

AI

ASSISTED CODING

Hall Ticket No: 2303A510G4

Batch:14

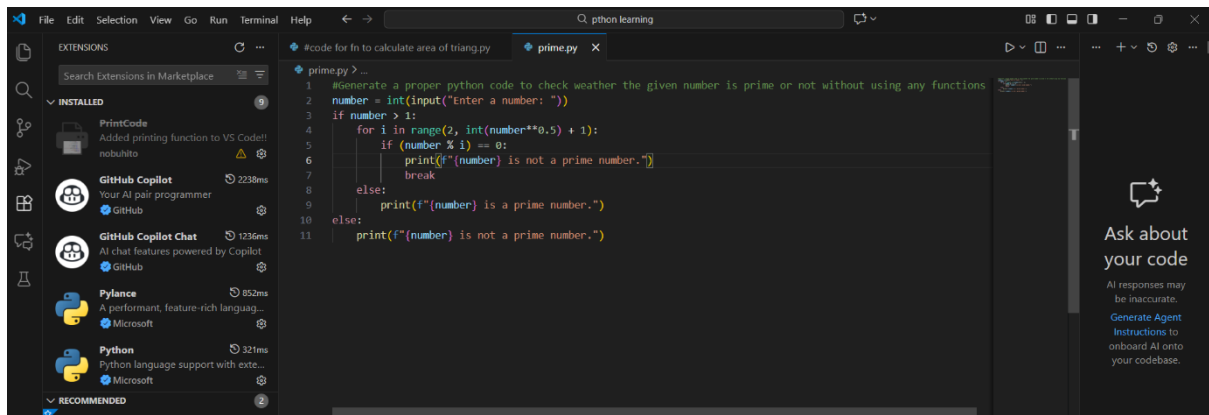
Assignment-1.4

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

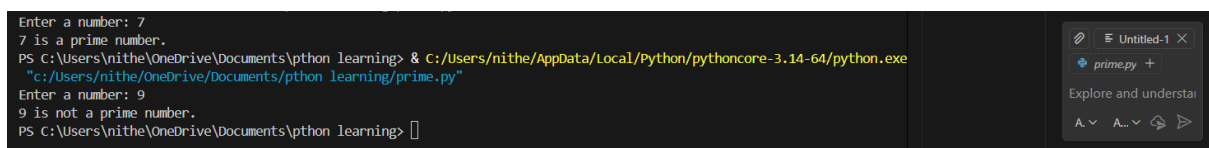
Prompt

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

Code



Output:



Justification:

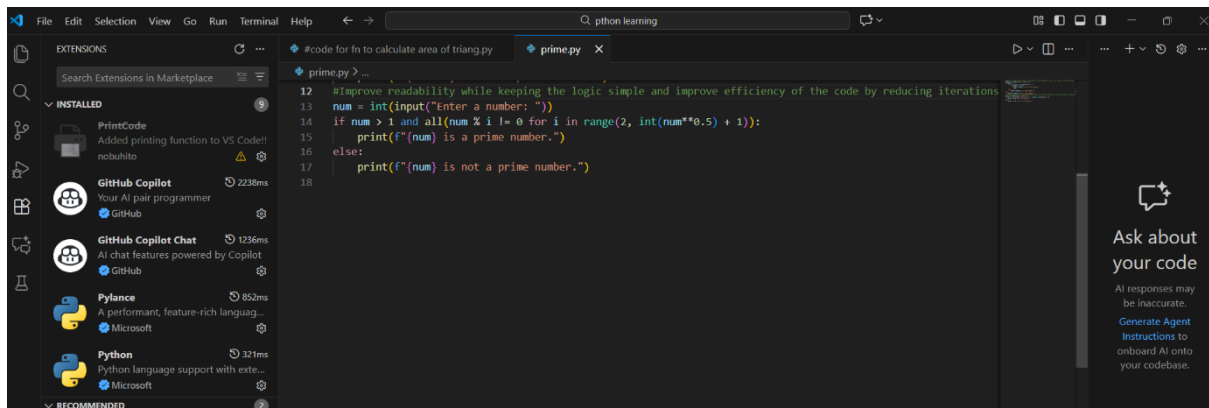
This program checks whether a given number is prime using direct conditional logic without defining any functions. All computations are performed sequentially in a single block, making the logic easy to follow and suitable for beginners.

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:



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\pthon 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)

```
Welcome  Settings  Untitled-2.py X
+ Untitled-2.py > ...
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 + v [icon] [icon] [icon]
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
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/Untitled-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:

```
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:

```
#code for fn to calculate area of triang.py  prime.py X
prime.py > ...
11  #A basic divisibility check approach that tests all possible divisors sequentially
12  # Implementation 2: Optimized approach
13  def is_prime_optimized(n):
14      """Check if a number is prime using an optimized approach."""
15      if n <= 1:
16          return False
17      if n <= 3:
18          return True
19      if n % 2 == 0 or n % 3 == 0:
20          return False
21      i = 5
22      while i * i <= n:
23          if n % i == 0 or n % (i + 2) == 0:
24              return False
25          i += 6
26      return True
27  #Prime Number Check - Basic vs Optimized Approach
28  #An optimized method that reduces the number of checks by eliminating even numbers and testing up to the square root
29  # Example usage
30  if __name__ == "__main__":
31      test_numbers = [1, 2, 3, 4, 5, 16, 17, 18, 19, 20]
32      for number in test_numbers:
33          print(f"Basic: Is {number} prime? {is_prime_basic(number)}")
34          print(f"Optimized: Is {number} prime? {is_prime_optimized(number)}")
35
36
37
```

Output:

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS
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
Optimized: 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.