

AI ASSISTED CODING

Hall Ticket No: 2303A51851

Batch:13

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

A screenshot of the Visual Studio Code interface. The top menu bar includes File, Edit, Selection, View, Go, Run, Terminal, Help, and a search bar for 'python learning'. The left sidebar shows the 'EXTENSIONS' view with several installed extensions: PrintCode, GitHub Copilot, GitHub Copilot Chat, Pylance, and Python. The main code editor window displays a Python script named 'prime.py' with the following code:

```
#generate a proper python code to check whether the given number is prime or not without using any functions
number = int(input("Enter a number: "))
if number > 1:
    for i in range(2, int(number**0.5) + 1):
        if (number % i) == 0:
            print(f"{number} is not a prime number.")
            break
    else:
        print(f"{number} is a prime number.")
else:
    print(f"{number} is not a prime number.")
```

The status bar at the bottom right indicates 'Ask about your code' with a message: 'AI responses may be inaccurate. Generate Agent Instructions to onboard AI onto your codebase.'

Output:

A screenshot of the terminal window in VS Code. The output shows the user entering '9' and the program responding that '9 is NOT a prime number'.

```
Enter a number to check if it is prime: 9
9 is NOT a prime number
=====
TEST CASES (TASK 1):
```

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:

```
#code for fn to calculate area of triang.py
prime.py > ...
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.")
18
```

Ask about your code
AI responses may be inaccurate.
Generate Agent Instructions to onboard AI onto your codebase.

Output:

```
Enter a number to check if it is prime: 234
X 234 is NOT a prime number
=====
```

TEST CASES (TASK 2):

Task2_Modularized.py +
Describe what to build next
Agent Auto ⚡

Justification:

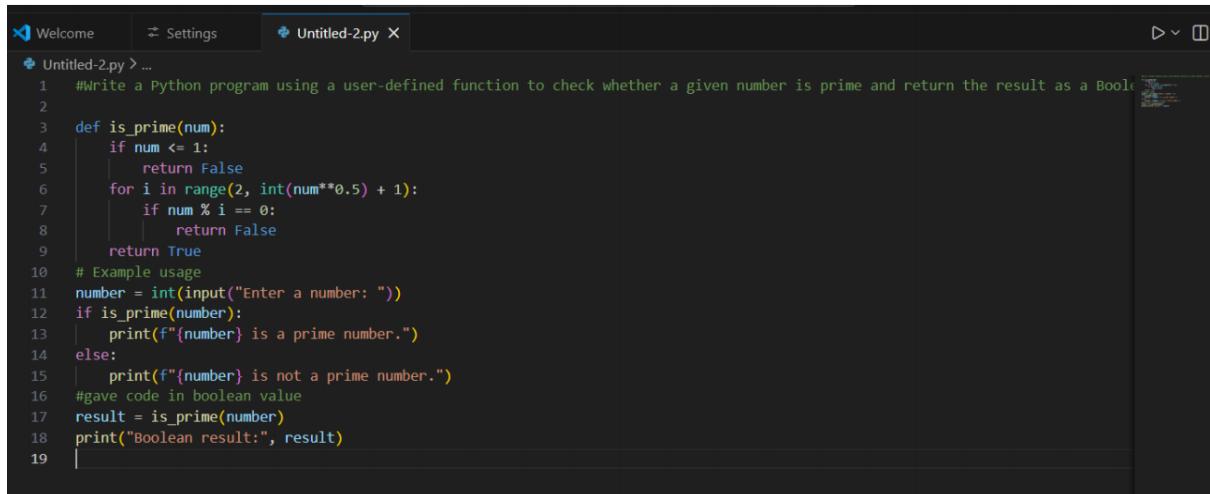
The optimized script improves performance by reducing unnecessary iterations and limiting the loop range, enabling faster execution for larger input values.

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:

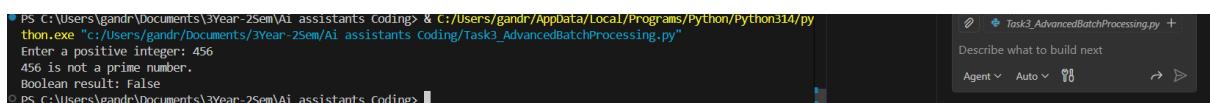
#The function must return a Boolean value (True if prime, False otherwise)



A screenshot of a code editor window titled "Untitled-2.py". The code defines a function "is_prime" that checks if a given number is prime. It includes a docstring, a check for numbers less than or equal to 1, a loop from 2 to the square root of the number, and a final check for divisibility by 2. An example usage is shown where the user inputs 456, the program prints it's not prime, and returns False. The code editor has tabs for "Untitled-2.py" and "Untitled-2.py ~", and a status bar at the bottom.

```
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 value.
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)
```

Output:



A screenshot of a terminal window titled "Task3_AdvancedBatchProcessing.py". It shows the command "python.exe c:/Users/gandr/Documents/3Year-2Sem/AI assistants Coding/Task3_AdvancedBatchProcessing.py" being run. The output indicates that 456 is not a prime number and the Boolean result is False. The terminal also shows the path "C:\Users\gandr\Documents\3Year-2Sem\AI assistants Coding".

```
PS C:\Users\gandr\Documents\3Year-2Sem\AI assistants Coding> & c:/Users/gandr/AppData/Local/Programs/Python/Python314/python.exe "c:/Users/gandr/Documents/3Year-2Sem/AI assistants Coding/Task3_AdvancedBatchProcessing.py"
Enter a positive integer: 456
456 is not a prime number.
Boolean result: False
PS C:\Users\gandr\Documents\3Year-2Sem\AI assistants Coding>
```

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:

```

❸ 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:

```

PS C:\Users\meteb\OneDrive\Desktop\python> & C:/Users/meteb/AppData/Local/Programs/Python/3.11/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 | ...
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
28 #Prime Number Check - Basic vs Optimized Approach
29 #An optimized method that reduces the number of checks by eliminating even numbers and testing up to the square root of the number
30 # Example usage
31 if __name__ == "__main__":
32     test_numbers = [1, 2, 3, 4, 5, 16, 17, 18, 19, 20]
33     for number in test_numbers:
34         print(f"basic: Is {number} prime? {is_prime_basic(number)}")
35
36
37
```

Output:

Number	Basic Method	Optimized Method
1	Is 1 prime? False	Is 1 prime? False
2	Is 2 prime? True	Is 2 prime? True
3	Is 3 prime? True	Is 3 prime? True
4	Is 4 prime? False	Is 4 prime? False
5	Is 5 prime? True	Is 5 prime? True
16	Is 16 prime? False	Is 16 prime? False
17	Is 17 prime? True	Is 17 prime? True
18	Is 18 prime? False	Is 18 prime? False
19	Is 19 prime? True	Is 19 prime? True
20	Is 20 prime? False	Is 20 prime? False

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.