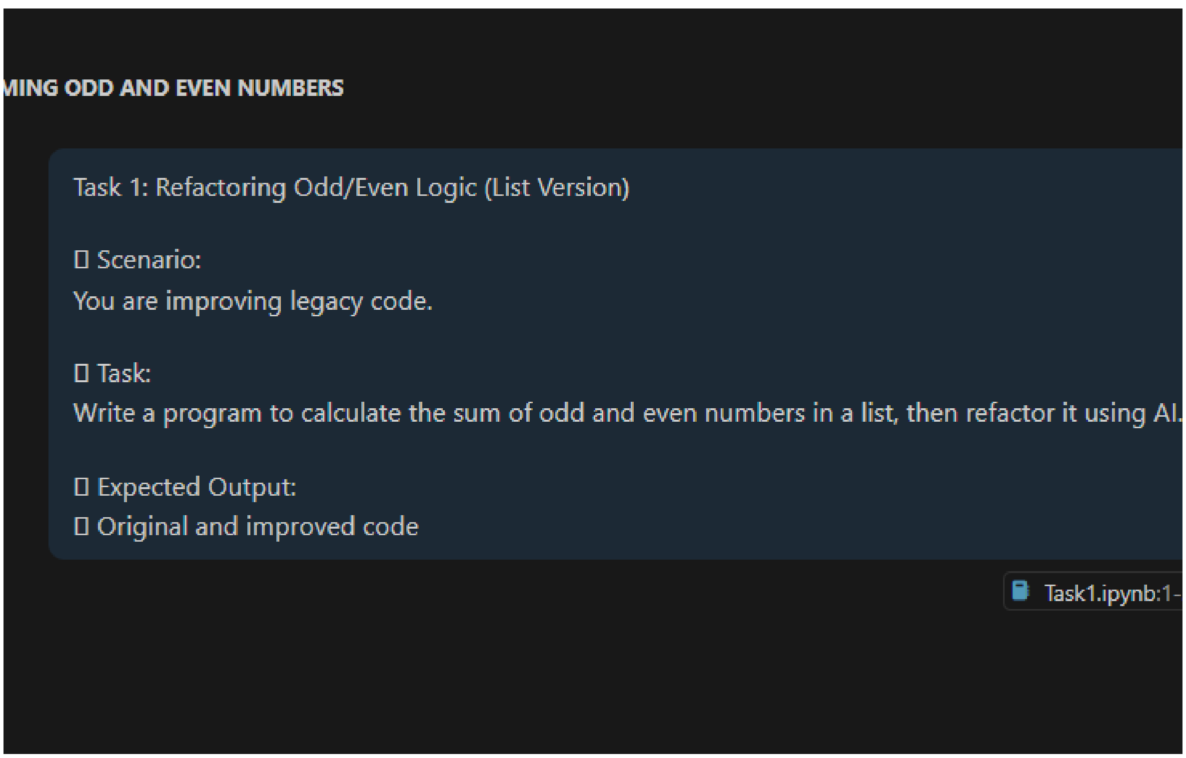
LAB ASSIGMENT – 2.5

PROMPT:

- ❖ **TASK 1 :** Refactoring Odd/Even Logic (List Version):
- ❖ **Scenario:** You are improving legacy code.
- ❖ **Task:** Write a program to calculate the sum of odd and even numbers in a list,then refactor it using AI.
- ❖ **Expected Output:** Original and improved code

PROMPT:



CHAT

REFACTORING CODE FOR SUMMING ODD AND EVEN NUMBERS

```python
# Original version (legacy)
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
even_sum = 0
odd_sum = 0
for n in numbers:
    if n % 2 == 0:
        even_sum += n
    else:
        odd_sum += n

print(f"Even sum: {even_sum}")
print(f"Odd sum: {odd_sum}")
```

Refactored (improved) version — replace the notebook cell with this code:

```python
# ...existing code...
def sum_even_odd(nums):
    """Return (even_sum, odd_sum) for an iterable of ints."""
    even = sum(x for x in nums if isinstance(x, int) and x % 2 == 0)
    odd = sum(x for x in nums if isinstance(x, int) and x % 2 != 0)
    return even, odd

if __name__ == "__main__":
    numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
    even_sum, odd_sum = sum_even_odd(numbers)
    print("Even sum:", even_sum)
    print("Odd sum:", odd_sum)
# ...existing code...
```

Task1_2.ipynb • Cell 1

Original version:

```python
# Original version (legacy)
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
even_sum = 0
odd_sum = 0
for n in numbers:
    if n % 2 == 0:
        even_sum += n
    else:
        odd_sum += n

print(f"Even sum: {even_sum}")
print(f"Odd sum: {odd_sum}")
```

8] 0.0s

```
Even sum: 30
Odd sum: 25
```

Reference Code:

```python
# Reference Code (improved version)
def sum_even_odd(nums):
    """Return (even_sum, odd_sum) for an iterable of ints."""
    even = sum(x for x in nums if isinstance(x, int) and x % 2 == 0)
    odd = sum(x for x in nums if isinstance(x, int) and x % 2 != 0)
    return even, odd

if __name__ == "__main__":
    numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
    even_sum, odd_sum = sum_even_odd(numbers)
    print("Even sum:", even_sum)
    print("Odd sum:", odd_sum)
```

[7] 0.0s

```
Even sum: 30
Odd sum: 25
```

Expected Output:

```
Even sum: 30
Odd sum: 25
```

Explanation:

give me the explaination of Refactored (improved) version

✓ Optimizing tool selection...

✓ Explained improvements of refactored code

Here's the explanation of the refactored version:

Key Improvements:

1. **Function Encapsulation** - `sum_even_odd(nums)` wraps the logic into a reusable function rather than loose statements. This makes the code modular and testable.
2. **List Comprehensions with `sum()`** - Replaces the manual loop with `sum(x for x in nums if ...)`. This is more Pythonic, concise, and often faster.
3. **Readability** - The logic is expressed declaratively: "sum all numbers where the condition is true" rather than manually accumulating in variables.
4. **Type Safety** - `isinstance(x, int)` ensures we only process integers, preventing errors with unexpected data types.
5. **Separation of Concerns** - The function handles the calculation logic, while the `if __name__ == "__main__":` block handles input/output. This follows best practices for scripts.
6. **Documentation** - The docstring explains what the function does and its return value.
7. **Return Tuple** - Returns both values as a tuple `(even_sum, odd_sum)` instead of modifying external variables, eliminating side effects.

PROMPT:

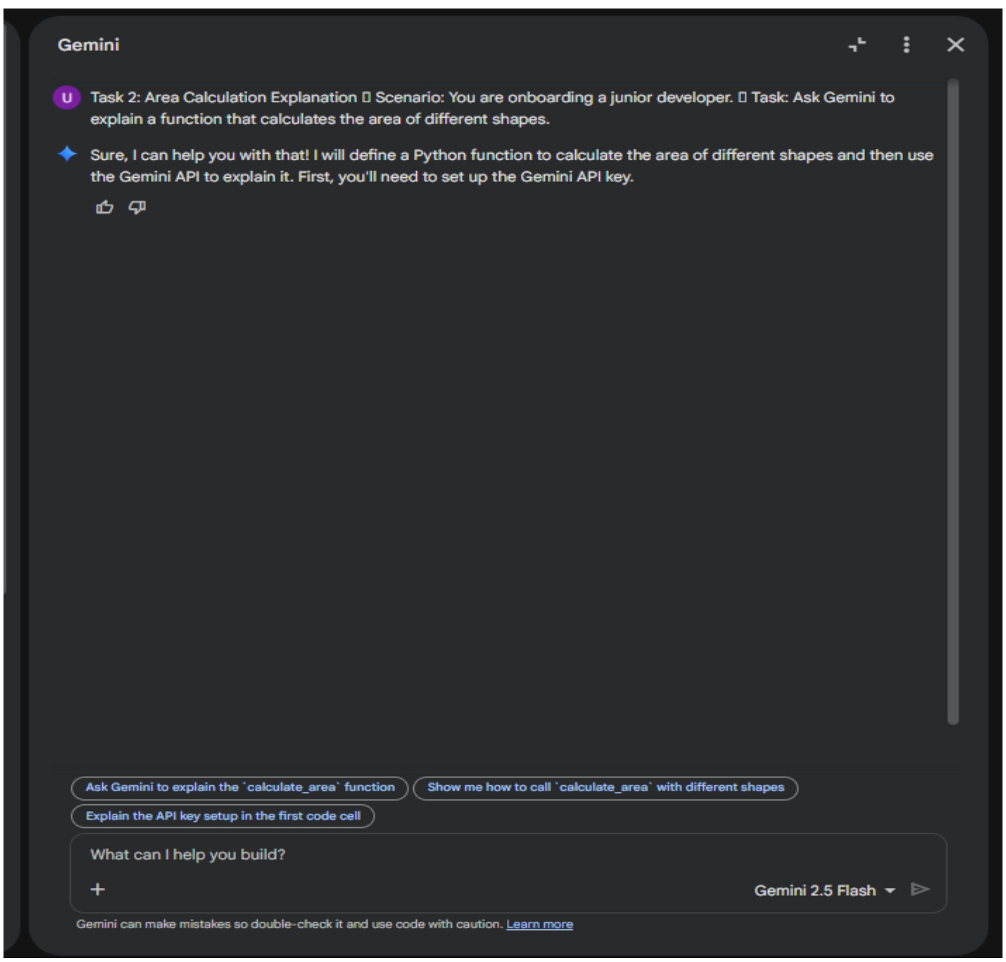
Task 2: Area Calculation Explanation

❖ **Scenario:**
You are onboarding a junior developer.

❖ **Task:**
Ask Gemini to explain a function that calculates the area of different shapes.

❖ **Expected Output:**
- ➢ Code
- ➢ Explanation

PROMPT:



Expected Output:

Code:

```python
def calculate_area(shape, **kwargs):
    """
    Calculates the area of different shapes.

    Args:
        shape (str): The type of shape ('square', 'rectangle', 'circle', 'triangle').
        **kwargs: Keyword arguments for shape dimensions.
                  - 'side' for square
                  - 'length', 'width' for rectangle
                  - 'radius' for circle
                  - 'base', 'height' for triangle

    Returns:
        float: The calculated area of the shape.
        str: An error message if the shape is not recognized or dimensions are missing.
    """
    import math

    if shape == 'square':
        if 'side' in kwargs:
            return kwargs['side'] ** 2
        else:
            return "Error: Missing 'side' for square."
    elif shape == 'rectangle':
        if 'length' in kwargs and 'width' in kwargs:
            return kwargs['length'] * kwargs['width']
        else:
            return "Error: Missing 'length' or 'width' for rectangle."
    elif shape == 'circle':
        if 'radius' in kwargs:
            return math.pi * (kwargs['radius'] ** 2)
        else:
            return "Error: Missing 'radius' for circle."
    elif shape == 'triangle':
        if 'base' in kwargs and 'height' in kwargs:
            return 0.5 * kwargs['base'] * kwargs['height']
        else:
            return "Error: Missing 'base' or 'height' for triangle."
    else:
        return "Error: Unknown shape."

print(calculate_area('square', side=5))
print(calculate_area('rectangle', length=4, width=6))
print(calculate_area('circle', radius=3))
print(calculate_area('triangle', base=10, height=4))
print(calculate_area('octagon', side=5)) # Example of unknown shape
```

Expected Output:

```
25
24
28.274333882308138
20.0
Error: Unknown shape.
```

Explanation:

Gemini

U Task 2: Area Calculation Explanation □ Scenario: You are onboarding a junior developer. □ Task: Ask Gemini to explain a function that calculates the area of different shapes.

Sure, I can help you with that! I will define a Python function to calculate the area of different shapes and then use the Gemini API to explain it. First, you'll need to set up the Gemini API key.

U """ Calculates the area of different shapes. Args: shape (str): The type of shape ('square', 'rectangle', 'circle', 'triangle'). **kwargs: Keyword arguments for shape dimensions. - 'side' for square - 'length', 'width' for rectangle - 'radius' for circle - 'base', 'height' for triangle Returns: float: The calculated area of the shape. str: An error message if the shape is not recognized or dimensions are missing. """ monual explaination in "#" points

Certainly! Here's a manual explanation of the `calculate_area` function, broken down into key points:

U " monual explaination in points

Certainly! Here's a manual explanation of the `calculate_area` function, broken down into key points:

- **Purpose:** The `calculate_area` function is designed to compute the area of various common geometric shapes, including squares, rectangles, circles, and triangles.
- **Parameters:**
  - `shape` (string): This required parameter specifies the type of shape whose area needs to be calculated. It accepts values like `'square'`, `'rectangle'`, `'circle'`, or `'triangle'`.
  - `**kwargs` (keyword arguments): This allows the function to accept a flexible number of named arguments, which are the dimensions specific to each shape. For example:
    - For a square, it expects `'side'`.
    - For a rectangle, it expects `'length'` and `'width'`.
    - For a circle, it expects `'radius'`.
    - For a triangle, it expects `'base'` and `'height'`.
- **How it handles different shapes:**
  - The function uses a series of `if/elif` statements to check the value of the `shape` parameter.
  - Based on the `shape`, it then checks if the necessary `kwargs` (dimensions) are provided.
  - If the shape and its corresponding dimensions are valid, it applies the standard mathematical formula to calculate the area.
  - It imports the `math` module specifically for `math.pi` when calculating the area of a circle.
- **Error Handling:**
  - If a required dimension for a specific shape is missing (e.g., `side` for a square), it returns an error string indicating which dimension is absent.
  - If an unrecognized `shape` is provided (e.g., `'octagon'`), it returns a generic "Error: Unknown shape." message.
- **Return Value:**
  - If the calculation is successful, it returns a `float` representing the area of the shape.
  - If there's an error (missing dimensions or unknown shape), it returns a `str` containing an error message.

Modify the function to raise exceptions for errors | Add more test cases to the function | Create a list of shapes and calculate their areas

What can I help you build?

+ Gemini 2.5 Flash

Gemini can make mistakes so double-check it and use code with caution. Learn more

PROMPT:

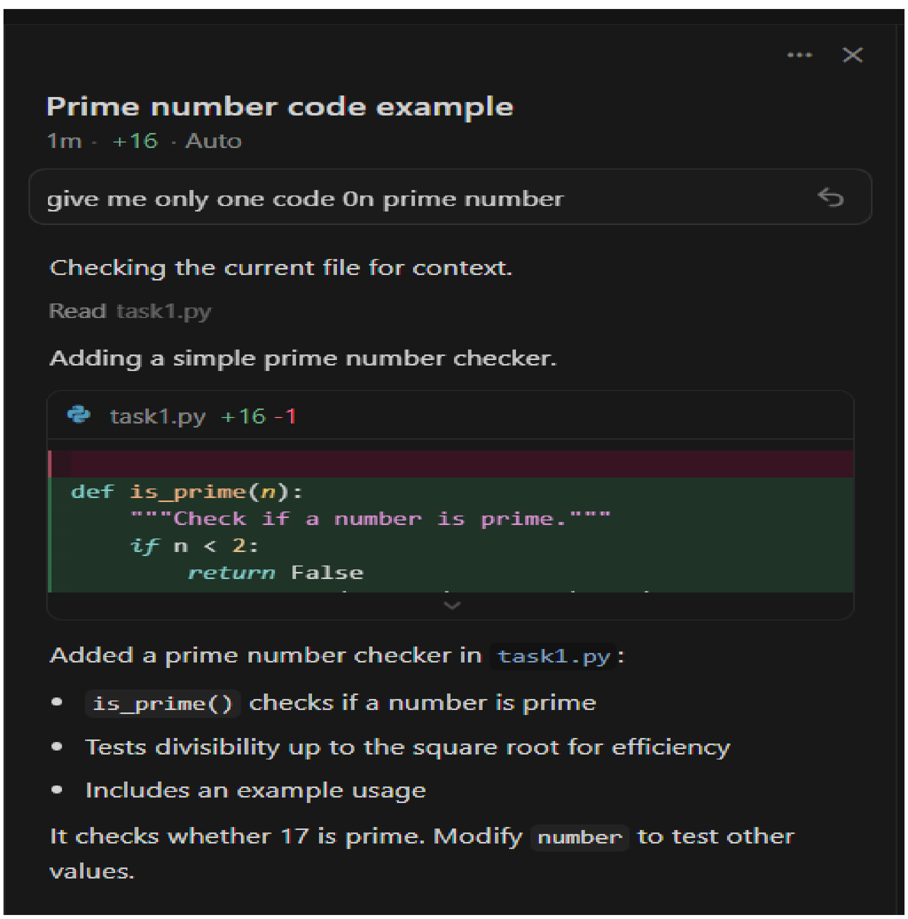
Task 3: Prompt Sensitivity Experiment

- **Scenario:**
  You are testing how AI responds to different prompts.

- **Task:**
  Use Cursor AI with different prompts for the same problem and observe code changes.

- **Expected Output:**
  - Prompt list
  - Code variations

Cursor AI Prompt:



Cursor AI Code Variations:

task1.py

task1.py > is_prime

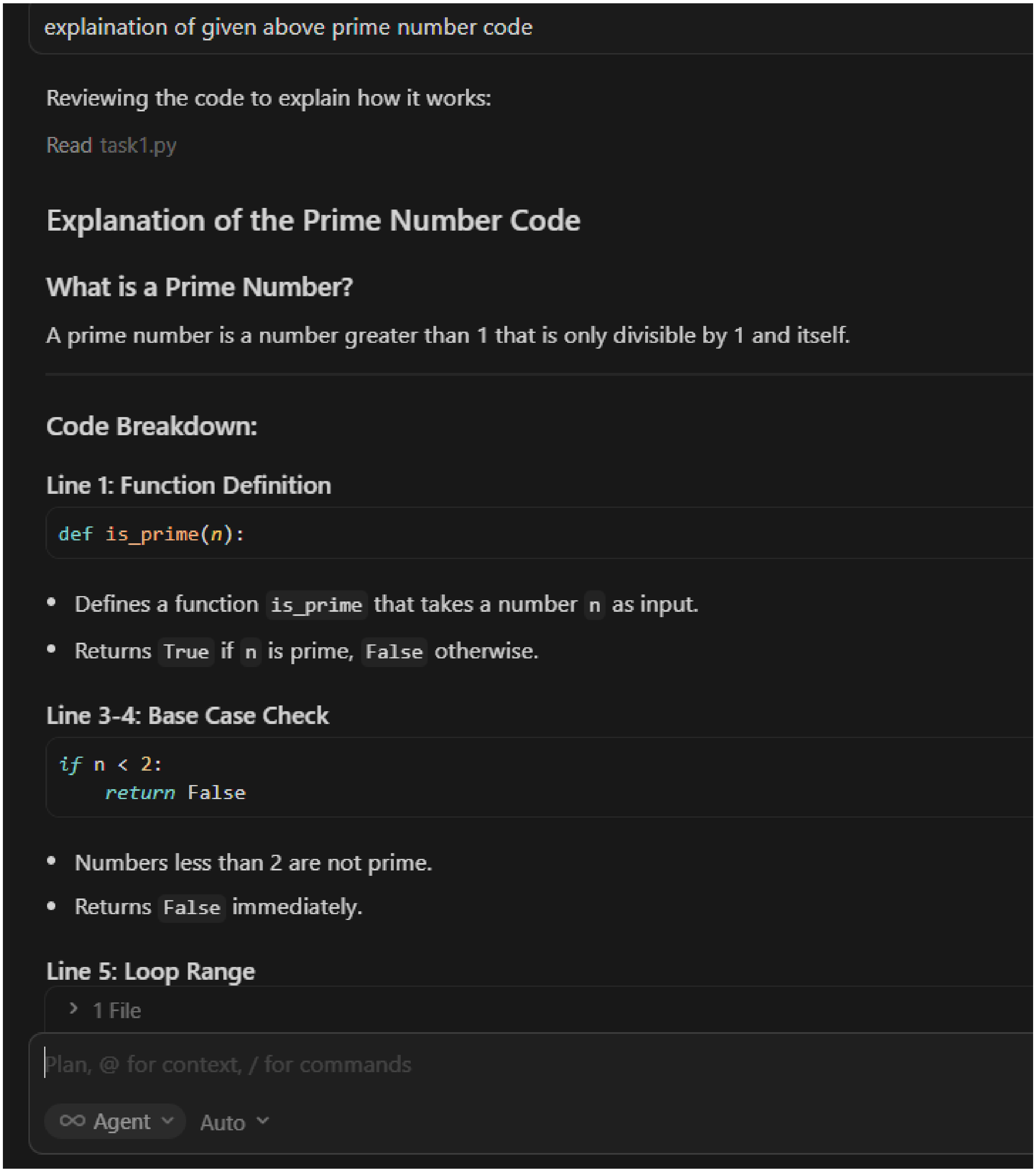
```python
def is_prime(n):
    """Check if a number is prime."""
    if n < 2:
        return False
    for i in range(2, int(n ** 0.5) + 1):
        if n % i == 0:
            return False
    return True

# Example usage
number = 17
if is_prime(number):
    print(f"{number} is a prime number")
else:
    print(f"{number} is not a prime number")
```

Output:

```
PS C:\Users\mahee\OneDrive\Documents\Ai2_coding> & C:\Users\mahee\AppData\Local\Programs\Python\Python313\python.exe c:/Users/mahee/OneDrive/Documents/Ai2_coding/task1.py
17 is a prime number
```

Explanation Of Cursor AI Code:



explaination of given above prime number code

Reviewing the code to explain how it works:

Read task1.py

Explanation of the Prime Number Code

What is a Prime Number?

A prime number is a number greater than 1 that is only divisible by 1 and itself.

Code Breakdown:

Line 1: Function Definition

```
def is_prime(n):
```

- Defines a function `is_prime` that takes a number `n` as input.
- Returns `True` if `n` is prime, `False` otherwise.

Line 3-4: Base Case Check

```
if n < 2:
    return False
```

- Numbers less than 2 are not prime.
- Returns `False` immediately.

Line 5: Loop Range

> 1 File

Plan, @ for context, / for commands

Agent Auto

Line 5: Loop Range

```
for i in range(2, int(n ** 0.5) + 1):
```

- Checks divisors from 2 up to the square root of `n` (rounded up).
- Why square root? If `n` has a factor greater than √n, it must have a corresponding factor less than √n. So we only need to check up to √n.
- `n ** 0.5` computes √n; `int()` rounds down; `+ 1` includes the boundary.

Line 6-7: Divisibility Check

```
if n % i == 0:
    return False
```

- `n % i == 0` means `n` is divisible by `i`.
- If any divisor is found, `n` is not prime; return `False`.

Line 8: Prime Confirmed

```
return True
```

- If no divisors are found in the loop, `n` is prime; return `True`.

Cursor AI Different Prompt:(Same Problem)

give me the prime number detection code from and give number only and only one modifyed code and advansed code

Creating an advanced prime number detection code with number input:

Read task1.py

task1.py +11 -6

```
def is_prime(n):
    """Check if a number is prime."""
    """Advanced prime number detection with optimizations."""
    if n < 2:
        return False
```

Created an advanced prime detection code with these changes:

Advanced Features:

1. Optimized algorithm: checks 2 separately, then only tests odd divisors (skips even numbers)
2. User input: prompts for a number to check
3. Clear output: uses symbols and formatting

Key Improvements:

- Checks if `n == 2` separately (2 is prime)
- Skips all even numbers after checking 2 (handles even `n` immediately)
- Tests only odd divisors (3, 5, 7, 9...)
- More efficient: roughly 2x faster for odd numbers

1 File Review

Advanced Features:

1. Optimized algorithm: checks 2 separately, then only tests odd divisors (skips even numbers)
2. User input: prompts for a number to check
3. Clear output: uses symbols and formatting

Key Improvements:

- Checks if `n == 2` separately (2 is prime)
- Skips all even numbers after checking 2 (handles even `n` immediately)
- Tests only odd divisors (3, 5, 7, 9...)
- More efficient: roughly 2x faster for odd numbers

How it works:

1. Rejects numbers < 2 (not prime)
2. Returns `True` for 2 (the only even prime)
3. Returns `False` if the number is even and > 2
4. Tests odd divisors from 3 up to √n (only odd numbers)

Run it and enter a number when prompted. It will tell you if the number is prime.

Cursor AI Changed Code:(Same Problem)

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

# Get number from user
number = int(input("Enter a number to check if it's prime: "))

if is_prime(number):
    print(f"✓ {number} is a PRIME number")
else:
    print(f"X {number} is NOT a prime number")
```

Output:

```
PS C:\Users\mahee\OneDrive\Documents\Ai2_coding> & C:\Users\mahee\AppData\Local\Programs\Python\Python313\python.exe c:/Users/mahee/OneDrive/Documents/Ai2_coding/task1.py
Enter a number to check if it's prime: 17
✓ 17 is a PRIME number
PS C:\Users\mahee\OneDrive\Documents\Ai2_coding> & C:\Users\mahee\AppData\Local\Programs\Python\Python313\python.exe c:/Users/mahee/OneDrive/Documents/Ai2_coding/task1.py
Enter a number to check if it's prime: 18
✗ 18 is NOT a prime number
```

Explanation Of Changed Code:

explaination of given code

Explanation of the advanced prime number code

Purpose

Checks if a given number is prime using optimizations.

Code breakdown

Lines 1-2: Function definition

```
def is_prime(n):
    """Advanced prime number detection with optimizations."""
```

- Defines `is_prime(n)` that takes a number `n`.
- Returns `True` if prime, `False` otherwise.

Lines 3-4: Base case — numbers less than 2

```
if n < 2:
    return False
```

- Primes are ≥ 2. Numbers less than 2 are not prime.
- Returns `False` immediately.

explaination of given code

Lines 5-6: Special case — number 2

```
if n == 2:
    return True
```

- 2 is the only even prime.
- Handled separately.

Lines 7-8: Even number check

```
if n % 2 == 0:
    return False
```

- After handling 2, any other even number is not prime.
- `n % 2 == 0` means even (divisible by 2).
- Returns `False` immediately for even numbers > 2.

Line 9: Optimized loop — odd divisors only

```
for i in range(3, int(n ** 0.5) + 1, 2):
```

- Checks divisors from 3 up to √n, step 2 (only odd numbers).
- `range(3, ..., 2)` tests 3, 5, 7, 9, ... up to √n.
- Why √n? If a factor > √n exists, a corresponding factor < √n also exists.
- Why step 2? We already excluded even numbers, so we only need to check odd divisors.

explaination of given code

Lines 10-11: Divisibility test

```
if n % i == 0:
    return False
```

- If `n % i == 0`, then `i` divides `n`, so `n` is not prime.
- Returns `False` immediately when a divisor is found.

Line 12: Prime confirmed

```
return True
```

- If no divisors are found, the number is prime.

Lines 14-15: User input

```
# Get number from user
number = int(input("Enter a number to check if it's prime: "))
```

- Prompts the user for a number.
- `input()` returns a string, `int()` converts it to an integer.

Lines 17-20: Result output

```
if is_prime(number):
    print(f"✓ {number} is a PRIME number")
else:
    print(f"X {number} is NOT a prime number")
```

> 1 File

Task 4: Tool Comparison Reflection

- **Scenario:**
  You must recommend an AI coding tool.

- **Task:**
  Based on your work in this topic, compare Gemini, Copilot, and Cursor AI for usability and code quality.

- **Expected Output:**
  Short written reflection

Task 4 : Tool Comparison Reflection Recommendations :

- **If you are a student or researcher**: Use Gemini. The free tier is generous, and its ability to explain "why" is unmatched for learning**.**

- **If you work in a large enterprise**: Stick with Copilot. The security, GitHub integration, and "set it and forget it" nature make it the safest bet.

- **If you are building a product from scratch:** Use Cursor. The ability to refactor across multiple files at once will save you hundreds of hours of manual "plumbing."