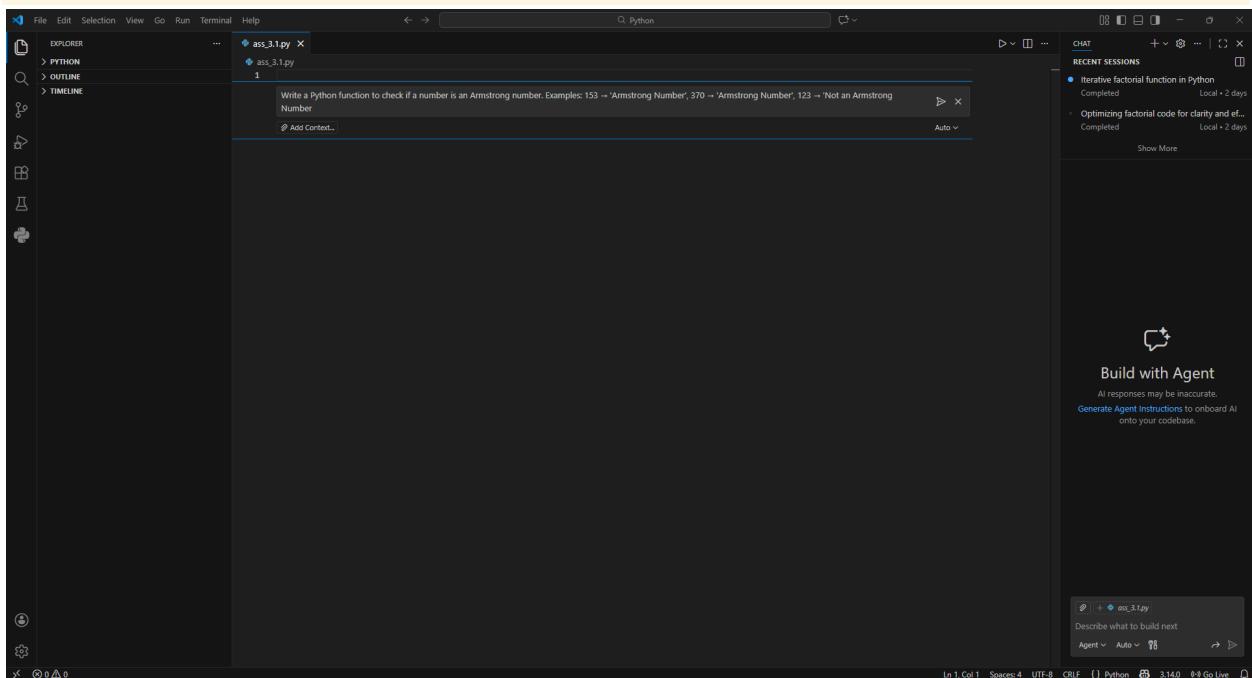
Task:

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



### Prompt Instructed:

Write a Python function to check if a number is an Armstrong number.  
 Examples: 153 → 'Armstrong Number', 370 → 'Armstrong Number', 123 → 'Not an Armstrong Number'



A screenshot of the Visual Studio Code interface. The left sidebar shows the Explorer, Python, Outline, and Timeline tabs. The main editor window displays a Python file named 'ass\_3.py' with the following code:

```
def is_armstrong_number(n):
    """
    Check if a number is an Armstrong number.
    An Armstrong number equals the sum of its digits each raised to the power of the number of digits.
    """
    digits = str(n)
    num_digits = len(digits)
    sum_of_powers = sum(int(digit) ** num_digits for digit in digits)

    if sum_of_powers == n:
        return "Armstrong Number"
    else:
        return "Not an Armstrong Number"

# Test cases
print(is_armstrong_number(153)) # Armstrong Number
print(is_armstrong_number(370)) # Armstrong Number
print(is_armstrong_number(123)) # Not an Armstrong Number
```

The status bar at the bottom indicates 'Ln 19, Col 59' and 'Python'. The right sidebar includes a 'CHAT' section with recent sessions and a 'Build with Agent' feature.

A second screenshot of the Visual Studio Code interface, showing the same Python file 'ass\_3.py' with the same code as the first screenshot. The interface and status bar are identical.

Code:

```
def is_armstrong_number(n):
    """
    Check if a number is an Armstrong number.
    An Armstrong number equals the sum of its digits each raised to the
    power of the number of digits.
    """

    digits = str(n)
```

```

num_digits = len(digits)
sum_of_powers = sum(int(digit) ** num_digits for digit in digits)

if sum_of_powers == n:
    return "Armstrong Number"
else:
    return "Not an Armstrong Number"

# Test cases
print(is_armstrong_number(153)) # Armstrong Number
print(is_armstrong_number(370)) # Armstrong Number
print(is_armstrong_number(123)) # Not an Armstrong Number

```

Examples and boundary values:

Enter a number: 153  
Armstrong Number

Enter a number: 999  
Not an Armstrong Number

Enter a number: 9474  
Armstrong Number

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

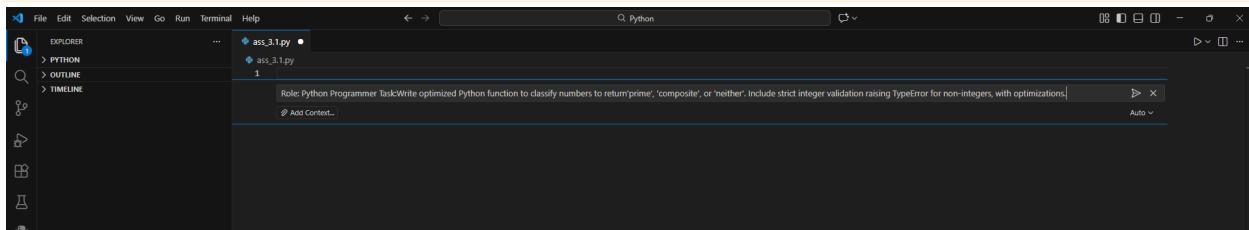
Design a context-managed prompt with clear instructions and constraints to generate an optimized Python program that classifies a number as prime, composite, or neither.

Task:

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

Prompt Instructed:

Role: Python Programmer Task: Write optimized Python function to classify numbers to return 'prime', 'composite', or 'neither'. Include strict integer validation raising TypeError for non-integers, with optimizations.



```
def classify_number(n):
    """
    Classify a number as 'prime', 'composite', or 'neither'.

    Args:
        n: The number to classify (must be an integer)

    Returns:
        str: 'prime', 'composite', or 'neither'

    Raises:
        TypeError: If n is not an integer
    """
    if not isinstance(n, int) or isinstance(n, bool):
        raise TypeError(f"Expected integer, got {type(n).__name__}")

    if n < 2:
        return 'neither'

    if n == 2:
        return 'prime'

    if n % 2 == 0:
        return 'composite'

    # Check odd divisors up to sqrt(n)
    i = 3
    while i * i <= n:
        if n % i == 0:
            return 'composite'
        i += 2

    return 'prime'
```

Code:

```
def classify_number(n):
    """
    Classify a number as 'prime', 'composite', or 'neither'.

    Args:
        n: The number to classify (must be an integer)

    Returns:
        str: 'prime', 'composite', or 'neither'

    Raises:
        TypeError: If n is not an integer
    """
    if not isinstance(n, int) or isinstance(n, bool):
        raise TypeError(f"Expected integer, got {type(n).__name__}")

    if n < 2:
        return 'neither'

    if n == 2:
        return 'prime'

    if n % 2 == 0:
```

```

        return 'composite'

# Check odd divisors up to sqrt(n)
i = 3
while i * i <= n:
    if n % i == 0:
        return 'composite'
    i += 2

return 'prime'

```

## Zero-Shot Prompt:

Write Python code to classify a number **as prime or composite**

A screenshot of the Visual Studio Code interface. The Explorer sidebar shows a folder named 'ass\_3.1'. The main editor area contains the following Python code:

```

def classify_number(n):
    """
    Classify a number as prime or composite.

    Args:
        n: Integer to classify

    Returns:
        String indicating if the number is prime or composite
    """
    if n < 2:
        return "Neither prime nor composite"
    if n == 2:
        return "Prime"
    if n % 2 == 0:
        return "Composite"

    # Check odd divisors up to sqrt(n)
    for i in range(3, int(n**0.5) + 1, 2):
        if n % i == 0:
            return "Composite"
    return "Prime"

```

A screenshot of the Visual Studio Code interface, showing the same file 'ass\_3.1.py' but with more extensive documentation and testing code. The code now includes a detailed docstring and type annotations:

```

def classify_number(n: int) -> str:
    """
    Classify a number as prime or composite.

    Args:
        n: Integer to classify

    Returns:
        String indicating if the number is prime or composite
    """
    if n < 2:
        return "Neither prime nor composite"
    if n == 2:
        return "Prime"
    if n % 2 == 0:
        return "Composite"

    # Check odd divisors up to sqrt(n)
    for i in range(3, int(n**0.5) + 1, 2):
        if n % i == 0:
            return "Composite"
    return "Prime"

# Test the function
if __name__ == "__main__":
    test_numbers = [2, 3, 4, 10, 17, 25, 29]

    for num in test_numbers:
        print(f"(num): {classify_number(num)}")

```

## Optimized changes (context-managed):

- **Algorithm:** Early case for 2, early even-number rejection, loops only odd divisors
- **Implementation:** Adds full docstring, clear error messages, comprehensive testing
- **Validation:** Strict integer type checking with `TypeError` for invalid inputs

**Performance impact:** Same  $O(\sqrt{n})$  complexity but halves divisor checks (skips evens) with early termination; ~2x faster for large odd numbers.

**Readability/correctness:** Clearer intent with structured flow, safer validation prevents crashes, better maintainability with documentation.

**Tradeoffs/next steps:** Still deterministic trial division; for huge numbers consider Miller-Rabin (probabilistic) or optimized sieves for batch classification.

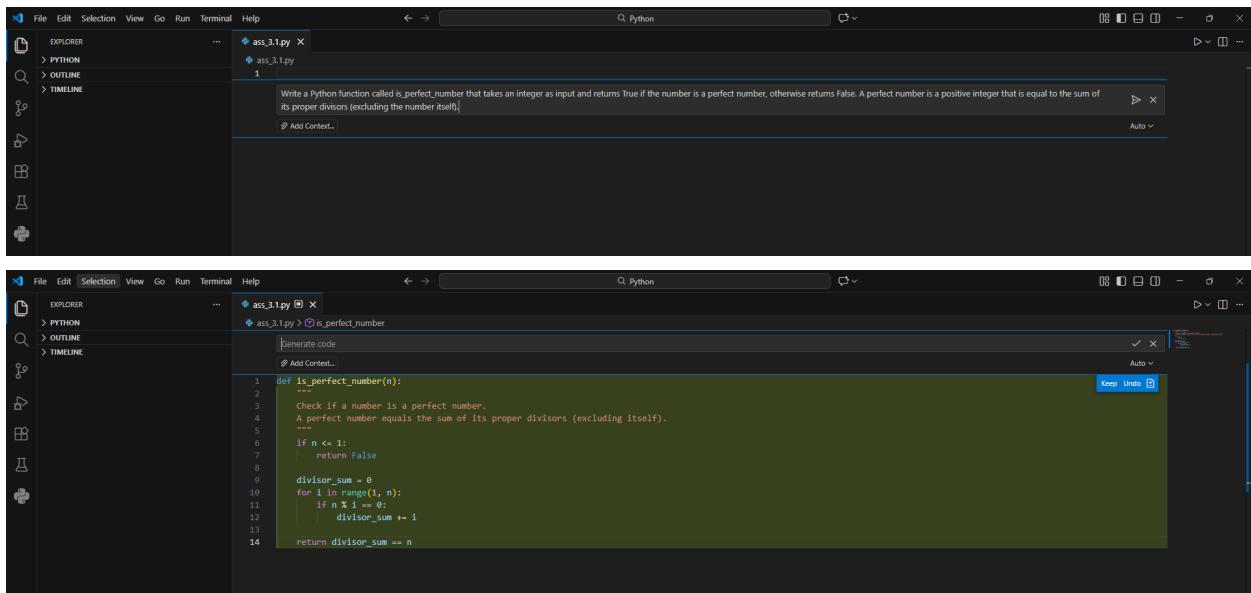
## Question 5: Zero-Shot Prompting (Perfect Number Check)

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 Instructed:



The image shows two screenshots of the Microsoft Visual Studio Code interface. The top screenshot shows the 'EXPLORER' view with a file named 'ass\_3.1.py'. A tooltip is displayed over the file, asking to write a Python function called 'is\_perfect\_number' that takes an integer as input and returns True if the number is a perfect number, otherwise returns False. The bottom screenshot shows the 'CODE' editor with the same file open. The AI has generated the following code:

```
def is_perfect_number(n):
    """
    Check if a number is a perfect number.
    A perfect number equals the sum of its proper divisors (excluding itself).
    """

    if n <= 1:
        return False

    divisor_sum = 0
    for i in range(1, n):
        if n % i == 0:
            divisor_sum += i

    return divisor_sum == n
```

AI-generated code:

```
def is_perfect_number(n):
    """
    Check if a number is a perfect number.
    A perfect number equals the sum of its proper divisors (excluding itself).
    """

    if n <= 1:
        return False
```

```

divisor_sum = 0
for i in range(1, n):
    if n % i == 0:
        divisor_sum += i

return divisor_sum == n

```

Testing it with multiple inputs:

```

Test Results:
is_perfect_number(6) = True
is_perfect_number(28) = True
is_perfect_number(496) = True
is_perfect_number(8128) = True
is_perfect_number(12) = False
is_perfect_number(0) = False
is_perfect_number(1) = False
is_perfect_number(-6) = False
is_perfect_number(10) = False
is_perfect_number(33550336) = True
PS C:\Users\Himes\Documents\.Dev\Python>

```

Missing Conditions:

- No input validation (fails on strings, floats, None)
- No early termination when sum exceeds n
- No handling for very large numbers (impractical for  $>10^7$ )

Inefficiencies:

- $O(n)$  time complexity → checks every number 1 to  $n-1$
- No square root optimization → should only check up to  $\sqrt{n}$
- No divisor pairing → duplicates checks ( $i$  and  $n/i$ )
- No mathematical property use → perfect numbers follow  $2^{(p-1)} \times (2^p - 1)$  pattern
- Worst case:  $n=33,550,336 \rightarrow 33.5$  million iterations instead of ~5,800 with optimization.

## 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 Instructed:

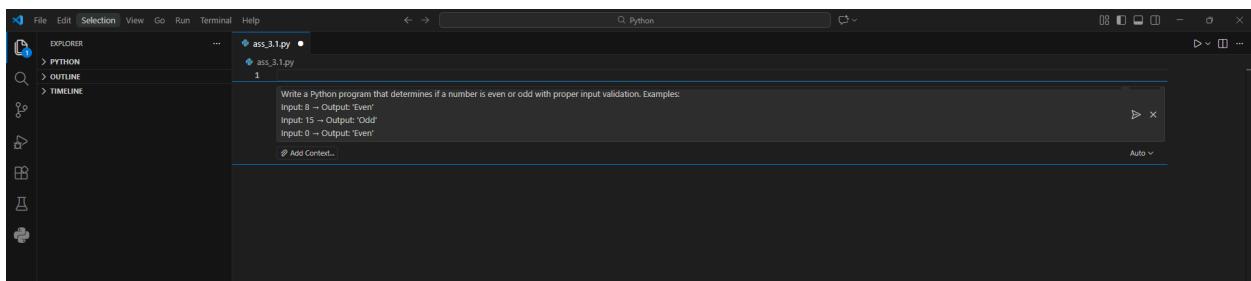
Write a Python program that determines if a number is even or odd with proper input validation. Examples:

Input: 8 → Output: 'Even'

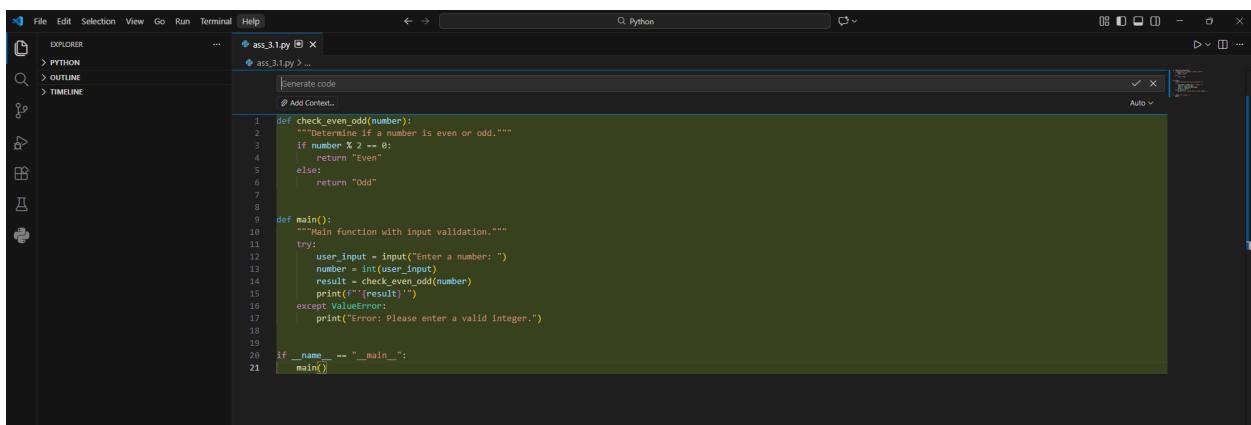
Input: 15 → Output: 'Odd'

Input: 0 → Output: 'Even'

The program should handle invalid inputs gracefully and provide clear error messages.



## Code:



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

def main():
    """Main function with input validation."""
    try:
        user_input = input("Enter a number: ")
        number = int(user_input)
```

```

    result = check_even_odd(number)
    print(f'{result}')
except ValueError:
    print("Error: Please enter a valid integer.")

if __name__ == "__main__":
    main()

```

### Test Cases:

Negative Numbers: All work correctly

- -8 → "Even"
- -15 → "Odd"
- -0 → "Even"

Non-Integer Inputs:

- 3.5 → "Error: Please enter a valid integer."
- "abc" → "Error: Please enter a valid integer."
- "" → "Error: Please enter a valid integer."
- "2.0" → "Error: Please enter a valid integer." (should accept 2.0)

Edge Cases:

- " 8 " → "Error: Please enter a valid integer." (spaces cause failure)
- "+8" → "Even"
- "0" → "Even"

### Examples improved code:

- Output format matches examples exactly ("Even"/"Odd" strings, not booleans)
- Zero handled correctly (0 → "Even" example ensured proper modulo logic)
- Clear function structure (separation of logic and validation)
- Error handling included (prompt mentioned "proper validation")

### Missing from examples:

- No handling for floats like 2.0 (int() fails on floats)
- No whitespace stripping (examples didn't show spaces)

### Comparison to Few-Shot Goal:

Conditions met:

- Returns "Even"/"Odd" exactly as examples show
- Handles all three provided examples (8, 15, 0)
- Includes error messages for invalid inputs

Conditions Not met:

- Doesn't accept floats that are integers (2.0)
- Doesn't strip whitespace
- Less robust than ideal few-shot implementation