

Name: Karabi Mandal **Assignment:** 1.4 **Batch:** 25

Hallticket Number: 2303a51620

SCHOOL OF COMPUTER SCIENCE AND ARTIFICIAL INTELLIGENCE		DEPARTMENT OF COMPUTER SCIENCE ENGINEERING	
Program Name:	B. Tech	Assignment Type:	Lab
Course Coordinator Name	Dr. Rishabh Mittal		
Instructor(s) Name	<p>Mr. S Naresh Kumar Ms. B. Swathi Dr. Sasanko Shekhar Gantayat Mr. Md Sallauddin Dr. Mathivanan Mr. Y Srikanth Ms. N Shilpa Dr. Rishabh Mittal (Coordinator) Dr. R. Prashant Kumar Mr. Ankushavali MD Mr. B Viswanath Ms. Sujitha Reddy Ms. A. Anitha Ms. M.Madhuri Ms. Katherashala Swetha Ms. Velpula sumalatha Mr. Bingi Raju</p>		
CourseCode	23CS002PC304	Course Title	AI Assisted Coding
Year/Sem	III/II	Regulation	R23
Date and Day of Assignment	Week1 – Thursday	Time(s)	23CSBTB01 To 23CSBTB52
Duration	2 Hours	Applicable to Batches	All batches
Assignment Number: 1.3(Present assignment number)/ 24 (Total number of assignments)			

Q.No.	Question	Expected Time to complete
1	Lab 1: Environment Setup – <i>Github Copilot and VS Code Integration + Understanding AI-assisted Coding Workflow</i> Lab Objectives:	Week1 - Monday

	<ul style="list-style-type: none"> • To install and configure GitHub Copilot in Visual Studio Code. • To explore AI-assisted code generation using GitHub Copilot. • To analyze the accuracy and effectiveness of Copilot's code suggestions. • To understand prompt-based programming using comments and code context <p>Lab Outcomes (LOs): After completing this lab, students will be able to:</p> <ul style="list-style-type: none"> • Set up GitHub Copilot in VS Code successfully. • Use inline comments and context to generate code with Copilot. • Evaluate AI-generated code for correctness and readability. • Compare code suggestions based on different prompts and programming styles. 	
	<p>Task 0</p> <ul style="list-style-type: none"> • Install and configure GitHub Copilot in VS Code. Take screenshots of each step. <p>Expected Output</p> <ul style="list-style-type: none"> • Install and configure GitHub Copilot in VS Code. Take screenshots of each step. 	
	<p>Task 1: AI-Generated Logic Without Modularization (Prime Number Check Without Functions)</p> <p>❖ Scenario</p> <ul style="list-style-type: none"> ➢ You are developing a basic validation script for a numerical learning application. <p>❖ Task Description</p> <p>Use GitHub Copilot to generate a Python program that:</p> <ul style="list-style-type: none"> ➢ Checks whether a given number is prime ➢ Accepts user input ➢ Implements logic directly in the main code ➢ Does not use any user-defined functions <p>❖ Expected Output</p> <ul style="list-style-type: none"> ➢ Correct prime / non-prime result ➢ Screenshots showing Copilot-generated code suggestions ➢ Sample inputs and outputs 	

```
simple.py U    num = 17 Untitled-1 U  practise.yaml U
1 num = 17
2 is_prime = True
3
4 if num < 2:
5     is_prime = False
6 else:
7     for i in range(2, int(num ** 0.5) + 1):
8         if num % i == 0:
9             is_prime = False
10            break
11
12 if is_prime:
13     print(f"{num} is prime")
14 else:
15     print(f"{num} is not prime")
16
17
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

PS C:\Users\GOLI AKASH\OneDrive\Desktop\python\DEVOPS> python -u "c:\Users\GOLI AKASH\OneDrive\Desktop\python\tempCodeRunnerFile.python"
17 is prime

Prompt:- Write a simple Python program to check whether a given number is a prime number **without using functions**. Use basic conditional statements and loops, assume the number is prime initially, and change the result only if a divisor is found. Keep the logic clear, beginner-friendly, and easy to understand.

Explanation:- This code checks whether a number (num = 17) is a prime number by first assuming it is prime using the variable is_prime = True. If the number is less than 2, it is immediately marked as not prime. Otherwise, the code checks whether the number is divisible by any value from 2 up to the square root of the number; if any divisor is found, is_prime is set to False and the loop stops. Finally, based on the value of is_prime, the program prints whether the number is prime or not.

Task 2: Efficiency & Logic Optimization (Cleanup)

❖ Scenario

The script must handle larger input values efficiently.

❖ Task Description

Review the Copilot-generated code from Task 1 and improve it by:

- Reducing unnecessary iterations
- Optimizing the loop range (e.g., early termination)
- Improving readability
- Use Copilot prompts like:
 - *"Optimize prime number checking logic"*
 - *"Improve efficiency of this code"*

Hint:

Prompt Copilot with phrases like

"optimize this code", *"simplify logic"*, or *"make it more readable"*

❖ Expected Output

- Original and optimized code versions
- Explanation of how the improvements reduce time complexity

The screenshot shows a code editor interface with several files listed in the Explorer sidebar under a 'DEVOPS' folder. The files include 'simple.py', 'tempCodeRunnerFile.python', 'practise.yaml', and 'practise.yml'. The 'simple.py' file contains the following Python code:

```

1 def is_prime(n):
2     if n < 2:
3         return False
4     for i in range(2, n): # Checks all numbers up to n
5         if n % i == 0:
6             return False
7     return True
8
9 # Test with dynamic input
10 num = int(input("Enter a number: "))
11 print(f"{num} is prime: {is_prime(num)}")

```

Below the code editor is a terminal window showing the execution of the code. The terminal output is:

```

PS C:\Users\GOLI AKASH\OneDrive\Desktop\python\DEVOPS> python -u "c:\Users\GOLI AKASH\OneDrive\Desktop\python\tempCodeRunnerFile.python"
Enter a number: 5
5 is prime: True
PS C:\Users\GOLI AKASH\OneDrive\Desktop\python\DEVOPS>

```

Prompt:- Optimize this prime number checking code to make it faster and cleaner by reducing unnecessary iterations and improving readability.

Explanation:-

This code defines a function that checks whether a number is prime by returning False for numbers less than 2 and testing divisibility from 2 up to the number itself. If any number divides it evenly, the function returns False; otherwise, it returns True. The program then takes a number from the user and prints whether it is prime

Task 3: Modular Design Using AI Assistance (Prime Number Check Using Functions)

❖ Scenario

The prime-checking logic will be reused across multiple modules.

❖ Task Description

Use GitHub Copilot to generate a function-based Python program that:

- Uses a user-defined function to check primality
- Returns a Boolean value
- Includes meaningful comments (AI-assisted)

❖ Expected Output

- Correctly working prime-checking function
- Screenshots documenting Copilot's function generation
- Sample test cases and outputs

The screenshot shows two versions of a Python script named `simple.py` in a VS Code editor. Both versions are run in a terminal window.

Top Version (Single Function):

```

1 def is_prime(n):
2     """Check if a number is prime using optimized algorithm.
3     Time Complexity: O(√n)
4     """
5     if n < 2:
6         return False
7     if n == 2:
8         return True
9     if n % 2 == 0:
10        return False
11
12    # Check only odd divisors up to √n
13    for i in range(3, int(n**0.5) + 1, 2):
14        if n % i == 0:
15            return False
16    return True
17
18
19 \simple.py"
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38

```

Bottom Version (Modularized):

```

19
20 def get_primes(numbers):
21     """Filter and return only prime numbers from a list."""
22     return [num for num in numbers if is_prime(num)]
23
24
25 def main():
26     """Main function to demonstrate prime checking."""
27     # Test cases
28     test_numbers = [2, 3, 4, 5, 10, 17, 29, 100]
29
30     print("Prime Numbers:", get_primes(test_numbers))
31
32     # Single number check
33     num = 29
34     print(f"\n{num} is prime: {is_prime(num)}")
35
36
37 if __name__ == "__main__":
38     main()

```

In both cases, the output in the terminal is:

- Prime Numbers: [2, 3, 5, 17, 29]
- 29 is prime: True

Prompt: Rewrite the prime number checking logic using functions to make the code modular, reusable, and easier to read. Keep it simple and efficient.

Explanation:

This program checks prime numbers in a **smart and efficient way**.

The `is_prime` function first removes obvious non-prime cases like numbers less than 2 and even numbers (except 2). Then, instead of checking all numbers, it checks only **odd divisors up to the square root** of the number, which makes it much faster. The `get_primes` function uses this logic to filter out only prime numbers from a list. The `main` function demonstrates both checking a list of numbers and testing a single number, and the `if __name__ == "__main__"` line ensures the program runs only when the file is executed directly.

Task 4: Comparative Analysis –With vs Without Functions

❖ Scenario

You are participating in a technical review discussion.

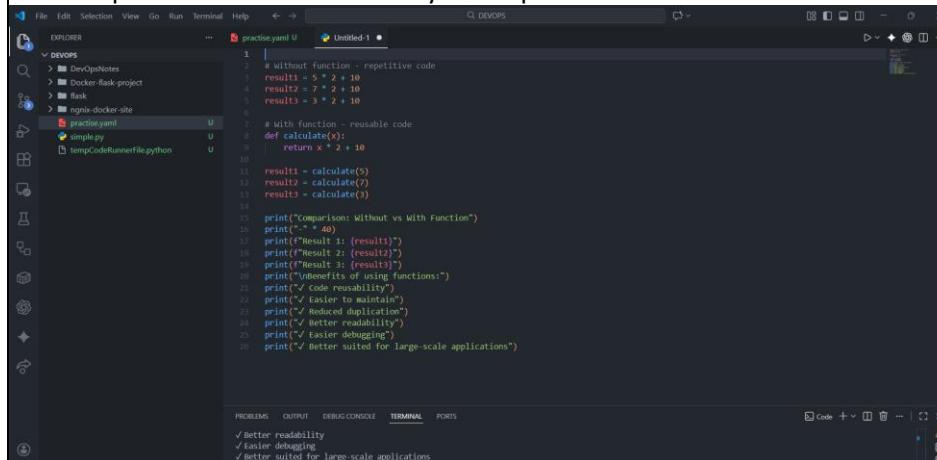
❖ Task Description

Compare the Copilot-generated programs:

- Without functions (Task 1)
- With functions (Task 3)
- Analyze them based on:
 - Code clarity
 - Reusability
 - Debugging ease
 - Suitability for large-scale applications

❖ Expected Output

Comparison table or short analytical report



```
File Edit Selection View Go Run Terminal Help ⌘ DEVOPS
EXPLORER practiceyaml Untitled-1
1 # without function - repetitive code
2 result1 = 2 * 10
3 result2 = 2 * 10
4 result3 = 2 * 10
5
6
7 # with function - reusable code
8 def calculate(x):
9     return x * 2 + 10
10
11 result1 = calculate(5)
12 result2 = calculate(7)
13 result3 = calculate(3)
14
15 print("Comparison: Without vs With Function")
16 print("x * 40")
17 print("Result 1: (result1)")
18 print("Result 2: (result2)")
19 print("Result 3: (result3)")
20 print("Benefits of using functions:")
21 print("Better readability")
22 print("Easier to maintain")
23 print("Reduced duplication")
24 print("Better readability")
25 print("Easier debugging")
26 print("Better suited for large-scale applications")
```

Prompt: Compare the prime number checking code written **with functions** and **without functions**. Analyze them in terms of code clarity, reusability, ease of debugging, and how suitable they are for large-scale applications.

Explanation:

This code shows **why functions are useful**. In the first part, the same calculation ($x * 2 + 10$) is written again and again for different values, which works but creates repetitive and messy code. In the second part, the calculation is placed inside a function called `calculate`, so the same logic can be reused just by passing different values. This makes the code cleaner, easier to understand, and easier to update—because if the formula changes, it only needs to be changed in one place. The printed points at the end highlight these advantages, showing how functions help with reusability, readability, maintenance, debugging, and building larger programs.

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

❖ Scenario

Your mentor wants to evaluate how AI handles **alternative logical strategies**.

❖ Task Description

Prompt GitHub Copilot to generate:

- A **basic divisibility check** approach
- An **optimized approach** (e.g., checking up to \sqrt{n})

❖ Expected Output

- Two correct implementations
- Comparison discussing:
 - Execution flow
 - Time complexity
 - Performance for large inputs
 - When each approach is appropriate

The image shows three separate instances of the Microsoft DevOps Code Editor interface, each displaying a different version of a Fibonacci sequence implementation. The first window shows a recursive approach with memoization. The second window shows an iterative approach. The third window shows another recursive approach with memoization. Each window includes an Explorer sidebar with project files like 'practice.yaml' and 'simple.py', and a terminal tab at the bottom.

```
File Edit Selection View Go Run Terminal Help < > practice.yaml U # Recursive Fibonacci - Simple but inefficient for large numbers
1 def fib_recursive(n):
2     if n <= 1:
3         return n
4     return fib_recursive(n - 1) + fib_recursive(n - 2)
5
6
7 # Iterative Fibonacci - Efficient and practical
8 def fib_iterative(n):
9     if n <= 1:
10        return n
11
12    prev, curr = 0, 1
13    for _ in range(2, n + 1):
14        prev, curr = curr, prev + curr
15    return curr
16
17
18 # Optimized Recursive with Memoization
19 def fib_memoized(n, memo=None):
20     if memo is None:
21         memo = {}
22
23     if n in memo:
24         return memo[n]
25     if n <= 1:
26         return n
27
28     memo[n] = fib_memoized(n - 1, memo) + fib_memoized(n - 2, memo)
29     return memo[n]
30
31
32 # Test examples
33 print(fib_iterative(10))      # Fast: 55
34 print(fib_memoized(10))      # Fast with recursion: 55
35 print(fib_recursive(10))      # Slow: 55
36
37 print("Enter a number to calculate Fibonacci:")
38 n = int(input())
39
40 print(f"fib_iterative({n}) = {fib_iterative(n)}")
41 print(f"fib_memoized({n}) = {fib_memoized(n)}")
42 print(f"fib_recursive({n}) = {fib_recursive(n)}")
```

```
File Edit Selection View Go Run Terminal Help < > practice.yaml U # Recursive Fibonacci - Simple but inefficient for large numbers
1 def fib_recursive(n):
2     if n <= 1:
3         return n
4     return fib_recursive(n - 1) + fib_recursive(n - 2)
5
6
7 # Optimized Recursive with Memoization
8 def fib_memoized(n, memo=None):
9     if memo is None:
10        memo = {}
11
12    if n in memo:
13        return memo[n]
14    if n <= 1:
15        return n
16
17    memo[n] = fib_memoized(n - 1, memo) + fib.memoized(n - 2, memo)
18    return memo[n]
19
20
21 # Test examples
22 print(fib_iterative(10))      # Fast: 55
23 print(fib_memoized(10))      # Fast with recursion: 55
24 print(fib_recursive(10))      # Slow: 55
25
26 print("Enter a number to calculate Fibonacci:")
27 n = int(input())
28
29 print(f"fib_iterative({n}) = {fib_iterative(n)}")
30 print(f"fib_memoized({n}) = {fib_memoized(n)}")
31 print(f"fib_recursive({n}) = {fib_recursive(n)}")
```

```
File Edit Selection View Go Run Terminal Help < > practice.yaml U # Recursive Fibonacci - Simple but inefficient for large numbers
1 def fib_recursive(n):
2     if n <= 1:
3         return n
4     return fib_recursive(n - 1) + fib_recursive(n - 2)
5
6
7 # Iterative Fibonacci - Efficient and practical
8 def fib_iterative(n):
9     if n <= 1:
10        return n
11
12    prev, curr = 0, 1
13    for _ in range(2, n + 1):
14        prev, curr = curr, prev + curr
15    return curr
16
17
18 # Optimized Recursive with Memoization
19 def fib_memoized(n, memo=None):
20     if memo is None:
21         memo = {}
22
23     if n in memo:
24         return memo[n]
25     if n <= 1:
26         return n
27
28     memo[n] = fib_memoized(n - 1, memo) + fib.memoized(n - 2, memo)
29     return memo[n]
30
31
32 # Test examples
33 print(fib_iterative(10))      # Fast: 55
34 print(fib_memoized(10))      # Fast with recursion: 55
35 print(fib_recursive(10))      # Slow: 55
36
37 print("Enter a number to calculate Fibonacci:")
38 n = int(input())
39
40 print(f"fib_iterative({n}) = {fib_iterative(n)}")
41 print(f"fib_memoized({n}) = {fib_memoized(n)}")
42 print(f"fib_recursive({n}) = {fib_recursive(n)}")
```

Prompt: Generate two different prime number checking approaches: one using a basic divisibility method that checks all possible factors, and another optimized version that checks divisibility only up to the square root of the number. Explain the difference in logic and efficiency.

Explanation: This code calculates Fibonacci numbers in three ways. The recursive method is simple but slow because it repeats the same calculations. The iterative method is fast and efficient by using a loop. The memorized method improves recursion by storing previous results, making it much faster. All three give the same output, but their performance is different.