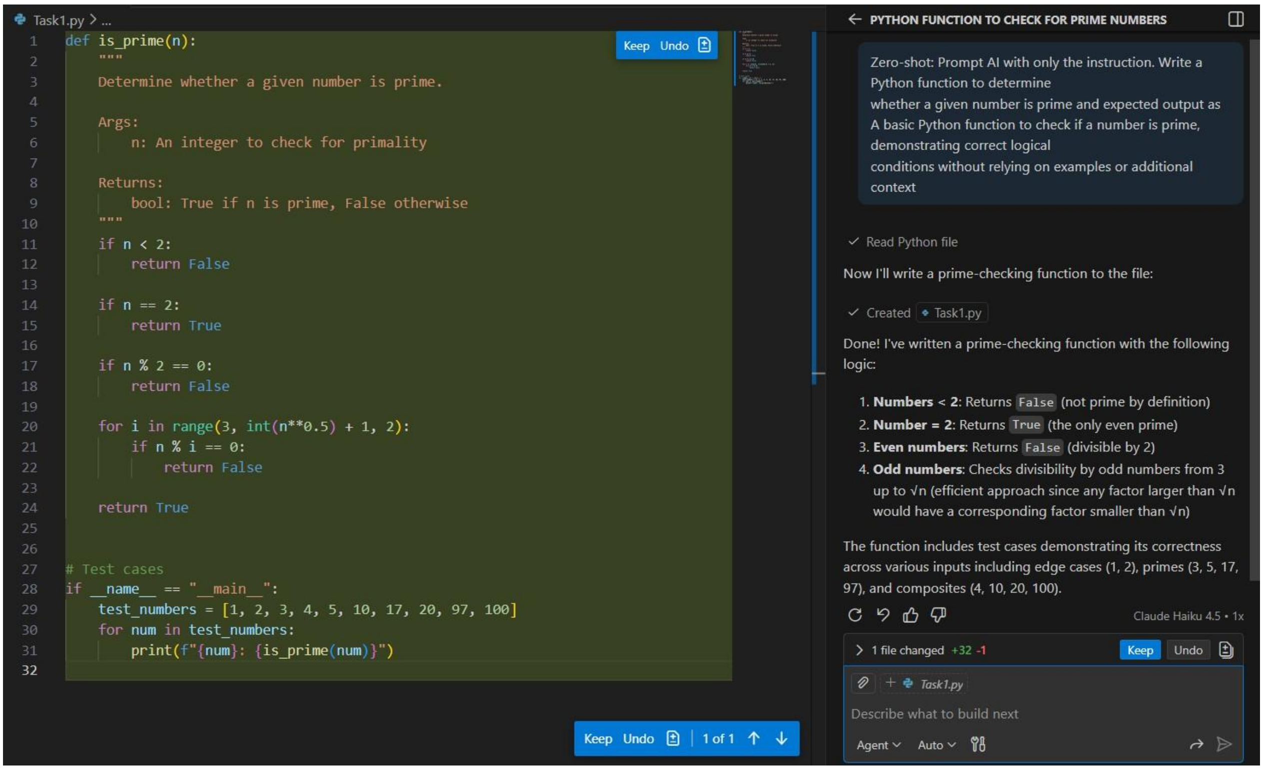
Assignment 4.2

Name: S.Varshith
HTNO: 2303A51837

Task01-Zero-shot: Prompt AI with only the instruction. Write a Python function to determine whether a given number is prime


Task1.py > ...
Keep
Undo
Keep
Undo
1 of 1
PYTHON FUNCTION TO CHECK FOR PRIME NUMBERS
Keep
Undo
Task1.py
Describe what to build next
Agent
Auto

```python
def is_prime(n):
    """
    Determine whether a given number is prime.

    Args:
        n: An integer to check for primality

    Returns:
        bool: True if n is prime, False otherwise
    """
    if n < 2:
        return False

    if n == 2:
        return True

    if n % 2 == 0:
        return False

    for i in range(3, int(n**0.5) + 1, 2):
        if n % i == 0:
            return False

    return True


# Test cases
if __name__ == "__main__":
    test_numbers = [1, 2, 3, 4, 5, 10, 17, 20, 97, 100]
    for num in test_numbers:
        print(f"{num}: {is_prime(num)}")

```

Zero-shot: Prompt AI with only the instruction. Write a Python function to determine
whether a given number is prime and expected output as A basic Python function to check if a number is prime, demonstrating correct logical
conditions without relying on examples or additional context

✓ Read Python file

Now I'll write a prime-checking function to the file:

✓ Created Task1.py

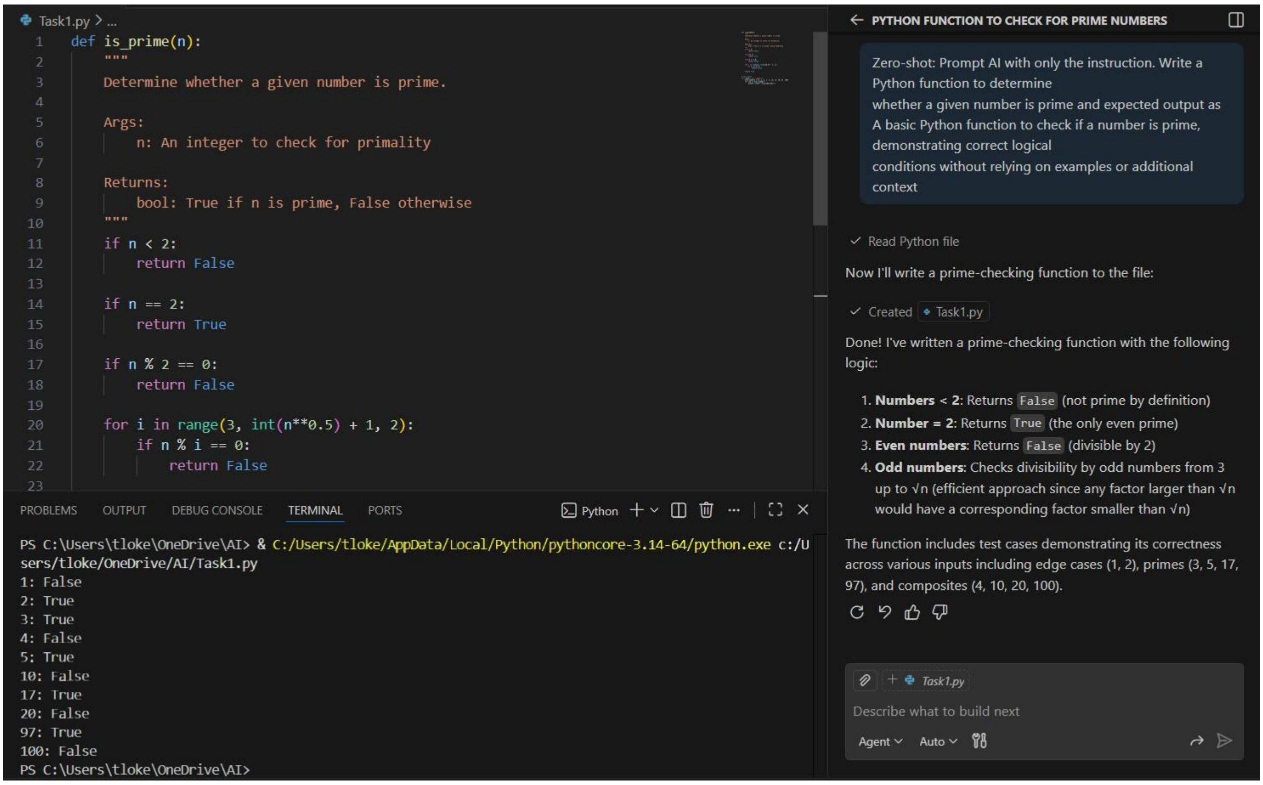
Done! I've written a prime-checking function with the following logic:

1. **Numbers < 2**: Returns `False` (not prime by definition)
2. **Number = 2**: Returns `True` (the only even prime)
3. **Even numbers**: Returns `False` (divisible by 2)
4. **Odd numbers**: Checks divisibility by odd numbers from 3 up to √n (efficient approach since any factor larger than √n would have a corresponding factor smaller than √n)

The function includes test cases demonstrating its correctness across various inputs including edge cases (1, 2), primes (3, 5, 17, 97), and composites (4, 10, 20, 100).

Claude Haiku 4.5 • 1x

1 file changed +32 -1



```python
def is_prime(n):
    """
    Determine whether a given number is prime.

    Args:
        n: An integer to check for primality

    Returns:
        bool: True if n is prime, False otherwise
    """
    if n < 2:
        return False

    if n == 2:
        return True

    if n % 2 == 0:
        return False

    for i in range(3, int(n**0.5) + 1, 2):
        if n % i == 0:
            return False
```

```
PS C:\Users\tloke\OneDrive\AI> & C:/Users/tloke/AppData/Local/Python/pythoncore-3.14-64/python.exe c:/U
sers/tloke/OneDrive/AI/Task1.py
1: False
2: True
3: True
4: False
5: True
10: False
17: True
20: False
97: True
100: False
PS C:\Users\tloke\OneDrive\AI>
```

Zero-shot: Prompt AI with only the instruction. Write a Python function to determine
whether a given number is prime and expected output as A basic Python function to check if a number is prime, demonstrating correct logical
conditions without relying on examples or additional context

✓ Read Python file

Now I'll write a prime-checking function to the file:

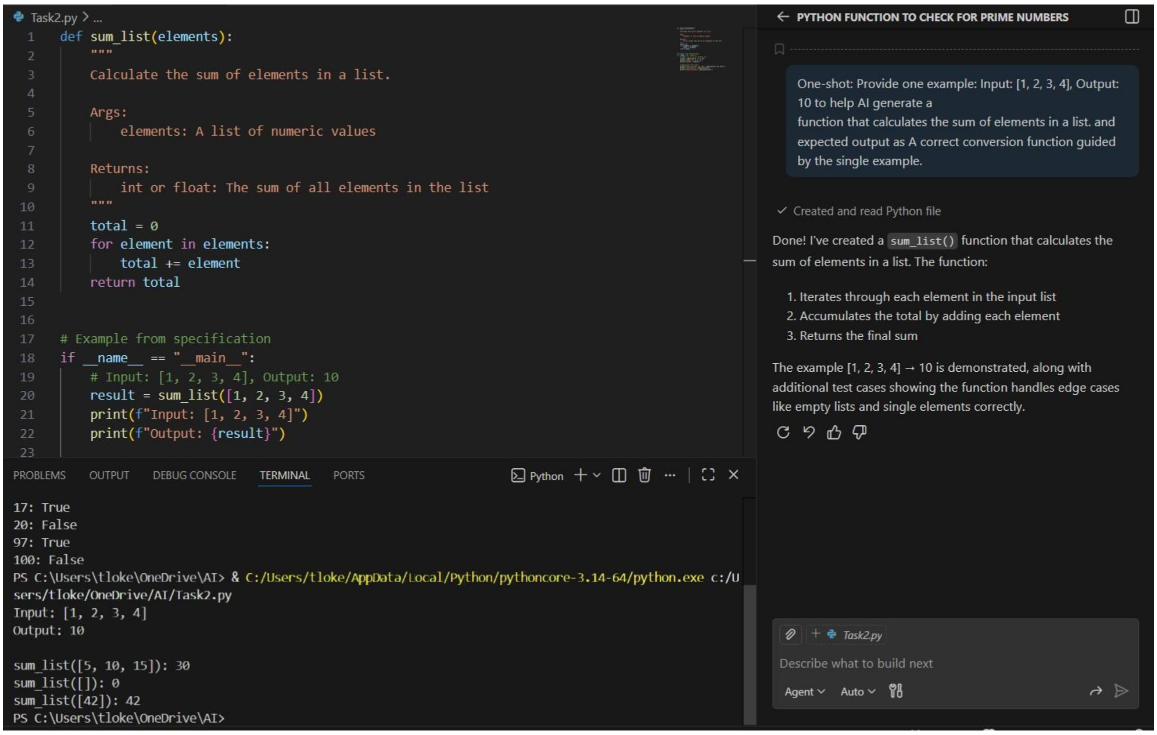
✓ Created Task1.py

Done! I've written a prime-checking function with the following logic:

1. **Numbers < 2**: Returns `False` (not prime by definition)
2. **Number = 2**: Returns `True` (the only even prime)
3. **Even numbers**: Returns `False` (divisible by 2)
4. **Odd numbers**: Checks divisibility by odd numbers from 3 up to √n (efficient approach since any factor larger than √n would have a corresponding factor smaller than √n)

The function includes test cases demonstrating its correctness across various inputs including edge cases (1, 2), primes (3, 5, 17, 97), and composites (4, 10, 20, 100).

Task02-One-shot: Provide one example: Input: [1, 2, 3, 4], Output: 10 to help AI generate a function that calculates the sum of elements in a list.



```python
def sum_list(elements):
    """
    Calculate the sum of elements in a list.

    Args:
        elements: A list of numeric values

    Returns:
        int or float: The sum of all elements in the list
    """
    total = 0
    for element in elements:
        total += element
    return total


# Example from specification
if __name__ == "__main__":
    # Input: [1, 2, 3, 4], Output: 10
    result = sum_list([1, 2, 3, 4])
    print(f"Input: [1, 2, 3, 4]")
    print(f"Output: {result}")
```

```
17: True
20: False
97: True
100: False
PS C:\Users\tloke\OneDrive\AI> & C:/Users/tloke/AppData/Local/Python/pythoncore-3.14-64/python.exe c:/U
sers/tloke/OneDrive/AI/Task2.py
Input: [1, 2, 3, 4]
Output: 10

sum_list([5, 10, 15]): 30
sum_list([]): 0
sum_list([42]): 42
PS C:\Users\tloke\OneDrive\AI>
```

One-shot: Provide one example: Input: [1, 2, 3, 4], Output: 10 to help AI generate a
function that calculates the sum of elements in a list. and expected output as A correct conversion function guided by the single example.

✓ Created and read Python file

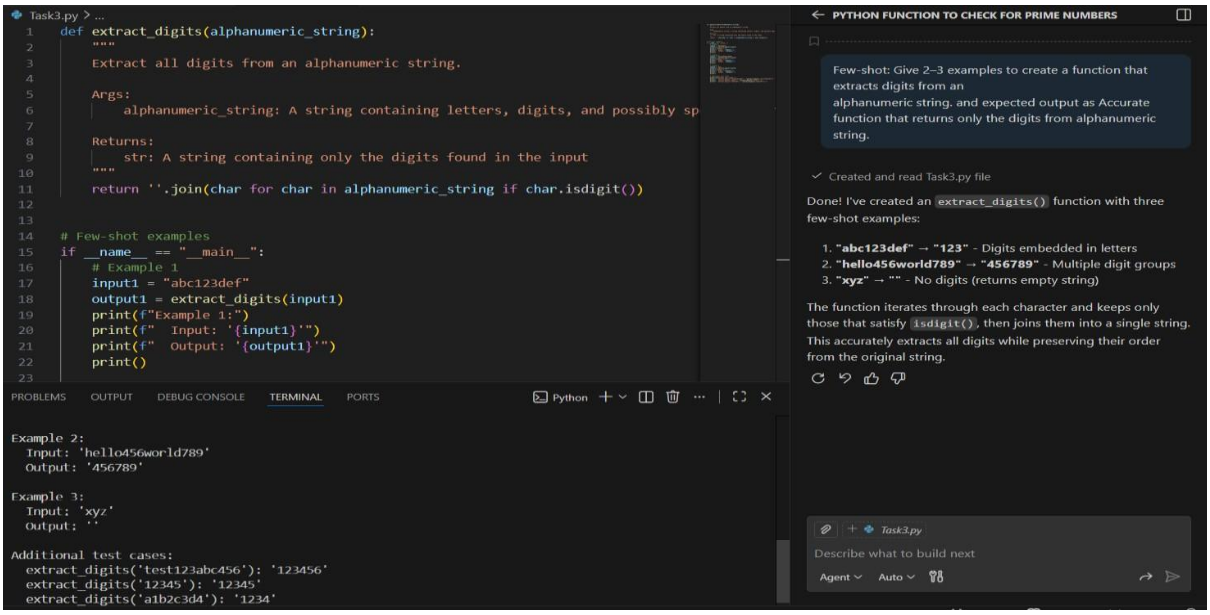
Done! I've created a `sum_list()` function that calculates the sum of elements in a list. The function:

1. Iterates through each element in the input list
2. Accumulates the total by adding each element
3. Returns the final sum

The example [1, 2, 3, 4] → 10 is demonstrated, along with additional test cases showing the function handles edge cases like empty lists and single elements correctly.

n

Task3-Give 2–3 examples to create a function that extracts digits from an alphanumeric string



```python
def extract_digits(alphanumeric_string):
    """
    Extract all digits from an alphanumeric string.

    Args:
        alphanumeric_string: A string containing letters, digits, and possibly sp

    Returns:
        str: A string containing only the digits found in the input
    """
    return ''.join(char for char in alphanumeric_string if char.isdigit())


# Few-shot examples
if __name__ == "__main__":
    # Example 1
    input1 = "abc123def"
    output1 = extract_digits(input1)
    print(f"Example 1:")
    print(f"  Input: '{input1}'")
    print(f"  Output: '{output1}'")
    print()
```

```
Example 2:
  Input: 'hello456world789'
  Output: '456789'

Example 3:
  Input: 'xyz'
  Output: ''

Additional test cases:
  extract_digits('test123abc456'): '123456'
  extract_digits('12345'): '12345'
  extract_digits('a1b2c3d4'): '1234'
```

Few-shot: Give 2–3 examples to create a function that extracts digits from an alphanumeric string. and expected output as Accurate function that returns only the digits from alphanumeric string.

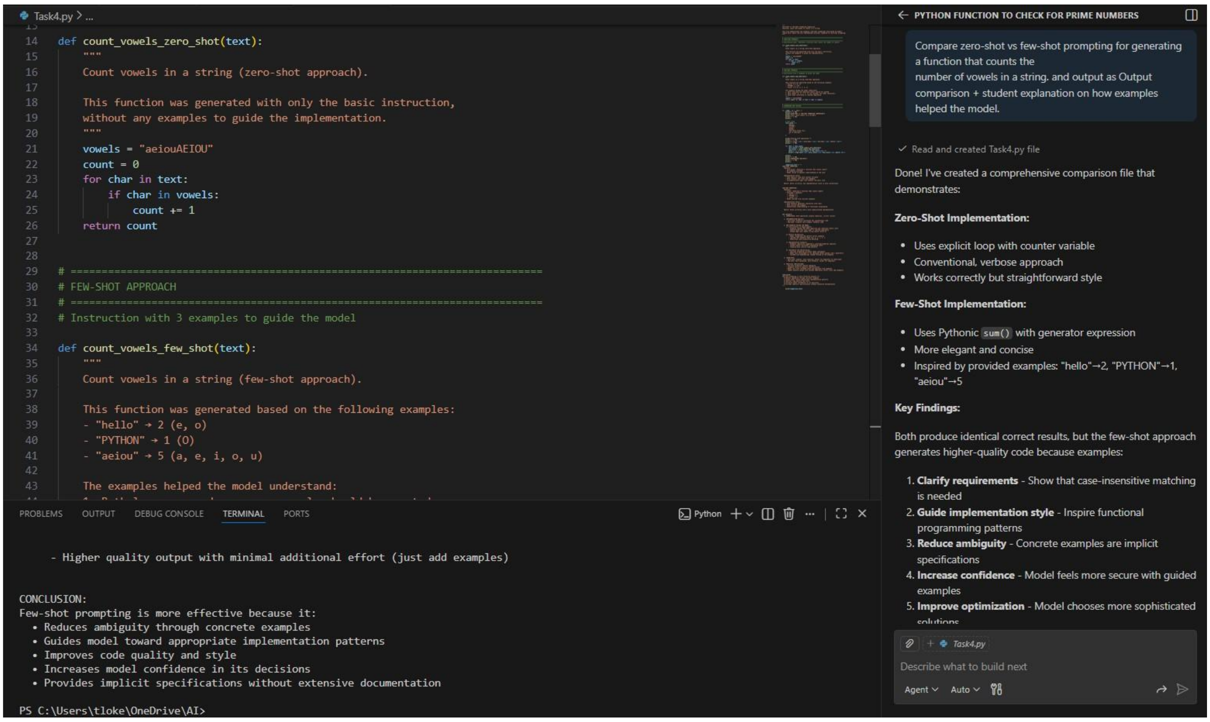
✓ Created and read Task3.py file

Done! I've created an `extract_digits()` function with three few-shot examples:

1. **"abc123def"** → **"123"** - Digits embedded in letters
2. **"hello456world789"** → **"456789"** - Multiple digit groups
3. **"xyz"** → **""** - No digits (returns empty string)

The function iterates through each character and keeps only those that satisfy `isdigit()`, then joins them into a single string. This accurately extracts all digits while preserving their order from the original string.

Task4-Compare zero-shot vs few-shot prompting for generating a function that counts the number of vowels in a string.



```python
def count_vowels_zero_shot(text):
    """
    Count vowels in a string (zero-shot approach).

    This function was generated with only the basic instruction,
    without any examples to guide the implementation.
    """
    vowels = "aeiouAEIOU"
    count = 0
    for char in text:
        if char in vowels:
            count += 1
    return count


# ========================================================================
# FEW-SHOT APPROACH
# ========================================================================
# Instruction with 3 examples to guide the model

def count_vowels_few_shot(text):
    """
    Count vowels in a string (few-shot approach).

    This function was generated based on the following examples:
    - "hello" → 2 (e, o)
    - "PYTHON" → 1 (O)
    - "aeiou" → 5 (a, e, i, o, u)

    The examples helped the model understand:
```

```
    - Higher quality output with minimal additional effort (just add examples)


CONCLUSION:
Few-shot prompting is more effective because it:
  • Reduces ambiguity through concrete examples
  • Guides model toward appropriate implementation patterns
  • Improves code quality and style
  • Increases model confidence in its decisions
  • Provides implicit specifications without extensive documentation

PS C:\Users\tloke\OneDrive\AI>
```

Compare zero-shot vs few-shot prompting for generating a function that counts the number of vowels in a string. and output as Output comparison + student explanation on how examples helped the model.

✓ Read and created Task4.py file

Done! I've created a comprehensive comparison file that demonstrates:

Zero-Shot Implementation:

- Uses explicit loop with counter variable
- Conventional, verbose approach
- Works correctly but straightforward style

Few-Shot Implementation:

- Uses Pythonic `sum()` with generator expression
- More elegant and concise
- Inspired by provided examples: "hello"→2, "PYTHON"→1, "aeiou"→5

Key Findings:

Both produce identical correct results, but the few-shot approach generates higher-quality code because examples:

1. **Clarify requirements** - Show that case-insensitive matching is needed
2. **Guide implementation style** - Inspire functional programming patterns
3. **Reduce ambiguity** - Concrete examples are implicit specifications
4. **Increase confidence** - Model feels more secure with guided examples
5. **Improve optimization** - Model chooses more sophisticated solutions

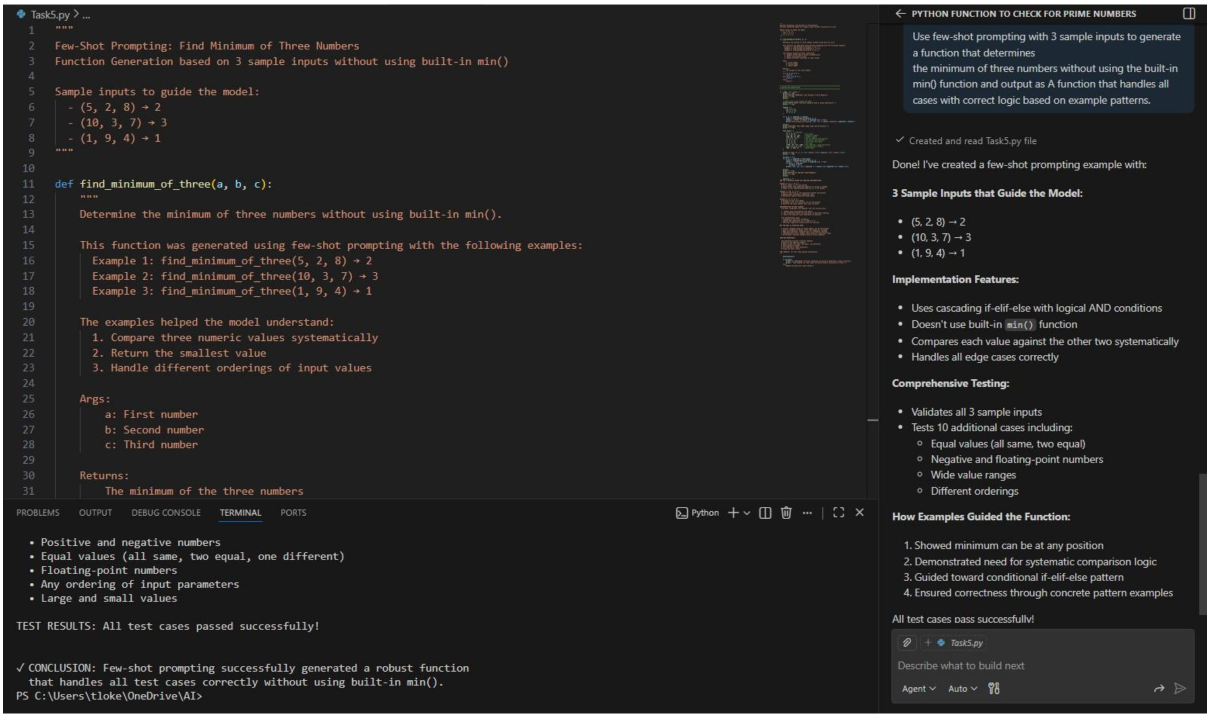
Explanation: In zero-shot prompting, the model generates the function using its general understanding of programming and vowels. Since no examples are given, the model decides on its own how to handle cases like uppercase letters.

In few-shot prompting, the examples clearly show what the expected input and output look like. From these examples, the model learns important rules such as:

- Uppercase vowels should be counted
- Strings without vowels should return 0

Because of this guidance, the model produces a more accurate and user-aligned function. Examples act like hints, reducing ambiguity and improving correctness.

Task5-Use few-shot prompting with 3 sample inputs to generate a function that determines the minimum of three numbers without using the built-in min() function.



```python
"""
Few-Shot Prompting: Find Minimum of Three Numbers
Function Generation based on 3 sample inputs without using built-in min()

Sample inputs to guide the model:
  - (5, 2, 8) → 2
  - (10, 3, 7) → 3
  - (1, 9, 4) → 1
"""


def find_minimum_of_three(a, b, c):
    """
    Determine the minimum of three numbers without using built-in min().

    This function was generated using few-shot prompting with the following examples:
      Example 1: find_minimum_of_three(5, 2, 8) → 2
      Example 2: find_minimum_of_three(10, 3, 7) → 3
      Example 3: find_minimum_of_three(1, 9, 4) → 1

    The examples helped the model understand:
      1. Compare three numeric values systematically
      2. Return the smallest value
      3. Handle different orderings of input values

    Args:
        a: First number
        b: Second number
        c: Third number

    Returns:
        The minimum of the three numbers
```

```
  • Positive and negative numbers
  • Equal values (all same, two equal, one different)
  • Floating-point numbers
  • Any ordering of input parameters
  • Large and small values

TEST RESULTS: All test cases passed successfully!


✓ CONCLUSION: Few-shot prompting successfully generated a robust function
  that handles all test cases correctly without using built-in min().
PS C:\Users\tloke\OneDrive\AI>
```

Use few-shot prompting with 3 sample inputs to generate a function that determines
the minimum of three numbers without using the built-in min() function and output as A function that handles all cases with correct logic based on example patterns.

✓ Created and read Task5.py file

Done! I've created a few-shot prompting example with:

3 Sample Inputs that Guide the Model:

- (5, 2, 8) → 2
- (10, 3, 7) → 3
- (1, 9, 4) → 1

Implementation Features:

- Uses cascading if-elif-else with logical AND conditions
- Doesn't use built-in `min()` function
- Compares each value against the other two systematically
- Handles all edge cases correctly

Comprehensive Testing:

- Validates all 3 sample inputs
- Tests 10 additional cases including:
  - Equal values (all same, two equal)
  - Negative and floating-point numbers
  - Wide value ranges
  - Different orderings

How Examples Guided the Function:

1. Showed minimum can be at any position
2. Demonstrated need for systematic comparison logic
3. Guided toward conditional if-elif-else pattern
4. Ensured correctness through concrete pattern examples

All test cases pass successfully!