

# AI ASSISTED CODING

Hall Ticket No: 2303A510B4

Batch:14

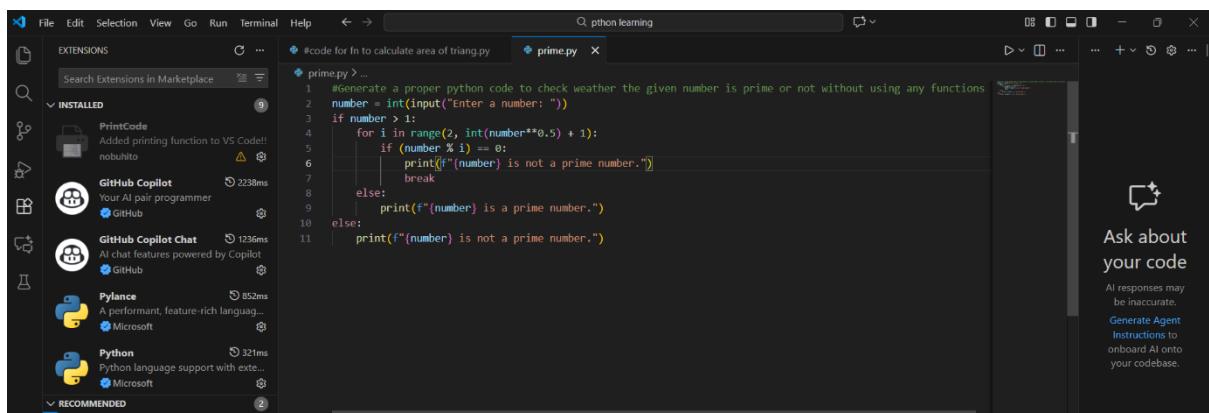
## Assignment-1.4

### Task-1. AI-Generated Logic Without Modularization (Prime Number Check Without Functions)

#### Prompt

#Generate a proper python code to check weather the given number is prime or not without using any functions

#### Code

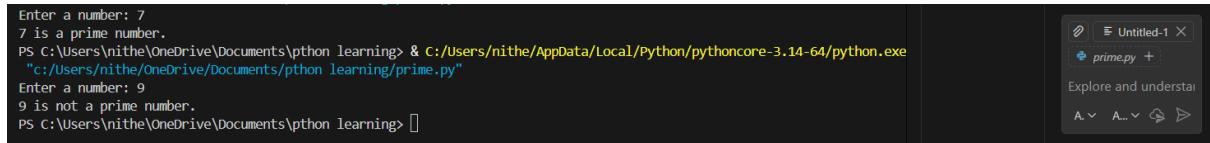


The screenshot shows the Visual Studio Code interface with the following details:

- File Bar:** File, Edit, Selection, View, Go, Run, Terminal, Help.
- Search Bar:** Search Extensions in Marketplace.
- Extensions Sidebar:** Shows installed extensions: PrintCode, GitHub Copilot, GitHub Copilot Chat, PyLance, and Python.
- Code Editor:** A Python file named "prime.py" containing the following code:

```
#code for In to calculate area of triang.py
#generate a proper python code to check weather the given number is prime or not without using any functions
prime.py > ...
1 number = int(input("Enter a number: "))
2 if number > 1:
3     for i in range(2, int(number**0.5) + 1):
4         if (number % i) == 0:
5             print(f"{number} is not a prime number.")
6             break
7     else:
8         print(f"{number} is a prime number.")
9
10 else:
11     print(f"{number} is not a prime number.")
```
- Right Panel:** Shows a "Ask about your code" button and a note: "AI responses may be inaccurate. Generate Agent Instructions to onboard AI onto your codebase."

# Output:



```
Enter a number: 7
7 is a prime number.
PS C:\Users\nithe\OneDrive\Documents\python learning> & C:/Users/nithe/AppData/Local/Python/pythoncore-3.14-64/python.exe
"c:/Users/nithe/OneDrive/Documents/python learning/prime.py"
Enter a number: 9
9 is not a prime number.
PS C:\Users\nithe\OneDrive\Documents\python learning>
```

## Justification:

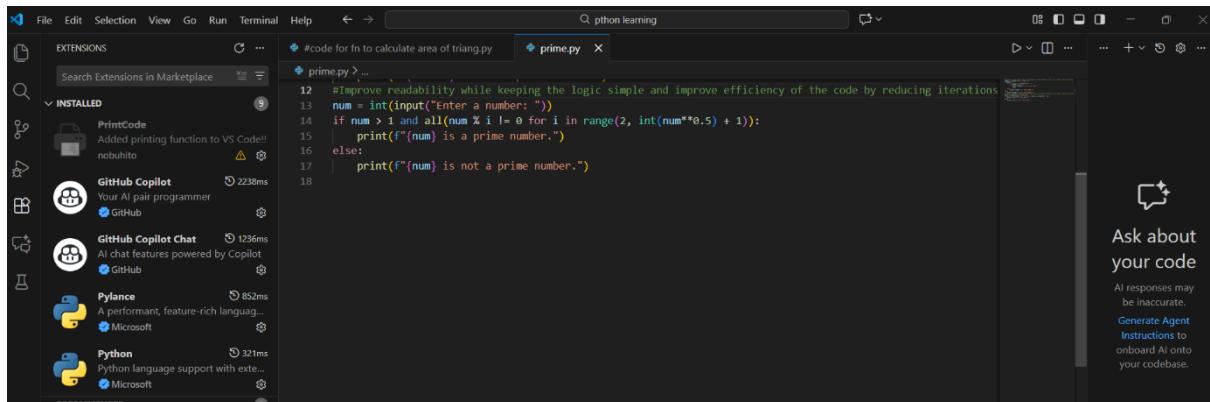
This program checks whether a given number is prime using direct conditional logic without defining any functions.  
All computations are performed sequentially in a single block, making the logic easy to follow and suitable for beginners.

## Task-2. Efficiency & Logic Optimization (Cleanup)

### Prompt

#Improve readability while keeping the logic simple and improve efficiency of the code by reducing iterations also minimize the code length

### Code:



```
12 #Improve readability while keeping the logic simple and improve efficiency of the code by reducing iterations
13 num = int(input("Enter a number: "))
14 if num > 1 and all(num % i != 0 for i in range(2, int(num**0.5) + 1)):
15     print(f"{num} is a prime number.")
16 else:
17     print(f"{num} is not a prime number.")
```

## Output:

```
Enter a number: 579
579 is not a prime number.
Enter a number: 1236
1236 is not a prime number.
PS C:\Users\nithe\OneDrive\Documents\python learning>
```

## Justification:

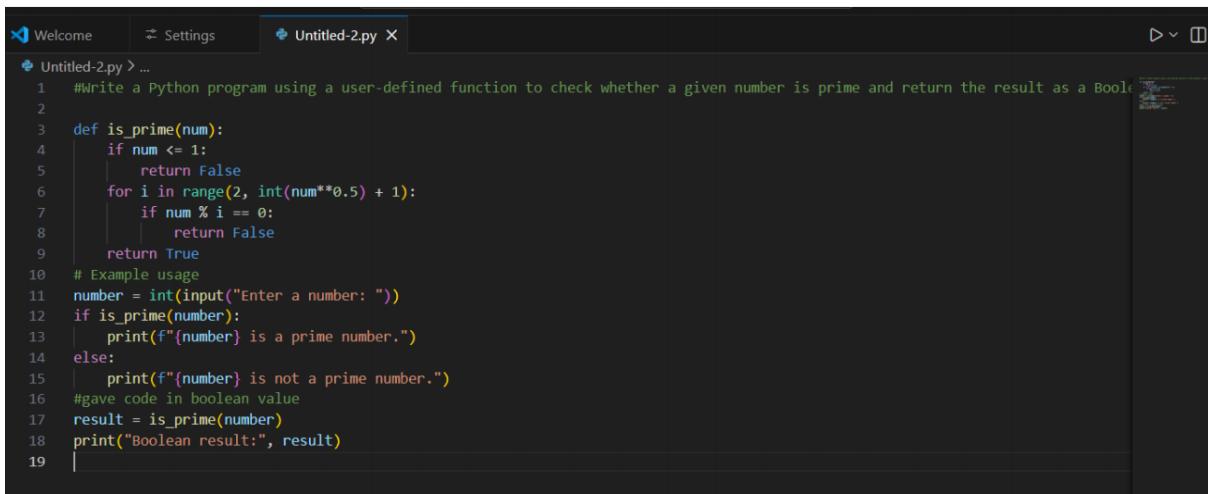
The optimized script improves performance by reducing unnecessary iterations and limiting the loop range, enabling faster execution for larger input values.

Early termination and simplified conditions lower the overall time complexity while maintaining correct prime number validation.

## Task-3. Modular Design Using AI Assistance (Prime Number Check Using Functions)

### Prompt:

#The function must return a Boolean value (True if prime, False otherwise)



A screenshot of a code editor window titled "Untitled-2.py". The code is a Python script for checking if a number is prime. It includes a function named "is\_prime" that takes a number as input and returns True if it's prime, and False otherwise. The script also includes example usage where it prompts the user for a number, checks it using the function, and prints the result. The code is color-coded with syntax highlighting.

```
1  #write a Python program using a user-defined function to check whether a given number is prime and return the result as a Boolean
2
3 def is_prime(num):
4     if num <= 1:
5         return False
6     for i in range(2, int(num**0.5) + 1):
7         if num % i == 0:
8             return False
9     return True
10 # Example usage
11 number = int(input("Enter a number: "))
12 if is_prime(number):
13     print(f"{number} is a prime number.")
14 else:
15     print(f"{number} is not a prime number.")
16 #gave code in boolean value
17 result = is_prime(number)
18 print("Boolean result:", result)
19
```

## Output:

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS Python + × └ ━ ⌂ PS C:\Users\meteb\OneDrive\Desktop\python> & C:/Users/meteb/AppData/Local/Microsoft/WindowsApps/python3.13.exe c:/Users/meteb/OneDrive/Desktop/python/Urled-2.py Enter a number: 571 571 is a prime number. Boolean result: True PS C:\Users\meteb\OneDrive\Desktop\python> & C:/Users/meteb/AppData/Local/Microsoft/WindowsApps/python3.13.exe c:/Users/meteb/OneDrive/Desktop/python/Urled-2.py Enter a number: 588
```

## **Justification:**

**Using a user-defined function makes the prime-checking logic reusable across multiple modules, improving code modularity and maintainability. Returning a Boolean value enables easy integration with conditional statements and other program components.**

## **Task-4: Comparative Analysis –With vs Without Functions**

# Prompt:

```
# Compare both code with function without function Analyze and  
# compare two Python programs for checking whether a number is  
# prime
```

## Code:

```
◆ Untitled-2.py > ...
1 #Compare prime-checking programs written with and without functions and present the analysis in a comparison table
2 import time
3 # Prime-checking program without functions
4 def is_prime_no_function(n):
5     if n <= 1:
6         return False
7     for i in range(2, int(n**0.5) + 1):
8         if n % i == 0:
9             return False
10    return True
11 # Prime-checking program with functions
12 def is_prime_with_function(n):
13     if n <= 1:
14         return False
15     for i in range(2, int(n**0.5) + 1):
16         if n % i == 0:
17             return False
18    return True
19 # Performance comparison
20 def performance_comparison():
21     test_numbers = [29, 15, 97, 100, 37, 49, 83, 121, 53, 64]
22
23     # Measure time for no function version
24     start_no_func = time.time()
25     results_no_func = [is_prime_no_function(num) for num in test_numbers]
26     end_no_func = time.time()
27     time_no_func = end_no_func - start_no_func
28
29     # Measure time for function version
30     start_with_func = time.time()
```

## Output:

The screenshot shows a terminal window with the following content:

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
PS C:\Users\meteb\OneDrive\Desktop> python & C:/Users/meteb/AppData/Local/Microsoft/WindowsApps/python3.13.exe c:/Users/meteb/OneDrive/Desktop/python/Untitled-2.py
PS C:\Users\meteb\OneDrive\Desktop> python & C:/Users/meteb/AppData/Local/Microsoft/WindowsApps/python3.13.exe c:/Users/meteb/OneDrive/Desktop/python/Untitled-2.py
Implementation      Time Taken (seconds)    Results
Without Functions   0.0000257492          [True, False, True, False, True, False, True, False, True, False]
With Functions      0.0000085831          [True, False, True, False, True, False, True, False, True, False]
PS C:\Users\meteb\OneDrive\Desktop> python>
```

## Justification:

Programs written with functions offer better code clarity by separating logic into well-defined blocks, making them easier to read and understand. Function-based designs improve reusability and debugging ease, as changes or fixes can be applied in one place without affecting the entire code.

## Task-5: AI-Generated Iterative vs Recursive Fibonacci Approaches (Different Algorithmic Approaches to Prime Checking)

**Prompt: Prime Number Check – Basic vs Optimized Approach**

**Code:**

```
#A basic divisibility check approach that tests all possible divisors sequentially
# Implementation 2: Optimized approach
def is_prime_optimized(n):
    """Check if a number is prime using an optimized approach."""
    if n <= 1:
        return False
    if n <= 3:
        return True
    if n % 2 == 0 or n % 3 == 0:
        return False
    i = 5
    while i * i <= n:
        if n % i == 0 or n % (i + 2) == 0:
            return False
        i += 6
    return True

#Prime Number Check - Basic vs Optimized Approach
#An optimized method that reduces the number of checks by eliminating even numbers and testing up to the square root of n
# Example usage
if __name__ == "__main__":
    test_numbers = [1, 2, 3, 4, 5, 16, 17, 18, 19, 20]
    for number in test_numbers:
        print(f"Basic: Is {number} prime? {is_prime_basic(number)}")
        print(f"Optimized: Is {number} prime? {is_prime_optimized(number)}")
```

## Output:

```
Basic: Is 1 prime? False
Optimized: Is 1 prime? False
Basic: Is 2 prime? True
Optimized: Is 2 prime? True
Basic: Is 3 prime? True
Optimized: Is 3 prime? True
Basic: Is 4 prime? False
Optimized: Is 4 prime? False
Basic: Is 5 prime? True
Optimized: Is 5 prime? True
Basic: Is 16 prime? False
Optimized: Is 5 prime? True
Basic: Is 16 prime? False
Basic: Is 16 prime? False
Optimized: Is 16 prime? False
Basic: Is 17 prime? True
Optimized: Is 17 prime? True
Basic: Is 18 prime? False
Optimized: Is 18 prime? False
Basic: Is 17 prime? True
Optimized: Is 17 prime? True
Basic: Is 18 prime? False
Optimized: Is 18 prime? False
Basic: Is 18 prime? False
Optimized: Is 18 prime? False
Basic: Is 19 prime? True
Optimized: Is 19 prime? True
```

## Justification:

**The basic approach checks divisibility up to N-1, resulting in unnecessary iterations and higher time complexity.**

**The optimized approach checks only up to  $\sqrt{N}$  because any factor larger than  $\sqrt{N}$  must have a corresponding smaller factor.**