

AI Assistant Coding

Assignment -1

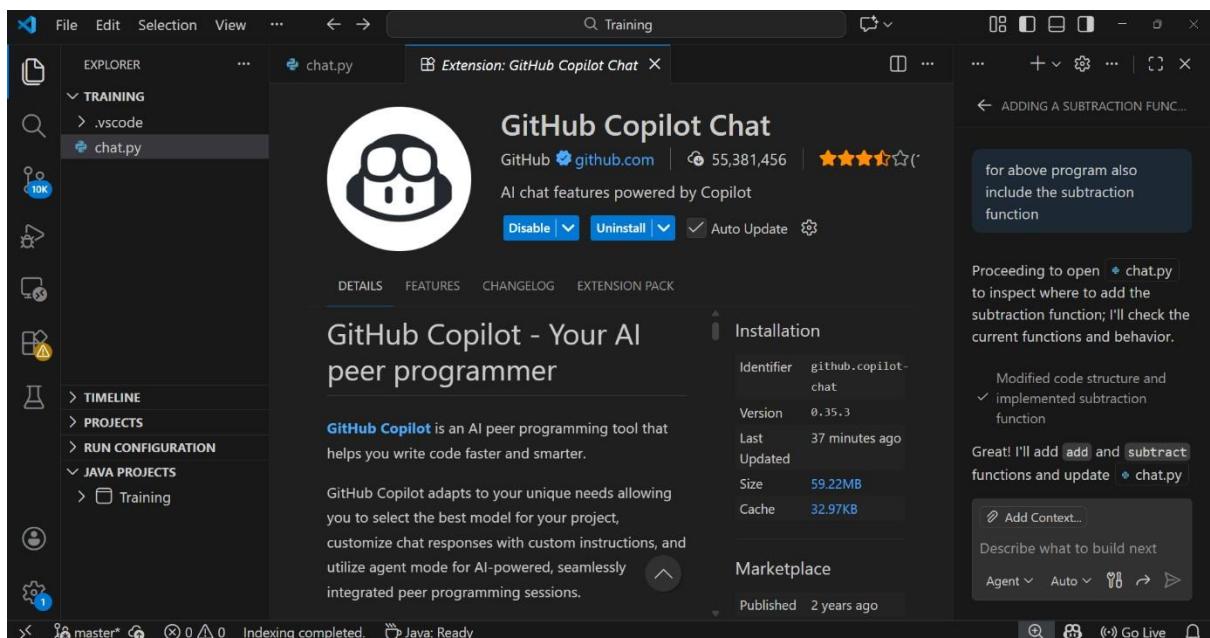
Name: Aman Sarkar

Ht.no: 2303A51273

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 a dark theme. The left sidebar includes sections for EXPLORER, TRAINING, and JAVA PROJECTS. The TRAINING section contains a .vscode folder and a file named chat.py. The code editor displays the following Python script:

```
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))
```

The status bar at the bottom indicates "Indexing completed.", "Java: Ready", and other system details.

This screenshot shows the same Visual Studio Code interface, but with a Chat panel open on the right side. The Chat panel has a title "Training — chat.py - Training - Visual Studio Code" and a message "← ADDING A SUBTRACTION FUNCTION TO A PROGRAM". Below it, a "What I changed" section lists:

- Added `def 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 updated code:

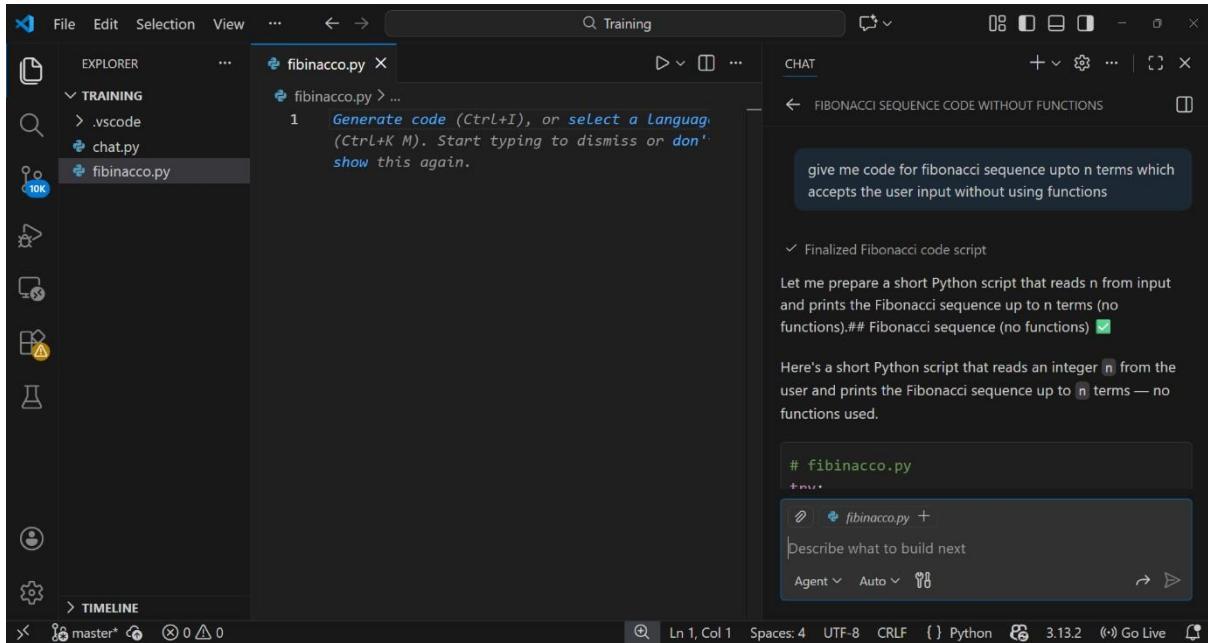
```
def add(a, b):
    return a + b

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

The Chat panel also includes a "Describe what to build next" input field and some status indicators at the bottom.

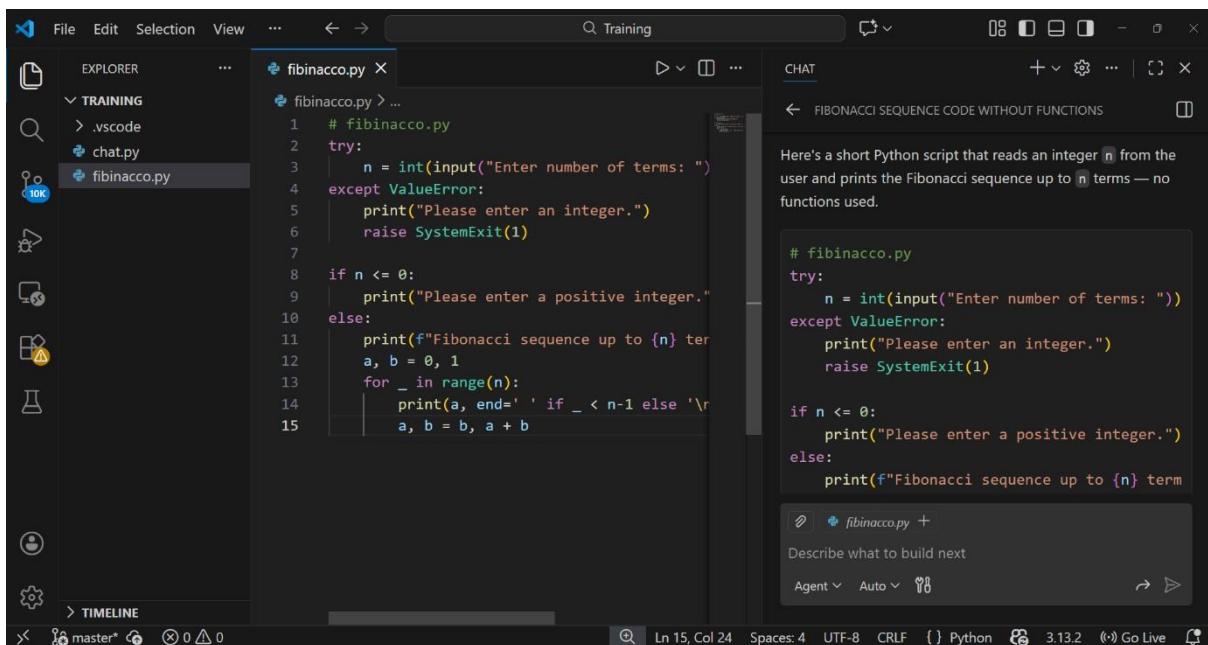
Task 1: AI-Generated Logic Without Modularization (Fibonacci Sequence Without Functions)

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



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

Step 2: Code



```
# fibinacco.py
# fibinacco.py > ...
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=' ')
15         a, b = b, a + b
```

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 Python code for generating a Fibonacci sequence.
- Terminal:** Shows the command "python fibinacco.py" being run, followed by the output: "Enter number of terms: 4" and "Fibonacci sequence up to 4 terms: 0 1 1 2".
- Status Bar:** Shows the file path "C:\Users\thaku\OneDrive\Desktop\Training", line count "Ln 15, Col 24", and other settings.

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 the Visual Studio Code (VS Code) interface. On the left is the Explorer sidebar with a 'TRAINING' folder containing files like '.vscode', 'chat.py', and 'fibinacco.py'. The main editor area displays the following Python 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:\n{seq}")
19

```

To the right of the editor is a Chat interface titled 'FIBONACCI SEQUENCE CODE WITHOUT FUNCTIONS'. 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

The AI has a checklist and notes:

- ✓ 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
- Perfect! The current script already uses two variables; I'll replace it with a minimal list-based loop that builds the sequence once

A message input field says 'Describe what to build next'.

Step 2: Code

The screenshot shows the VS Code interface. The left sidebar has 'EXPLORER' and 'TRAINING' sections. The 'TRAINING' section contains files '.vscode', 'chat.py', and 'fibinacco.py'. The main editor window 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()
```

The right side features a 'CHAT' panel with a message from 'FIBONACCI SEQUENCE CODE WITHOUT FUNCTIONS' suggesting optimization by removing the per-iteration conditional check. A note indicates that the minimal optimization has been applied.

Step 3: Output

The screenshot shows the VS Code interface with the 'TERMINAL' tab selected. The terminal window displays the following session:

```
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
```

The right side of the screen shows the same 'CHAT' panel as in Step 2, with the message about the minimal optimization being applied.

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 VS Code interface. On the left is the Explorer sidebar with files like '.vscode', 'chat.py', and 'fibinacco.py'. The main editor window contains the following Python 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
```

The terminal tab shows the output of running the script: "Enter number of terms: 4" followed by "Fibonacci sequence up to 4 terms: 0 1 1 2". The status bar at the bottom indicates the file is saved in 'master' branch.

In the top right, there's a 'CHAT' interface titled 'FIBONACCI SEQUENCE CODE WITHOUT FUNCTIONS'. A message box says 'give code for fibonacci series using functions'. Below it, a 'Thinking...' section and a 'Preparing the response' section are visible, along with a code completion suggestion for 'fibinacco.py'.

Step 2: Code:

This screenshot shows the same VS Code setup as above, but the AI chat interface has provided some code completion. The 'Thinking...' section now includes a note about generating memory-efficient code. The 'Preparing the response' section discusses keeping the response concise. The code completion suggestion for 'fibinacco.py' now includes a checked checkbox for 'Iterative function (returns list)'.

The code completion suggestion box displays the following code:

```
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
```

The status bar at the bottom indicates the file is saved in 'master' branch.

Step 3: Output

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

- EXPLORER:** Shows files .vscode, chat.py, and fibinacco.py.
- EDITOR:** Displays the content of fibinacco.py. The code defines two functions: fib(n) which returns a list of Fibonacci terms, and main() which prompts the user for the number of terms and prints the result.
- TERMINAL:** Shows the output of running the script: "Enter number of terms: 5" followed by the sequence [0, 1, 2, 3].
- CHAT:** A sidebar titled "FIBONACCI SEQUENCE CODE WITHOUT FUNCTIONS" contains a message from ChatGPT: "I'm going to provide a concise implementation with a reusable function and a generator variant; next I'll show the code and a short note on usage.## Fibonacci using functions". Below it, a section titled "Iterative function (returns list)" has a checked checkbox and displays the same code as the editor.

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

Sequence Generation

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

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

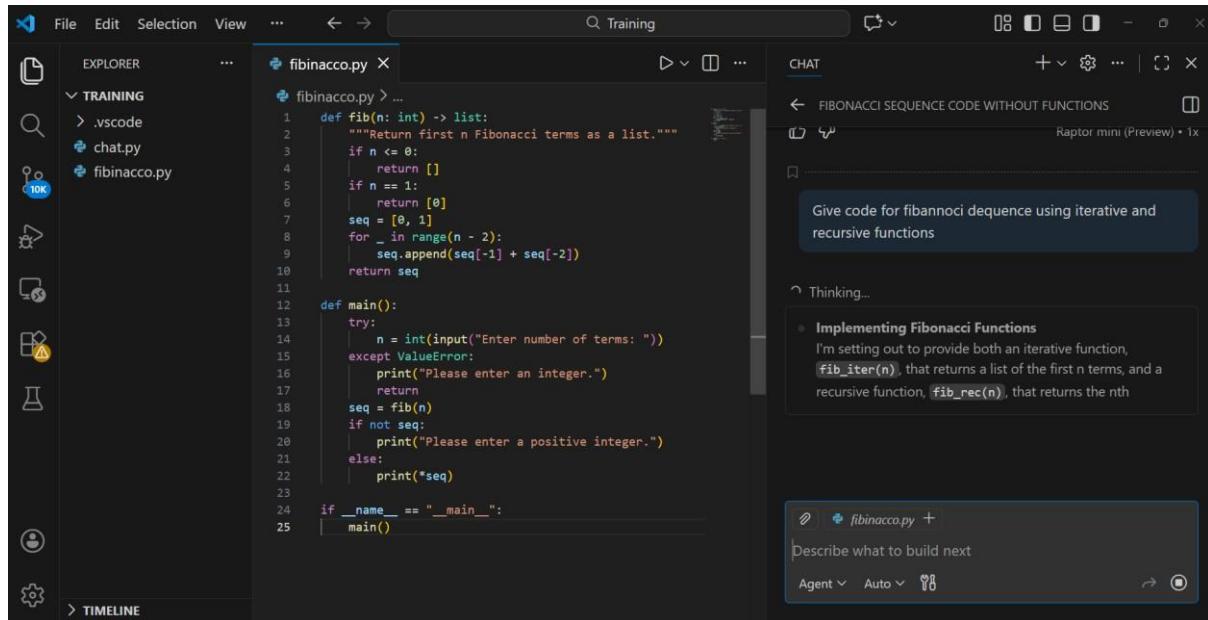
- Includes a docstring for clarity.
- Uses `__name__ == "__main__"` guard for modularity.
- Keeps logic reusable and testable.

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



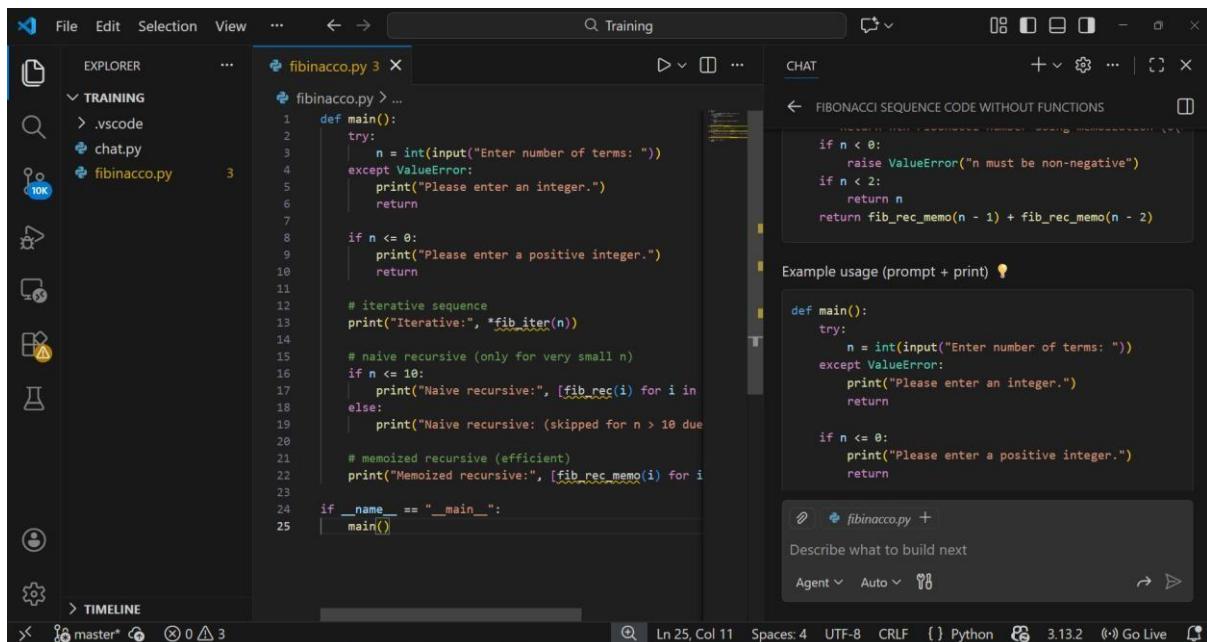
```
File Edit Selection View ... < > Training CHAT
EXPLORER fibinocco.py > ...
TRAINING .vscode chat.py fibinocco.py
fibinocco.py > ...
1 def fib(n: int) -> list:
2     """Return first n Fibonacci terms as a list."""
3     if n <= 0:
4         return []
5     if n == 1:
6         return [0]
7     seq = [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()

Give code for fibanocci dequence using iterative and recursive functions

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

fibinocco.py +
Describe what to build next
Agent Auto 90
```

Step 2: Code Iterative:



```
File Edit Selection View ... < > Training CHAT
EXPLORER fibinocco.py 3 > ...
TRAINING .vscode chat.py fibinocco.py 3
fibinocco.py 3 > ...
1 def main():
2     try:
3         n = int(input("Enter number of terms: "))
4     except ValueError:
5         print("Please enter an integer.")
6         return
7
8     if n <= 0:
9         print("Please enter a positive integer.")
10    return
11
12 # iterative sequence
13 print("Iterative:", *fib_iter(n))
14
15 # naive recursive (only for very small n)
16 if n <= 10:
17     print("Naive recursive:", [fib_rec(i) for i in
18     else:
19         print("Naive recursive: (skipped for n > 10 due to time complexity)")
20
21 # memoized recursive (efficient)
22 print("Memoized recursive:", [fib_rec_memo(i) for i in
23
24 if __name__ == "__main__":
25     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.")
        return

    print("Iterative:", *fib_iter(n))
    print("Naive recursive:", [fib_rec(i) for i in
    else:
        print("Naive recursive: (skipped for n > 10 due to time complexity)")
    print("Memoized recursive:", [fib_rec_memo(i) for i in

fibinocco.py +
Describe what to build next
Agent Auto 90
```

Step 3: Output

The screenshot shows the VS Code interface with three code snippets side-by-side:

- fibonacci.py** (Iterative): A function `fib_iter` that uses a loop to build a sequence from 0 to n .
- Naive recursive (returns nth term)**: A function `fib_rec` that uses direct recursion to calculate the n th Fibonacci number.
- def fib_rec(n: int)**: A partially visible function definition for `fib_rec`.

The terminal output shows the results for $n=4$:

```
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 →
 - 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