

AI Assistant Coding Assignment-1

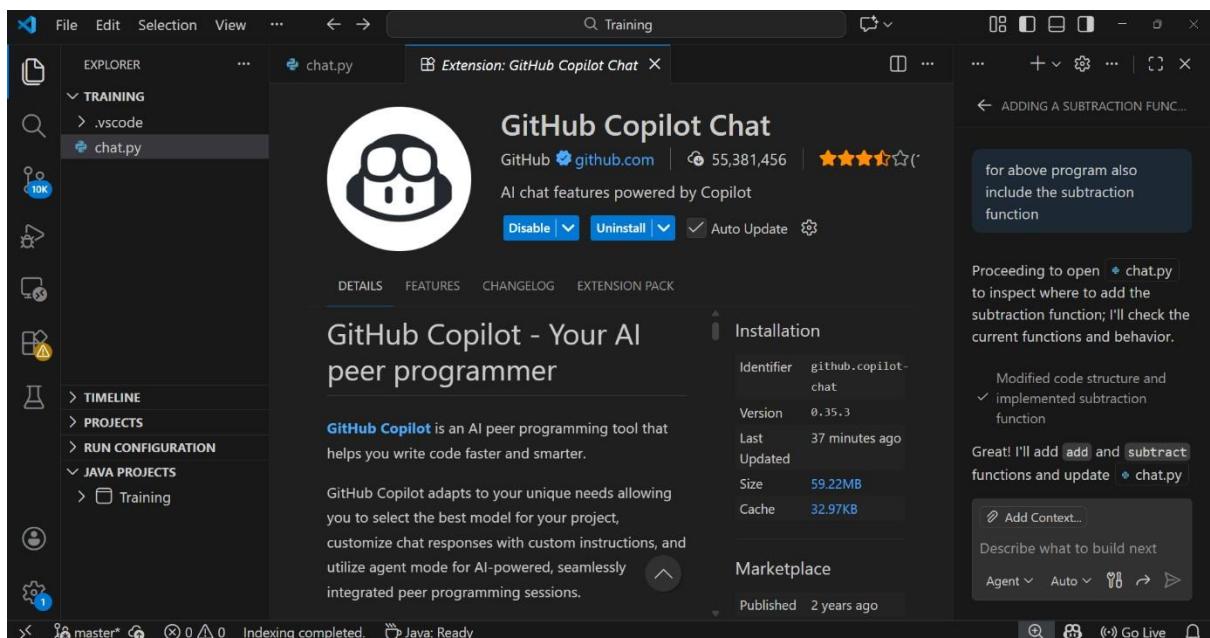
Name : Saketh Birru

Roll no : 2303A51403

Batch : 06

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





The screenshot shows a dark-themed instance of Visual Studio Code. The Explorer sidebar on the left lists a .vscode folder and a Java project named 'Training'. The main editor area displays a file named 'chat.py' containing the following Python code:

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

def subtract(a, b):
    return a - b

if __name__ == "__main__":
    a = 10
    b = 20
    print("Sum:", add(a, b))
    print("Difference:", subtract(a, b))
```

The status bar at the bottom indicates 'Indexing completed.' and 'Java: Ready'.

This screenshot shows the same instance of Visual Studio Code after an AI interaction. The 'chat.py' file now includes a new subtraction function and updated print statements. The right-hand panel displays the AI's notes:

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

The 'File updated' section shows the modified code:

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

def subtract(a, b):
    return a - b

if __name__ == "__main__":
    a = 10
    b = 20
    print("Sum:", add(a, b))
    print("Difference:", subtract(a, b))
```

The terminal at the bottom shows the output: 'Difference: -10'.

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

A screenshot of the Visual Studio Code (VS Code) interface. The top bar shows the title "Training". The left sidebar has sections for "EXPLORER", "TRAINING" (containing ".vscode", "chat.py", and "fibinacco.py"), and "TIMELINE". The main editor area shows a Python file "fibinacco.py" with the following code:

```
1  # Generate code (Ctrl+I), or select a language (Ctrl+K M). Start typing to dismiss or don't show this again.
```

The status bar at the bottom indicates "Ln 1, Col 1" and "Spaces: 4".

The right side features a "CHAT" panel titled "FIBONACCI SEQUENCE CODE WITHOUT FUNCTIONS". It contains a message from the AI asking for Fibonacci sequence code up to n terms without functions. A note says "Finalized Fibonacci code script". Below it, a snippet of code is shown:

```
# fibinacco.py
```

Step 2: Code

A screenshot of the Visual Studio Code (VS Code) interface, similar to the previous one but showing the completion of the code. The editor now contains:

```
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} term")
12     a, b = 0, 1
13     for _ in range(n):
14         print(a, end=' ' if _ < n-1 else '\r')
15         a, b = b, a + b
```

The status bar at the bottom indicates "Ln 15, Col 24".

The right side "CHAT" panel shows the AI's response to the completed code:

Here's 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")
```

Step 3: Output

The screenshot shows the 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 line output:

```
Enter number of terms: 4
Fibonacci sequence up to 4 terms:
0 1 1 2
```
- Status Bar:** Shows the path "C:\Users\thaku\OneDrive\Desktop\Training", line 15, column 24, spaces 4, encoding UTF-8, and version 3.13.2.

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 $_ \in \text{range}(n)$ 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 a Microsoft Visual Studio Code (VS Code) interface. On the left is the Explorer sidebar with a 'TRAINING' folder containing '.vscode', 'chat.py', and 'fibinacco.py'. The main editor area shows a Python script named 'fibinacco.py' with 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)
```

To the right of the editor is the AI Chat interface. A message from the AI says: "using the above code improve it by removing abundant variables and reduce to small loop logic and avoid unnecessary computations". Below this, a list of tasks is shown:

- ✓ 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: "Perfect! The current script already uses two variables; I'll replace it with a minimal list-based loop that builds the sequence once". At the bottom of the AI Chat window, there is a text input field with placeholder text "Describe what to build next" and a button labeled "Agent".

Step 2: Code

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

- File Explorer:** Shows files in the 'TRAINING' folder: '.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} terms")
    a, b = 0, 1
    for _ in range(n):
        print(a, end=' ')
        a, b = b, a + b
    print()
```
- Chat Panel:** Shows a conversation with an AI agent about optimization.
- Status Bar:** Shows the file is updated, the current line is Ln 16, Col 12, and the Python version is 3.13.2.

Step 3: Output

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

- File Explorer:** Shows files in the 'TRAINING' folder: '.vscode', 'chat.py', and 'fibinacco.py'.
- Code Editor:** Displays the 'fibinacco.py' file with the same code as in Step 2.
- Terminal:** Shows the output of running the script:

```
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 Panel:** Shows a conversation with an AI agent about optimization.
- Status Bar:** Shows the file is updated, the current line is Ln 17, Col 5, and the Python version is 3.13.2.

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 $\text{range}(n)$ 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 a VS Code interface with the following details:

- EXPLORER** sidebar: TRAINING folder containing .vscode, chat.py, and fibinacco.py.
- EDITOR**: A code editor window titled "fibinacco.py" showing the following Python 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):
```
- TERMINAL**: Shows the command line output of running the script:

```
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**: A sidebar with a message from a user asking for code for the Fibonacci series using functions.
- Bottom Status Bar**: Shows the file path "C:\Users\thaku\OneDrive\Desktop\Training\", line 17, column 5, and other VS Code settings like Python 3.13.2 and Go Live.

Step 2: Code:

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

- EXPLORER** sidebar: TRAINING folder containing .vscode, chat.py, and fibinacco.py.
- EDITOR**: A code editor window titled "fibinacco.py" showing the following Python 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     seq = fib(n)
18     if not seq:
19         print("Please enter a positive integer.")
20     else:
21         print(*seq)
22
23 if __name__ == "__main__":
24     main()
```
- CHAT**: A sidebar with a message from a user confirming they have provided and presented the Fibonacci code.
- Bottom Status Bar**: Shows the file path "C:\Users\thaku\OneDrive\Desktop\Training\", line 25, column 11, and other VS Code settings like Python 3.13.2 and Go Live.

Step 3: Output

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

- EXPLORER:** Shows files .vscode, chat.py, and fibinacco.py.
- CHAT:** A sidebar titled "FIBONACCI SEQUENCE CODE WITHOUT FUNCTIONS" contains a note about Fibonacci using functions and a checked checkbox for "Iterative function (returns list)".
- TERMINAL:** Displays the following text:

```
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
```
- PROBLEMS:** Shows two Python errors.
- OUTPUT:** Shows the command run in the terminal.
- TIMELINE:** Shows recent activity.

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 fibanocci series using iterative and recursive functions

The screenshot shows the Visual Studio Code interface with the Chat feature open. The user has typed "Give code for fibannoci dequence using iterative and recursive functions". The AI is responding with examples:

```
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()
```

Thinking...

- Implementing Fibonacci Functions
I'm setting out to provide both an iterative function, `fib_iter(n)`, that returns a list of the first n terms, and a recursive function, `fib_rec(n)`, that returns the nth

Describe what to build next

Step 2: Code Iterative:

The screenshot shows the Visual Studio Code interface with the Chat feature open. The AI has provided the iterative Fibonacci code and examples of its usage:

```
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.")
    else:
        # 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 + 1)])
        else:
            print("Naive recursive: (skipped for n > 10 due to inefficiency)")

        # memoized recursive (efficient)
        print("Memoized recursive:", [fib_rec_memo(i) for i in
                                         range(n + 1)])

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

FIBONACCI SEQUENCE CODE WITHOUT FUNCTIONS

```
if n < 0:
    raise ValueError("n must be non-negative")
if n < 2:
    return n
return fib_rec_memo(n - 1) + fib_rec_memo(n - 2)
```

Example usage (prompt + print)

```
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.")
    else:
        # 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 + 1)])
        else:
            print("Naive recursive: (skipped for n > 10 due to inefficiency)")

        # memoized recursive (efficient)
        print("Memoized recursive:", [fib_rec_memo(i) for i in
                                         range(n + 1)])
```

Describe what to build next

Step 3: Output

The screenshot shows the VS Code interface with three Python files open in the Explorer sidebar: `.vscode`, `chat.py`, and `fibinacco.py`. The `fibinacco.py` file contains three functions:

```

def fib_iter(n: int) -> 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 fib_rec(n: int) -> int:
    """Naive recursive: return the nth Fibonacci number (0-indexed)."""
    if n < 0:
        raise ValueError("n must be non-negative")
    if n < 2:
        return n
    else:
        return fib_rec(n - 1) + fib_rec(n - 2)

def fib_rec_memo(n: int) -> int:
    """Return the nth Fibonacci number via memoization.
    If n < 0, raise ValueError.
    If n < 2, return n.
    Otherwise, return fib_rec_memo(n - 1) + fib_rec_memo(n - 2).
    """
    if n < 0:
        raise ValueError("n must be non-negative")
    if n < 2:
        return n
    else:
        return fib_rec_memo(n - 1) + fib_rec_memo(n - 2)

```

The terminal output shows the results for `fib_iter(6)`:

```

Enter number of terms: 4
Iterative: [0, 1, 1, 2]
Naive recursive: [0, 1, 1, 2]
Memoized recursive: [0, 1, 1, 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 → o Prints iterative sequence always. o Prints naive recursive only if $n \leq 10$. o 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