

AI Assistant Coding

Assignment -1

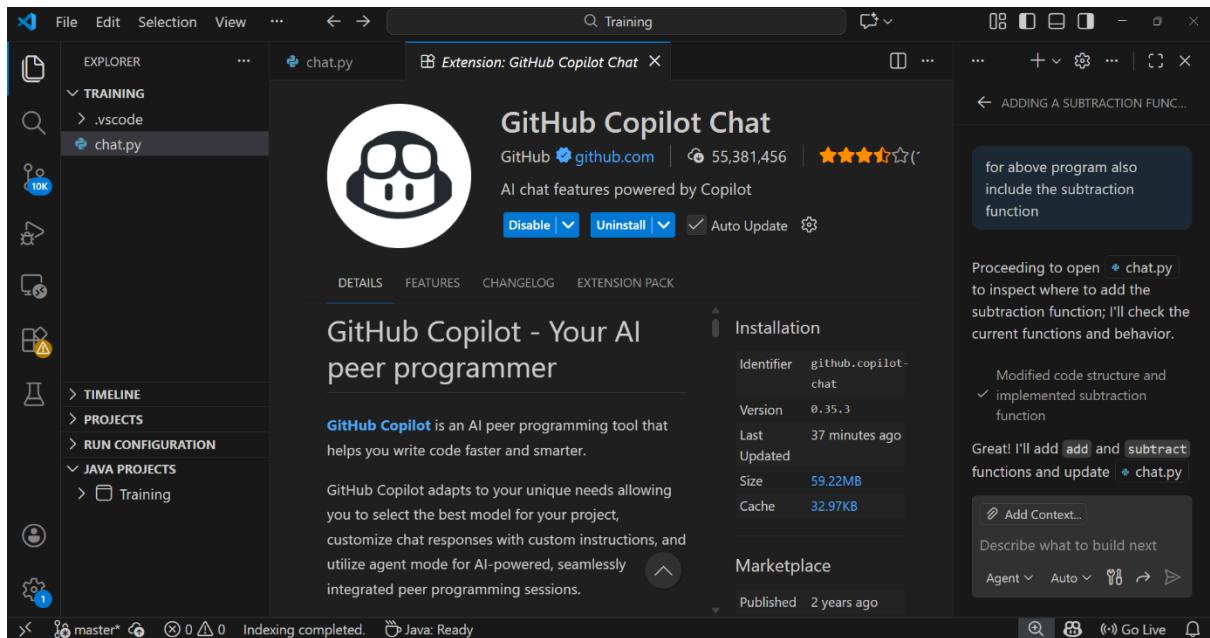
Name: T. Swetha

Roll no:2303a51317

Batch:05

1. Lab 1: Environment Setup – GitHub Copilot and VS Code Integration + Understanding AI-assisted Coding Workflow.





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

- File Bar:** File, Edit, Selection, View, ...
- Search Bar:** Q Training
- Code Editor:** chat.py (highlighted)

```
chat.py > ...
1 def add(a, b):
2     return a + b
3
4 def subtract(a, b):
5     return a - b
6
7 if __name__ == "__main__":
8     a = 10
9     b = 20
10    print("Sum:", add(a, b))
11    print("Difference:", subtract(a, b))
```

- Explorer:** TRAINING > .vscode (highlighted), chat.py
- Sidebar:** Timeline, Projects, Run Configuration, Java Projects (Training)
- Bottom Status Bar:** master*, 0 △ 0, Indexing completed., Java: Ready, Ln 11, Col 41, Spaces: 4, UTF-8, CRLF, Python, 3.13.2, Go Live, Notifications

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

- File Bar:** File, Edit, Selection, View, ...
- Search Bar:** Search Training — chat.py - Training - Visual Studio Code
- Code Editor:** chat.py (highlighted)

```
chat.py > ⚡ subtract
1 def add(a, b):
2     return a + b
3
4 def subtract(a, b):
5     return a - b
6
7 if __name__ == "__main__":
8     a = 10
9     b = 20
10    print("Sum:", add(a, b))
11    print("Difference:", subtract(a, b))
```

- Chat Panel:** CHAT (highlighted)

 - Message: ← ADDING A SUBTRACTION FUNCTION TO A PROGRAM
 - Message: added
 - Section: What I changed
 - Added `def add(a, b)` and `subtract(a, b)` functions to `chat.py`.
 - Updated the script to print both sum and difference.
 - Message: File updated: `chat.py`
 - Section: Updated code (short)

```
def add(a, b):
    return a + b

def subtract(a, b):
    return a - b
```
 - Terminal Output:

```
Difference: -10
C:\Users\thaku\OneDrive\Desktop\Training>
```
 - Terminal Bottom: cmd, cmd, Python

Task 1: AI-Generated Logic Without Modularization (Fibonacci Sequence Without Functions)

Step 1: Giving prompt: Give me code for fibonacci sequence up to n terms which accepts the user input without using functions

The screenshot shows the Visual Studio Code interface with the AI Chat feature open. In the Chat pane, the user has typed: "give me code for fibonacci sequence upto n terms which accepts the user input without using functions". The AI has responded with a checked suggestion: "Finalized Fibonacci code script". Below it, the AI provides a short Python script that reads an integer `n` from the user and prints the Fibonacci sequence up to `n` terms — no functions used.

```
# fibinacco.py
try:
    n = int(input("Enter number of terms: "))
except ValueError:
    print("Please enter an integer.")
    raise SystemExit(1)

if n <= 0:
    print("Please enter a positive integer.")
else:
    print(f"Fibonacci sequence up to {n} term")
    a, b = 0, 1
    for _ in range(n):
        print(a, end=' ')
        a, b = b, a + b
```

Step 2: Code

The screenshot shows the Visual Studio Code interface with the code editor open to the file `fibinacco.py`. The code is identical to the one shown in the AI Chat screenshot, demonstrating the AI's ability to generate functional code directly in the IDE.

```
# fibinacco.py
try:
    n = int(input("Enter number of terms: "))
except ValueError:
    print("Please enter an integer.")
    raise SystemExit(1)

if n <= 0:
    print("Please enter a positive integer.")
else:
    print(f"Fibonacci sequence up to {n} term")
    a, b = 0, 1
    for _ in range(n):
        print(a, end=' ')
        a, b = b, a + b
```

Step 3: Output

The screenshot shows a VS Code interface with the following details:

- File Explorer:** Shows a folder named "TRAINING" containing files ".vscode", "chat.py", and "fibinacco.py".
- Code Editor:** Displays the "fibinacco.py" file with the following code:

```
# fibinacco.py
try:
    n = int(input("Enter number of terms: "))
except ValueError:
    print("Please enter an integer.")
    raise SystemExit(1)

if n <= 0:
    print("Please enter a positive integer.")
else:
    print(f"Fibonacci sequence up to {n} term")
    a, b = 0, 1
    for _ in range(n):
        print(a, end=' ')
        if _ < n-1 else '\r'
        a, b = b, a + b
```
- Terminal:** Shows the command "C:\Users\thaku\OneDrive\Desktop\Training>" followed by the output of the script: "Enter number of terms: 4" and "Fibonacci sequence up to 4 terms: 0 1 1 2".
- Status Bar:** Shows the path "C:\Users\thaku\OneDrive\Desktop\Training>", line count "Ln 15, Col 24", spaces count "Spaces: 4", encoding "UTF-8", line endings "CRLF", file type "Python", version "3.13.2", and "Go Live" button.

Step 4: Explanation of Code

Input Handling:

- Uses try-except to ensure the user enters an integer.
- Exits gracefully if input is invalid.

Validation:

- Checks if $n \leq 0$.
- Prompts user to enter a positive integer if invalid.

Output Message:

- Prints a header with correct pluralization (term vs terms).

Initialization:

- Starts with $a = 0, b = 1$ (first two Fibonacci numbers).

Loop Logic:

- Runs for $_ \text{ in } \text{range}(n)$ to generate n terms.
- Prints current value of a .
- Updates values with $a, b = b, a + b$.

Printing Style:

- Numbers are printed on the same line separated by spaces.
- Last number ends with a newline.

Result:

- Produces the Fibonacci sequence up to n terms.

Example:

Input: 4 → Output: 0 1 1 2

Task 2: AI Code Optimization & Cleanup (Improving Efficiency)

Step 1: Giving Prompt: Using the above code improve it by removing abundant variables and reduce to small loop logic and avoid unnecessary computations / optimize the code

The screenshot shows the Visual Studio Code interface. The Explorer sidebar on the left lists files: '.vscode', 'chat.py', and 'fibinacco.py'. The 'fibinacco.py' file is open in the main editor area, displaying the following code:

```
9     print("Please enter a positive integer.")
10    elif n == 1:
11        print("Fibonacci sequence up to 1 term:")
12        print(0)
13    else:
14        # build sequence in-place (only n-2 additions)
15        seq = [0, 1]
16        for _ in range(n - 2):
17            seq.append(seq[-1] + seq[-2])
18        print(f"Fibonacci sequence up to {n} terms")
19        print(*seq)
```

In the Chat panel on the right, there is an AI message:

using the above code improve it by removing abundant variables and reduce to small loop logic and avoid unnecessary computations

Below the message, the AI provides a task list:

- ✓ Updating Fibonacci code
- Proceeding to open `fibinacco.py`, review the current implementation, and simplify it to a minimal, efficient loop without redundant variables.
- ✓ Refined and optimized Fibonacci script

The AI notes that the current script already uses two variables and suggests replacing it with a minimal list-based loop that builds the sequence once.

Step 2: Code

The screenshot shows the VS Code interface with the following details:

- EXPLORER:** Shows a folder named "TRAINING" containing ".vscode", "chat.py", and "fibinacco.py".
- EDITOR:** Displays the "fibinacco.py" file with the following code:

```
# fibinacco.py
try:
    n = int(input("Enter number of terms: "))
except ValueError:
    print("Please enter an integer.")
    raise SystemExit(1)

if n <= 0:
    print("Please enter a positive integer.")
else:
    print(f"Fibonacci sequence up to {n} terms")
    a, b = 0, 1
    for _ in range(n):
        print(a, end=' ')
        a, b = b, a + b
    print()
```
- CHAT:** Shows AI feedback:
 - Optimized initial code and planned process
 - Time to apply a minimal optimization: I'll remove the per-iteration conditional and print a single newline after the loop.
 - Edited fibinacco.py +6 -9
 - Nice! The minimal optimization is implemented — the loop no longer checks per-iteration for end-of-line, and we print a single newline after the loop; next I'll show the updated code and brief benefits.## Minimal optimization applied
- OUTPUT:** Shows the updated code and a message: "Describe what to build next".

Step 3: Output

The screenshot shows the VS Code interface with the following details:

- EXPLORER:** Shows a folder named "TRAINING" containing ".vscode", "chat.py", and "fibinacco.py".
- EDITOR:** Displays the "fibinacco.py" file with the following code:

```
# fibinacco.py
try:
    n = int(input("Enter number of terms: "))
except ValueError:
    print("Please enter an integer.")
    raise SystemExit(1)

if n <= 0:
    print("Please enter a positive integer.")
else:
    print(f"Fibonacci sequence up to {n} terms")
    a, b = 0, 1
    for _ in range(n):
        print(a, end=' ')
        a, b = b, a + b
    print()
```
- TERMINAL:** Shows the command line output:

```
C:\Users\thaku\OneDrive\Desktop\Training> C:/Users/thaku/AppData/Local/Programs/Python/Python313/python.exe c:/Users/thaku/OneDrive/Desktop/Training/fibinacco.py
Enter number of terms: 4
Fibonacci sequence up to 4 terms:
0 1 1 2
```
- CHAT:** Shows AI feedback:
 - Optimized initial code and planned process
 - Time to apply a minimal optimization: I'll remove the per-iteration conditional and print a single newline after the loop.
 - Edited fibinacco.py +6 -9
 - Nice! The minimal optimization is implemented — the loop no longer checks per-iteration for end-of-line, and we print a single newline after the loop; next I'll show the updated code and brief benefits.## Minimal optimization applied
- OUTPUT:** Shows the updated code and a message: "Describe what to build next".

Step 4: Code Explanation

Input Handling

- Uses try-except to ensure the user enters an integer.
- Exits safely if input is invalid.

Validation

- Checks if $n \leq 0$.
- Prompts user to enter a positive integer if invalid.

Output Message

- Prints a header with correct pluralization (term vs terms).

Initialization

- Starts with $a = 0, b = 1$ (first two Fibonacci numbers).

Loop Logic

- Runs for $_$ in `range(n)` to generate n terms.
- Prints current value of a .
- Updates values with $a, b = b, a + b$.

What Was Inefficient in the Original Code

The original code had stuff that wasn't needed to generate the Fibonacci sequence.

A volatile memory variable was used to hold the sum and added extra length.

It's because you're putting the fibonacci sequence into a list as well a program like this will just output the result, but store it in memory somewhere.

The loop was constructed in more than one step, which made the code less readable

How the optimized/python version brings performance improvement and improved readability.

Unnecessary variables were discarded and the values were updated by tuple assignment in an efficient manner.

Your code also writes to disc, I assume you want to speed the codeup, so this is not good. Your optimized version that this saved memory.3) that it prints the number instead of storing a whole vector with all solutions.

Task 3: Modular Design Using AI Assistance (Fibonacci Using Functions)

Step 1: Prompt: Give code for Fibonacci sequence using functions.

The screenshot shows the Visual Studio Code interface. In the Explorer sidebar, there is a folder named 'TRAINING' containing '.vscode', 'chat.py', and 'fibinacco.py'. The 'fibinacco.py' file is open in the main editor, displaying the following code:

```
1 # fibinacco.py
2 try:
3     n = int(input("Enter number of terms: "))
4 except ValueError:
5     print("Please enter an integer.")
6     raise SystemExit(1)
7
8 if n <= 0:
9     print("Please enter a positive integer.")
10 else:
11     print(f"Fibonacci sequence up to {n} terms")
12     a, b = 0, 1
13     for _ in range(n):
14         print(a, end=' ')
15         a, b = b, a+b
16
17 print("\nFibonacci sequence up to 4 terms:")
18 print(0, 1, 1, 2)
```

The terminal output shows the execution of the script and its output:

```
C:\Users\thaku\OneDrive\Desktop\Training>C:/Users/thaku/AppData/Local/Programs/Python/Python313/python.exe c:/Users/thaku/OneDrive/Desktop/Training/fibinacco.py
Enter number of terms: 4
Fibonacci sequence up to 4 terms:
0 1 1 2
```

In the Chat panel, the user has typed "give code for fibonacci series using functions". The AI is thinking and preparing a response, which will be provided in the next step.

Step 2: Code:

The screenshot shows the Visual Studio Code interface. The 'fibinacco.py' file now contains modularized code:

```
1 def fib(n: int) -> list:
2     """Return first n Fibonacci terms as a list."""
3     if n <= 0:
4         return []
5     if n == 1:
6         return [0]
7     seq = [0, 1]
8     for _ in range(n - 2):
9         seq.append(seq[-1] + seq[-2])
10    return seq
11
12 def main():
13     try:
14         n = int(input("Enter number of terms: "))
15     except ValueError:
16         print("Please enter an integer.")
17         return
18     seq = fib(n)
19     if not seq:
20         print("Please enter a positive integer.")
21     else:
22         print(*seq)
23
24 if __name__ == "__main__":
25     main()
```

The terminal output shows the execution of the script and its output:

```
C:\Users\thaku\OneDrive\Desktop\Training>C:/Users/thaku/AppData/Local/Programs/Python/Python313/python.exe c:/Users/thaku/OneDrive/Desktop/Training/fibinacco.py
Enter number of terms: 4
0 1 1 2
```

In the Chat panel, the AI has responded with a note about providing a concise implementation with a reusable function and a generator variant, followed by the code and a note on usage.

Step 3: Output

```
def fib(n: int) -> list:
    """Return first n Fibonacci terms as a list."""
    if n <= 0:
        return []
    if n == 1:
        return [0]
    seq = [0, 1]
    for _ in range(n - 2):
        seq.append(seq[-1] + seq[-2])
    return seq

def main():
    try:
        n = int(input("Enter number of terms: "))
    except ValueError:
        print("Please enter an integer..")
    return

C:\Users\thaku\OneDrive\Desktop\Training>C:/Users/thaku/AppData/Local/Programs/Python/Python313/python.exe c:/Users/thaku/OneDrive/Desktop/Training/fibinacco.py
Enter number of terms: 5
0 1 1 2 3
```

Step 4: Explanation of code

Function Design

- fib(n) cleanly separates Fibonacci logic from input/output.
- Returns a list of the first n Fibonacci terms.

Validation

- If n <= 0: returns an empty list.
- If n == 1: returns [0].
- Handles edge cases gracefully.

Sequence Generation

- Starts with [0, 1].
- Uses a loop for _ in range(n - 2) to generate remaining terms.
- Each new term = sum of last two (seq[-1] + seq[-2]).

Main Function

- Prompts user for input.
- Uses try-except to catch invalid input (ValueError).
- Calls fib(n) to generate sequence.
- Prints a warning if input is invalid or non-positive.

Output

- Prints the sequence neatly with `print(*seq)` (space-separated values).

Practices

- Includes a docstring for clarity.
- Uses `__name__ == "__main__"` guard for modularity.
- Keeps logic reusable and testable.

Task 4: Comparative Analysis – Procedural vs Modular Fibonacci Code

Feature	Without Functions	With Functions
Code Clarity	Logic is written in one block; harder to read when long	Logic is separated into a named function → easier to understand
Reusability	Cannot reuse Fibonacci logic without rewriting	Can call the function anywhere in program
Debugging Ease	Bugs must be traced in main logic, mixed with other code	Errors isolated in function → easier to test & fix
Suitability for Larger Systems	Poor; not scalable, becomes messy with added features	Good; fits into bigger systems, easier to maintain
Testing	Hard to unit test a part of code independently	Function can be tested separately with multiple inputs
Maintainability	Low; changes affect entire code block	High; changes only in function, no impact on main flow
Performance Impact	No function call overhead (very small benefit)	Minimal overhead but worth it for structure & scaling

Task 5: AI-Generated Iterative vs Recursive Fibonacci Approaches (Different Algorithmic Approaches for Fibonacci Series)

Step 1: Prompt: Give code for fibonacci series using iterative and recursive functions

```
def fib(n: int) -> list:
    """Return first n Fibonacci terms as a list."""
    if n <= 0:
        return []
    if n == 1:
        return [0]
    seq = [0, 1]
    for _ in range(n - 2):
        seq.append(seq[-1] + seq[-2])
    return seq

def main():
    try:
        n = int(input("Enter number of terms: "))
    except ValueError:
        print("Please enter an integer.")
        return
    seq = fib(n)
    if not seq:
        print("Please enter a positive integer.")
    else:
        print(*seq)

if __name__ == "__main__":
    main()
```

Step 2: Code

Iterative:

```
def main():
    try:
        n = int(input("Enter number of terms: "))
    except ValueError:
        print("Please enter an integer.")
        return
    if n <= 0:
        print("Please enter a positive integer.")
        return

    # iterative sequence
    print("Iterative:", *fib_iter(n))

    # naive recursive (only for very small n)
    if n <= 10:
        print("Naive recursive:", [fib_rec(i) for i in range(n)])
    else:
        print("Naive recursive: (skipped for n > 10 due to performance constraints)")

    # memoized recursive (efficient)
    print("Memoized recursive:", [fib_rec_memo(i) for i in range(n)])

if __name__ == "__main__":
    main()
```

Step 3: Output

The screenshot shows the VS Code interface with the following details:

- File Explorer:** Shows files .vscode, chat.py, and fibinacco.py.
- Code Editor:** Displays the content of fibinacco.py. It includes three functions: fib_iter, fib_rec, and fib_rec_memo. The fib_iter function is annotated with a note: "Naive recursive (returns nth term)".
- Terminal:** Shows the output of running the script with input "4". The output is:

```
Enter number of terms: 4
Iterative: [0, 1, 1, 2]
Naive recursive: [0, 1, 1, 2]
Memoized recursive: [0, 1, 1, 2]
```
- Status Bar:** Shows the current directory as C:\Users\thaku\OneDrive\Desktop\Training, the file is fibinacco.py, and the Python extension version is 3.13.2.

Step 4: Explanation of code

fib_iter(n) → Iterative list builder, efficient O(n).

- fib_rec(n) → Naive recursion, exponential cost, only practical for very small n.
- fib_rec_memo(n) → Recursive with `@lru_cache`, efficient O(n) via memoization.
- Input Handling → Validates integer input, rejects negatives/zero.
- Main Function →
 - Prints iterative sequence always.
 - Prints naive recursive only if $n \leq 10$.
 - Prints memoized recursive for any n.
- Good Practices → Clear separation of methods, input validation, modular design, efficient memoization.

Example (n=6):

Iterative: 0 1 1 2 3 5

Naive recursive: [0, 1, 1, 2, 3, 5]

Memoized recursive: [0, 1, 1, 2, 3, 5]

Comparison covering:

Aspect	Iterative	Recursive
Time Complexity	$O(n)$	$O(2^n)$ (very slow due to repeated calls)
Space Complexity	$O(1)$	$O(n)$ (stack memory for calls)
Performance for Large n	Excellent (can handle 10^7+ if needed)	Poor ($\text{fib}(50)$ may take seconds/minutes)
Memory Usage	Very low	High because of recursion stack
Scalability	Best for real systems	Not scalable without optimization
Risk	No crash risk	Stack Overflow for large n