Assignment 11.2

Name :R Ajay

HT.NO : 2303A51072

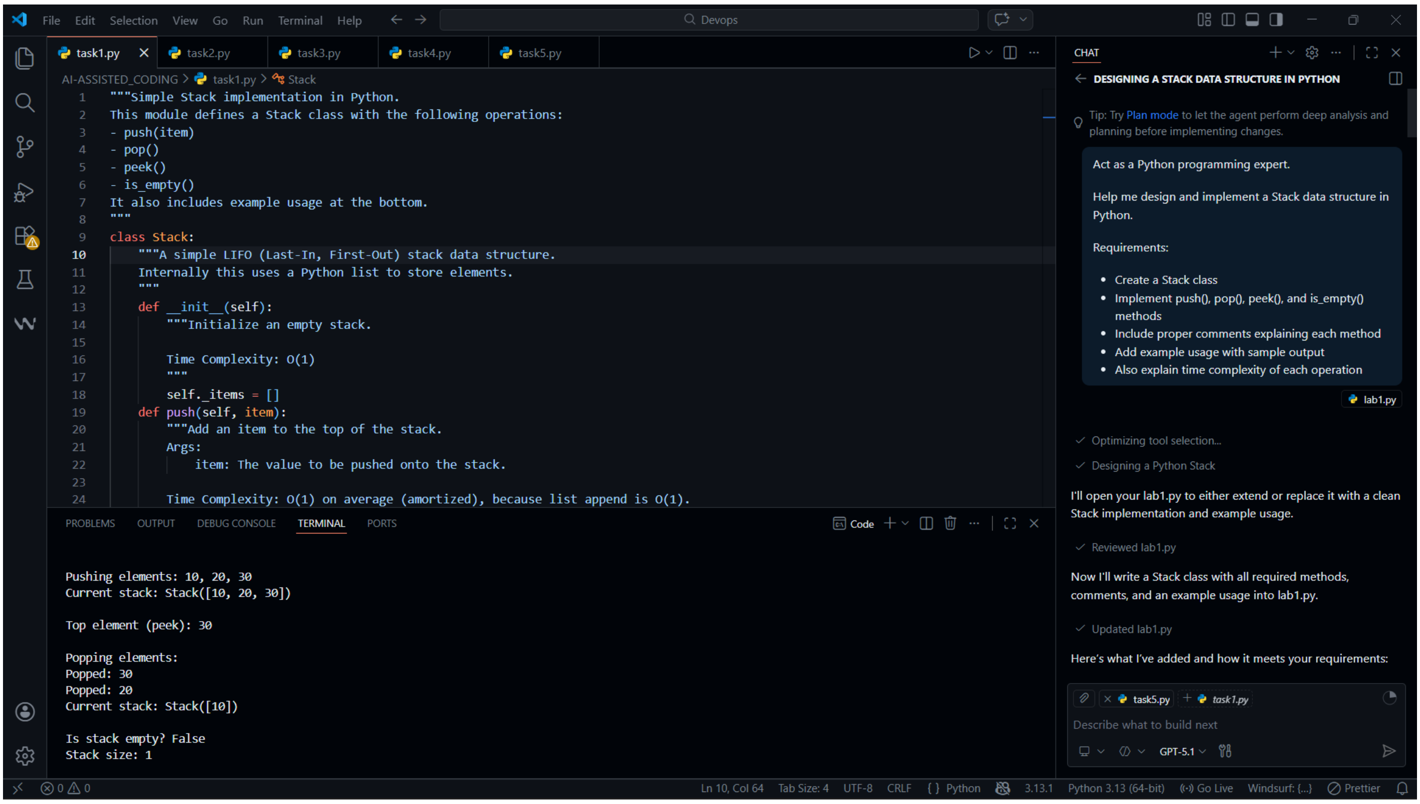
B.NO : 16

Task Description -1 – (Stack Using AI Guidance)

- **Task: With the help of AI, design and implement a Stack data structure supporting basic stack operations.**

Expected Output:

- **A Python Stack class supporting push, pop, peek, and empty-check operations with proper documentation.**


File Edit Selection View Go Run Terminal Help
Devops
task1.py
task2.py
task3.py
task4.py
task5.py
AI-ASSISTED_CODING > task1.py > Stack
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
Code
CHAT
Ln 10, Col 64 Tab Size: 4 UTF-8 CRLF {} Python 3.13.1 Python 3.13 (64-bit) Go Live Windsurf: {...} Prettier

```python
"""Simple Stack implementation in Python.
This module defines a Stack class with the following operations:
- push(item)
- pop()
- peek()
- is_empty()
It also includes example usage at the bottom.
"""
class Stack:
    """A simple LIFO (Last-In, First-Out) stack data structure.
    Internally this uses a Python list to store elements.
    """
    def __init__(self):
        """Initialize an empty stack.

        Time Complexity: O(1)
        """
        self._items = []
    def push(self, item):
        """Add an item to the top of the stack.
        Args:
            item: The value to be pushed onto the stack.

        Time Complexity: O(1) on average (amortized), because list append is O(1).
```

```
Pushing elements: 10, 20, 30
Current stack: Stack([10, 20, 30])

Top element (peek): 30

Popping elements:
Popped: 30
Popped: 20
Current stack: Stack([10])

Is stack empty? False
Stack size: 1
```

DESIGNING A STACK DATA STRUCTURE IN PYTHON

Tip: Try Plan mode to let the agent perform deep analysis and planning before implementing changes.

Act as a Python programming expert.

Help me design and implement a Stack data structure in Python.

Requirements:

- Create a Stack class
- Implement push(), pop(), peek(), and is_empty() methods
- Include proper comments explaining each method
- Add example usage with sample output
- Also explain time complexity of each operation

lab1.py

✓ Optimizing tool selection...

✓ Designing a Python Stack

I'll open your lab1.py to either extend or replace it with a clean Stack implementation and example usage.

✓ Reviewed lab1.py

Now I'll write a Stack class with all required methods, comments, and an example usage into lab1.py.

✓ Updated lab1.py

Here's what I've added and how it meets your requirements:

task5.py task1.py

Describe what to build next

GPT-5.1

Implemented a Stack class backed by a Python list.

Methods:

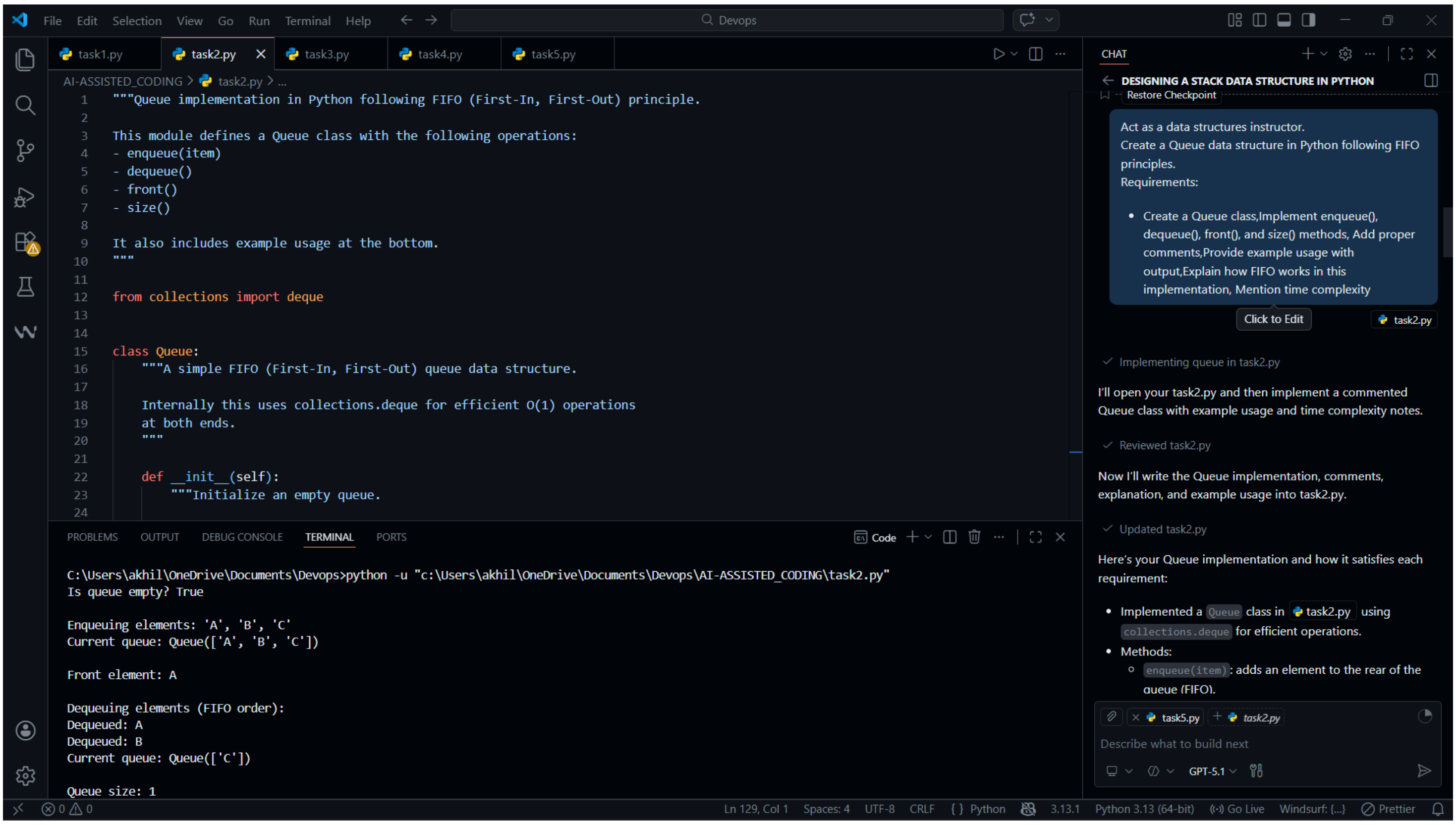
- push(item): adds an element to the top.
- pop(): removes and returns the top element, raises IndexError if empty.
- peek(): returns (without removing) the top element, raises IndexError if empty.
  is_empty(): returns True if there are no elements.

Task Description -2 – (Queue Design)

• Task:Use AI assistance to create a Queue data structure following FIFO principles

Expected Output:

• A complete Queue implementation including enqueue, dequeue, front element access, and size calculation



```python
"""Queue implementation in Python following FIFO (First-In, First-Out) principle.

This module defines a Queue class with the following operations:
- enqueue(item)
- dequeue()
- front()
- size()

It also includes example usage at the bottom.
"""

from collections import deque


class Queue:
    """A simple FIFO (First-In, First-Out) queue data structure.

    Internally this uses collections.deque for efficient O(1) operations
    at both ends.
    """

    def __init__(self):
        """Initialize an empty queue.

```

```
C:\Users\akhil\OneDrive\Documents\Devops>python -u "c:\Users\akhil\OneDrive\Documents\Devops\AI-ASSISTED_CODING\task2.py"
Is queue empty? True

Enqueuing elements: 'A', 'B', 'C'
Current queue: Queue(['A', 'B', 'C'])

Front element: A

Dequeuing elements (FIFO order):
Dequeued: A
Dequeued: B
Current queue: Queue(['C'])

Queue size: 1
```

DESIGNING A STACK DATA STRUCTURE IN PYTHON

Act as a data structures instructor.
Create a Queue data structure in Python following FIFO principles.
Requirements:

- Create a Queue class,Implement enqueue(), dequeue(), front(), and size() methods, Add proper comments,Provide example usage with output,Explain how FIFO works in this implementation, Mention time complexity

✓ Implementing queue in task2.py

I'll open your task2.py and then implement a commented Queue class with example usage and time complexity notes.

✓ Reviewed task2.py

Now I'll write the Queue implementation, comments, explanation, and example usage into task2.py.

✓ Updated task2.py

Here's your Queue implementation and how it satisfies each requirement:

- Implemented a `Queue` class in task2.py using `collections.deque` for efficient operations.
- Methods:
  - `enqueue(item)`: adds an element to the rear of the queue (FIFO).

- Implemented a Queue class using collections.deque for efficient operations.
- Methods:

  enqueue(item): adds an element to the rear of the queue (FIFO).

dequeue(): removes and returns the front element; raises IndexError if empty.

front(): returns (without removing) the front element; raises IndexError if empty.

size(): returns the current number of elements.

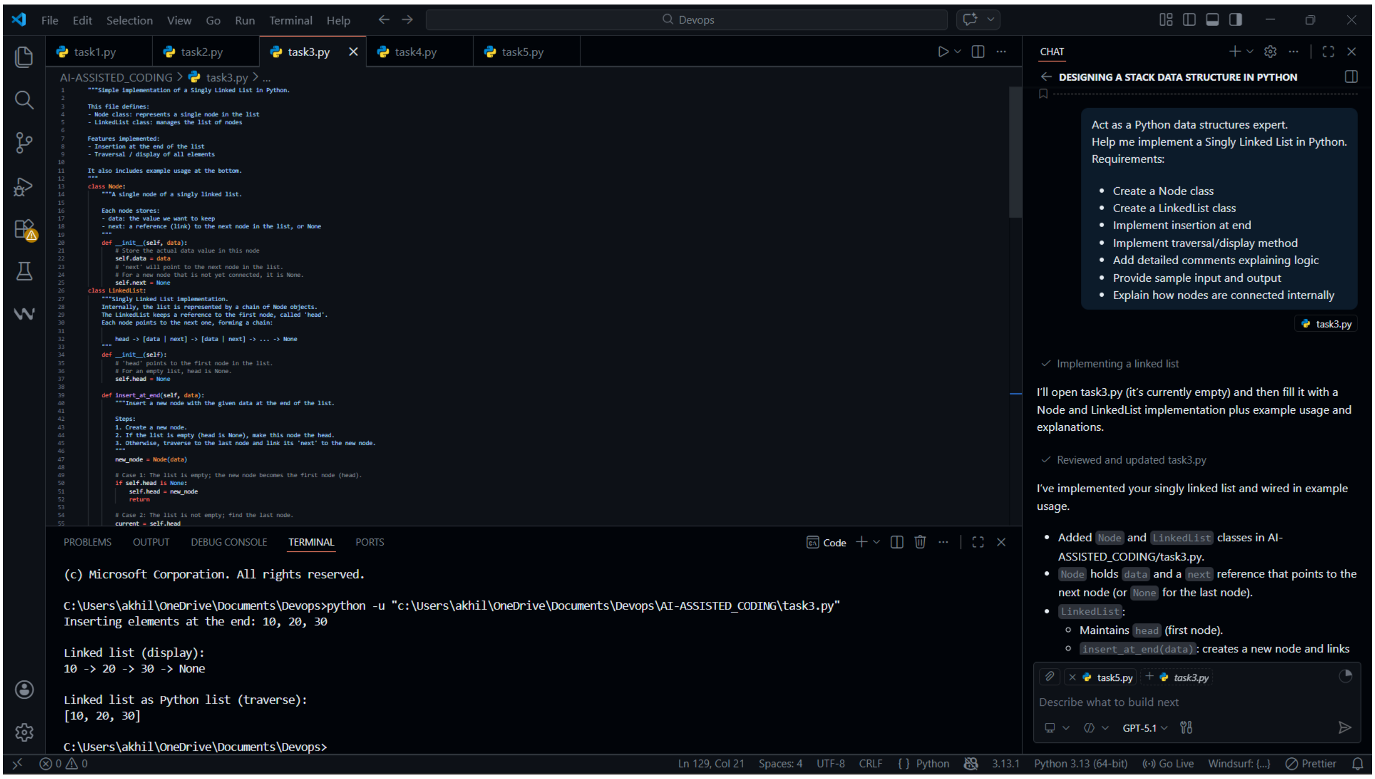
is_empty(): helper to check if the queue has no elements.

Task Description -3 –(Singly Linked List Construction)

• Task: Utilize AI to build a singly linked list supporting insertion and traversal.

Expected Output:

• Correctly functioning linked list with node creation, insertion logic, and display functionality.



```python
"""Simple implementation of a Singly Linked List in Python.

This file defines:
- Node class: represents a single node in the list
- LinkedList class: manages the list of nodes

Features implemented:
- Insertion at the end of the list
- Traversal / display of all elements

It also includes example usage at the bottom.
"""
class Node:
    """A single node of a singly linked list.

    Each node stores:
    - data: the value we want to keep
    - next: a reference (link) to the next node in the list, or None
    """
    def __init__(self, data):
        # Store the actual data value in this node
        self.data = data
        # 'next' will point to the next node in the list.
        # For a new node that is not yet connected, it is None.
        self.next = None
class LinkedList:
    """Singly Linked List implementation.
    Internally, the list is represented by a chain of Node objects.
    The LinkedList keeps a reference to the first node, called 'head'.
    Each node points to the next one, forming a chain:

        head -> [data | next] -> [data | next] -> ... -> None
    """
    def __init__(self):
        # 'head' points to the first node in the list.
        # For an empty list, head is None.
        self.head = None

    def insert_at_end(self, data):
        """Insert a new node with the given data at the end of the list.

        Steps:
        1. Create a new node.
        2. If the list is empty (head is None), make this node the head.
        3. Otherwise, traverse to the last node and link its 'next' to the new node.
        """
        new_node = Node(data)

        # Case 1: The list is empty; the new node becomes the first node (head).
        if self.head is None:
            self.head = new_node
            return

        # Case 2: The list is not empty; find the last node.
        current = self.head
```

```
(c) Microsoft Corporation. All rights reserved.

C:\Users\akhil\OneDrive\Documents\Devops>python -u "c:\Users\akhil\OneDrive\Documents\Devops\AI-ASSISTED_CODING\task3.py"
Inserting elements at the end: 10, 20, 30

Linked list (display):
10 -> 20 -> 30 -> None

Linked list as Python list (traverse):
[10, 20, 30]

C:\Users\akhil\OneDrive\Documents\Devops>
```

DESIGNING A STACK DATA STRUCTURE IN PYTHON

Act as a Python data structures expert.
Help me implement a Singly Linked List in Python.
Requirements:

- Create a Node class
- Create a LinkedList class
- Implement insertion at end
- Implement traversal/display method
- Add detailed comments explaining logic
- Provide sample input and output
- Explain how nodes are connected internally

task3.py

✓ Implementing a linked list

I'll open task3.py (it's currently empty) and then fill it with a Node and LinkedList implementation plus example usage and explanations.

✓ Reviewed and updated task3.py

I've implemented your singly linked list and wired in example usage.

- Added `Node` and `LinkedList` classes in AI-ASSISTED_CODING/task3.py.
- `Node` holds `data` and a `next` reference that points to the next node (or `None` for the last node).
- `LinkedList`:
  - Maintains `head` (first node).
  - `insert_at_end(data)`: creates a new node and links

task5.py task3.py

Describe what to build next

GPT-5.1

- Added Node and LinkedList classes.
- Node holds data and a next reference that points to the next node (or None for the last node).
- LinkedList:

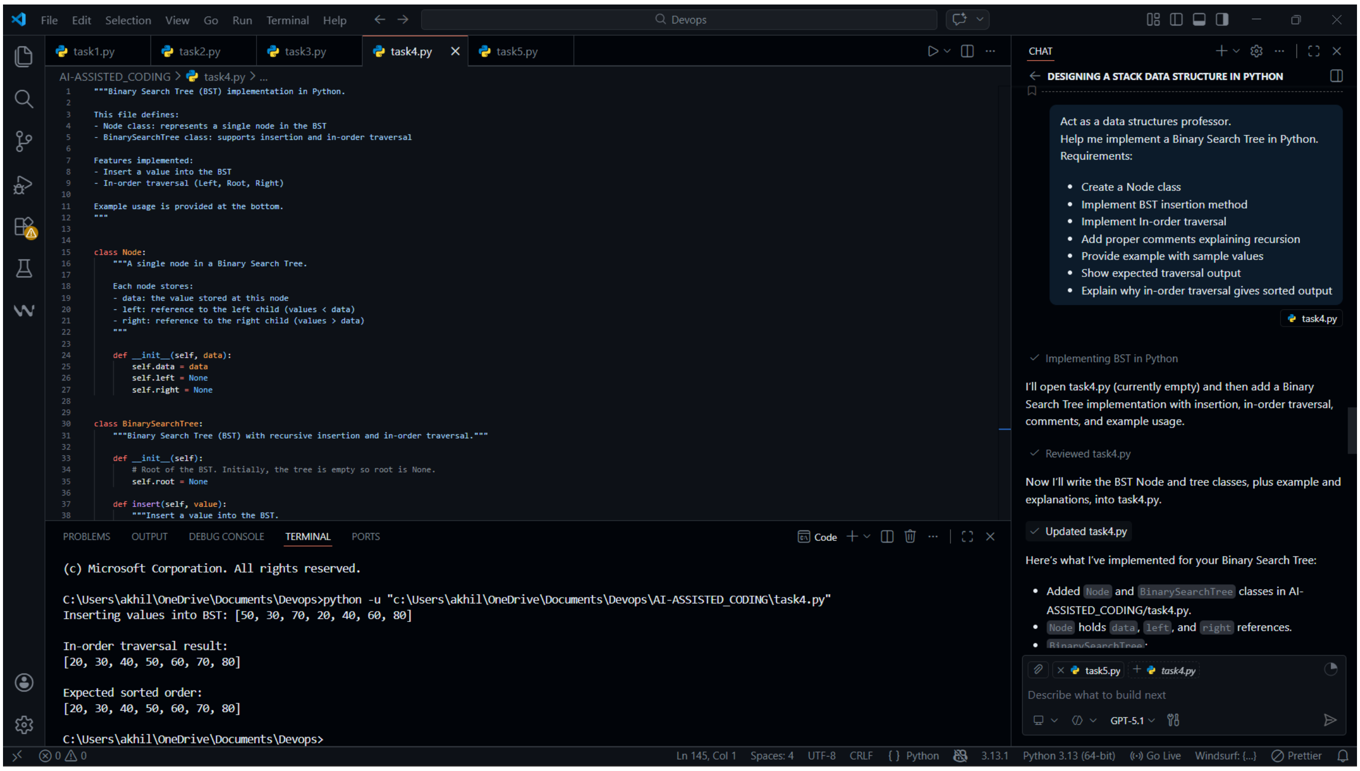
- Maintains head (first node).
- insert_at_end(data): creates a new node and links it at the end by walking from head to the last node and updating its next.
- traverse(): walks from head via next, collects data values into a Python list, and returns it.
- display(): prints the list in the form 10 -> 20 -> 30 -> None.

Task Description -4 – (Binary Search Tree Operations)

• Task: Implement a Binary Search Tree with AI support focusing on insertion and traversal.

Expected Output:

• BST program with correct node insertion and in-order traversal output.



```python
"""Binary Search Tree (BST) implementation in Python.

This file defines:
- Node class: represents a single node in the BST
- BinarySearchTree class: supports insertion and in-order traversal

Features implemented:
- Insert a value into the BST
- In-order traversal (Left, Root, Right)

Example usage is provided at the bottom.
"""


class Node:
    """A single node in a Binary Search Tree.

    Each node stores:
    - data: the value stored at this node
    - left: reference to the left child (values < data)
    - right: reference to the right child (values > data)
    """

    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None


class BinarySearchTree:
    """Binary Search Tree (BST) with recursive insertion and in-order traversal."""

    def __init__(self):
        # Root of the BST. Initially, the tree is empty so root is None.
        self.root = None

    def insert(self, value):
        """Insert a value into the BST.
```

```
(c) Microsoft Corporation. All rights reserved.

C:\Users\akhil\OneDrive\Documents\Devops>python -u "c:\Users\akhil\OneDrive\Documents\Devops\AI-ASSISTED_CODING\task4.py"
Inserting values into BST: [50, 30, 70, 20, 40, 60, 80]

In-order traversal result:
[20, 30, 40, 50, 60, 70, 80]

Expected sorted order:
[20, 30, 40, 50, 60, 70, 80]

C:\Users\akhil\OneDrive\Documents\Devops>
```

Act as a data structures professor.
Help me implement a Binary Search Tree in Python.
Requirements:

- Create a Node class
- Implement BST insertion method
- Implement In-order traversal
- Add proper comments explaining recursion
- Provide example with sample values
- Show expected traversal output
- Explain why in-order traversal gives sorted output

task4.py

✓ Implementing BST in Python

I'll open task4.py (currently empty) and then add a Binary Search Tree implementation with insertion, in-order traversal, comments, and example usage.

✓ Reviewed task4.py

Now I'll write the BST Node and tree classes, plus example and explanations, into task4.py.

✓ Updated task4.py

Here's what I've implemented for your Binary Search Tree:

- Added `Node` and `BinarySearchTree` classes in AI-ASSISTED_CODING/task4.py.
- `Node` holds `data`, `left`, and `right` references.
- `BinarySearchTree`:

Added Node and BinarySearchTree classes in AI-ASSISTED_CODING/task4.py.

Node holds data, left, and right references.

BinarySearchTree:

insert(value): public insert method using _insert_recursive(node, value):

Base case: if node is None, create and return a new Node.

Recursive case: go left if value <node.data, right if value >node.data, then return node to maintain links.

inorder_traversal(): returns a list of values; uses _inorder_recursive(node, result):

Base case: node is None → return.

Recursive case: traverse left, visit node (append data), traverse right.

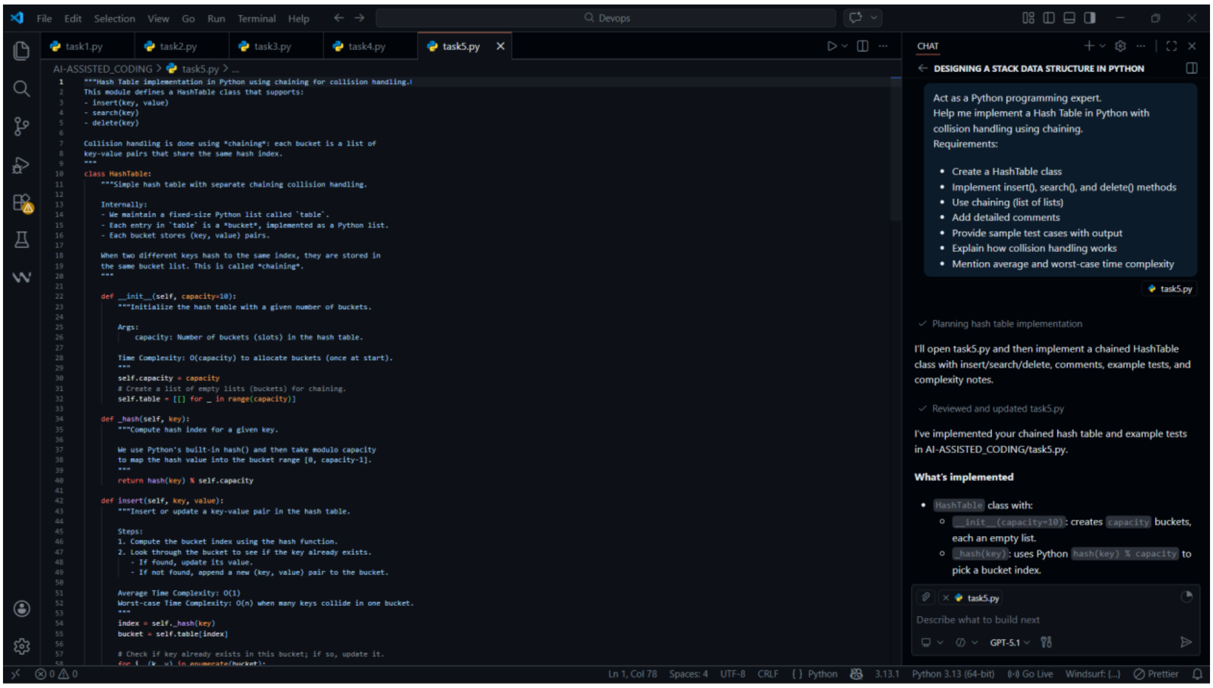
Comments in both recursive helpers explain base/recursive cases clearly.

Task Description -5 – (Hash Table Implementation)

• Task: Create a hash table using AI with collision handling

Expected Output:

• Hash table supporting insert, search, and delete using chainingor open



AI-ASSISTED_CODING > task5.py > ...

```python
"""Hash Table implementation in Python using chaining for collision handling.
This module defines a HashTable class that supports:
- insert(key, value)
- search(key)
- delete(key)

Collision handling is done using *chaining*: each bucket is a list of
key-value pairs that share the same hash index.
"""
class HashTable:
    """Simple hash table with separate chaining collision handling.

    Internally:
    - We maintain a fixed-size Python list called `table`.
    - Each entry in `table` is a *bucket*, implemented as a Python list.
    - Each bucket stores (key, value) pairs.

    When two different keys hash to the same index, they are stored in
    the same bucket list. This is called *chaining*.
    """

    def __init__(self, capacity=10):
        """Initialize the hash table with a given number of buckets.

        Args:
            capacity: Number of buckets (slots) in the hash table.

        Time Complexity: O(capacity) to allocate buckets (once at start).
        """
        self.capacity = capacity
        # Create a list of empty lists (buckets) for chaining.
        self.table = [[] for _ in range(capacity)]

    def _hash(self, key):
        """Compute hash index for a given key.

        We use Python's built-in hash() and then take modulo capacity
        to map the hash value into the bucket range [0, capacity-1].
        """
        return hash(key) % self.capacity

    def insert(self, key, value):
        """Insert or update a key-value pair in the hash table.

        Steps:
        1. Compute the bucket index using the hash function.
        2. Look through the bucket to see if the key already exists.
          - If found, update its value.
          - If not found, append a new (key, value) pair to the bucket.

        Average Time Complexity: O(1)
        Worst-case Time Complexity: O(n) when many keys collide in one bucket.
        """
        index = self._hash(key)
        bucket = self.table[index]

        # Check if key already exists in this bucket; if so, update it.
        for i, (k, v) in enumerate(bucket):
```

CHAT

DESIGNING A STACK DATA STRUCTURE IN PYTHON

Act as a Python programming expert.
Help me implement a Hash Table in Python with collision handling using chaining.
Requirements:

- Create a HashTable class
- Implement insert(), search(), and delete() methods
- Use chaining (list of lists)
- Add detailed comments
- Provide sample test cases with output
- Explain how collision handling works
- Mention average and worst-case time complexity

task5.py

✓ Planning hash table implementation

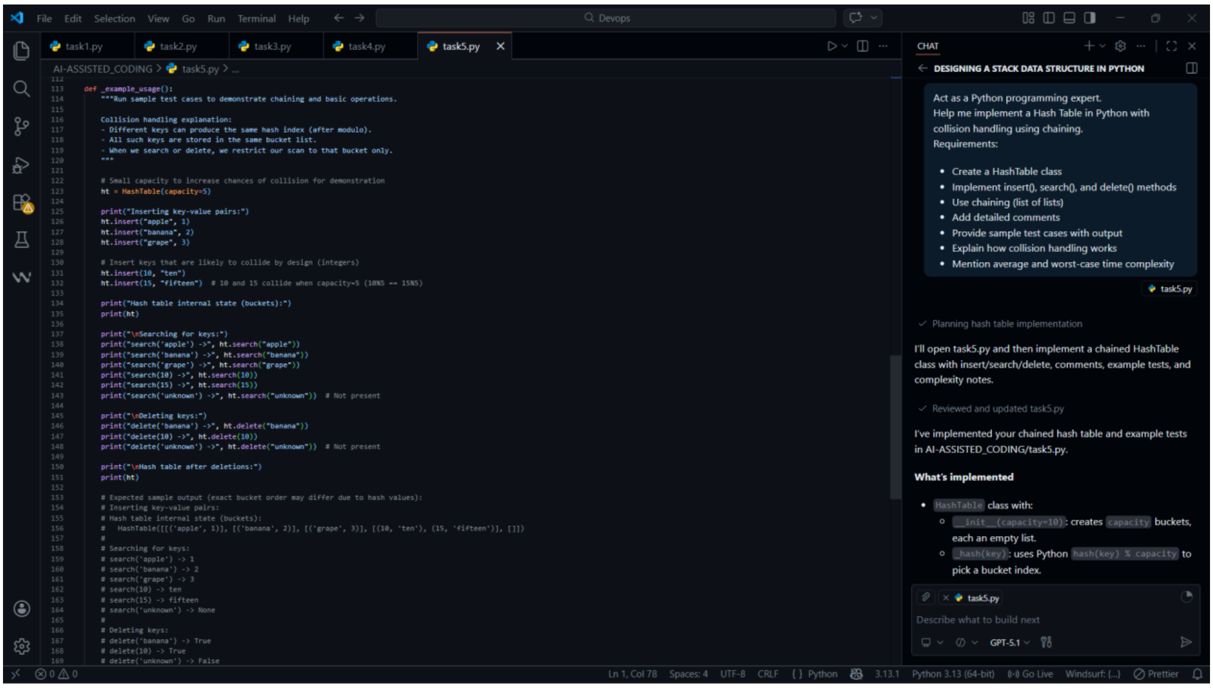
I'll open task5.py and then implement a chained HashTable class with insert/search/delete, comments, example tests, and complexity notes.

✓ Reviewed and updated task5.py

I've implemented your chained hash table and example tests in AI-ASSISTED_CODING/task5.py.

What's implemented

- `HashTable` class with:
    - `__init__(capacity=10)`: creates `capacity` buckets, each an empty list.
    - `_hash(key)`: uses Python `hash(key) % capacity` to pick a bucket index.


File Edit Selection View Go Run Terminal Help
Devops
task1.py
task2.py
task3.py
task4.py
task5.py
AI-ASSISTED_CODING > task5.py > ...
CHAT
Ln 1, Col 78 Spaces: 4 UTF-8 CRLF {} Python 3.13.1 Python 3.13 (64-bit) Go Live Windsurf: {...} Prettier

```python
def _example_usage():
    """Run sample test cases to demonstrate chaining and basic operations.

    Collision handling explanation:
    - Different keys can produce the same hash index (after modulo).
    - All such keys are stored in the same bucket list.
    - When we search or delete, we restrict our scan to that bucket only.
    """

    # Small capacity to increase chances of collision for demonstration
    ht = HashTable(capacity=5)

    print("Inserting key-value pairs:")
    ht.insert("apple", 1)
    ht.insert("banana", 2)
    ht.insert("grape", 3)

    # Insert keys that are likely to collide by design (integers)
    ht.insert(10, "ten")
    ht.insert(15, "fifteen")  # 10 and 15 collide when capacity=5 (10%5 == 15%5)

    print("Hash table internal state (buckets):")
    print(ht)

    print("\nSearching for keys:")
    print("search('apple') ->", ht.search("apple"))
    print("search('banana') ->", ht.search("banana"))
    print("search('grape') ->", ht.search("grape"))
    print("search(10) ->", ht.search(10))
    print("search(15) ->", ht.search(15))
    print("search('unknown') ->", ht.search("unknown"))  # Not present

    print("\nDeleting keys:")
    print("delete('banana') ->", ht.delete("banana"))
    print("delete(10) ->", ht.delete(10))
    print("delete('unknown') ->", ht.delete("unknown"))  # Not present

    print("\nHash table after deletions:")
    print(ht)

    # Expected sample output (exact bucket order may differ due to hash values):
    # Inserting key-value pairs:
    # Hash table internal state (buckets):
    #   HashTable([[('apple', 1)], [('banana', 2)], [('grape', 3)], [(10, 'ten'), (15, 'fifteen')], []])
    #
    # Searching for keys:
    # search('apple') -> 1
    # search('banana') -> 2
    # search('grape') -> 3
    # search(10) -> ten
    # search(15) -> fifteen
    # search('unknown') -> None
    #
    # Deleting keys:
    # delete('banana') -> True
    # delete(10) -> True
    # delete('unknown') -> False
```

DESIGNING A STACK DATA STRUCTURE IN PYTHON

Act as a Python programming expert.
Help me implement a Hash Table in Python with collision handling using chaining.
Requirements:

- Create a HashTable class
- Implement insert(), search(), and delete() methods
- Use chaining (list of lists)
- Add detailed comments
- Provide sample test cases with output
- Explain how collision handling works
- Mention average and worst-case time complexity

task5.py

✓ Planning hash table implementation

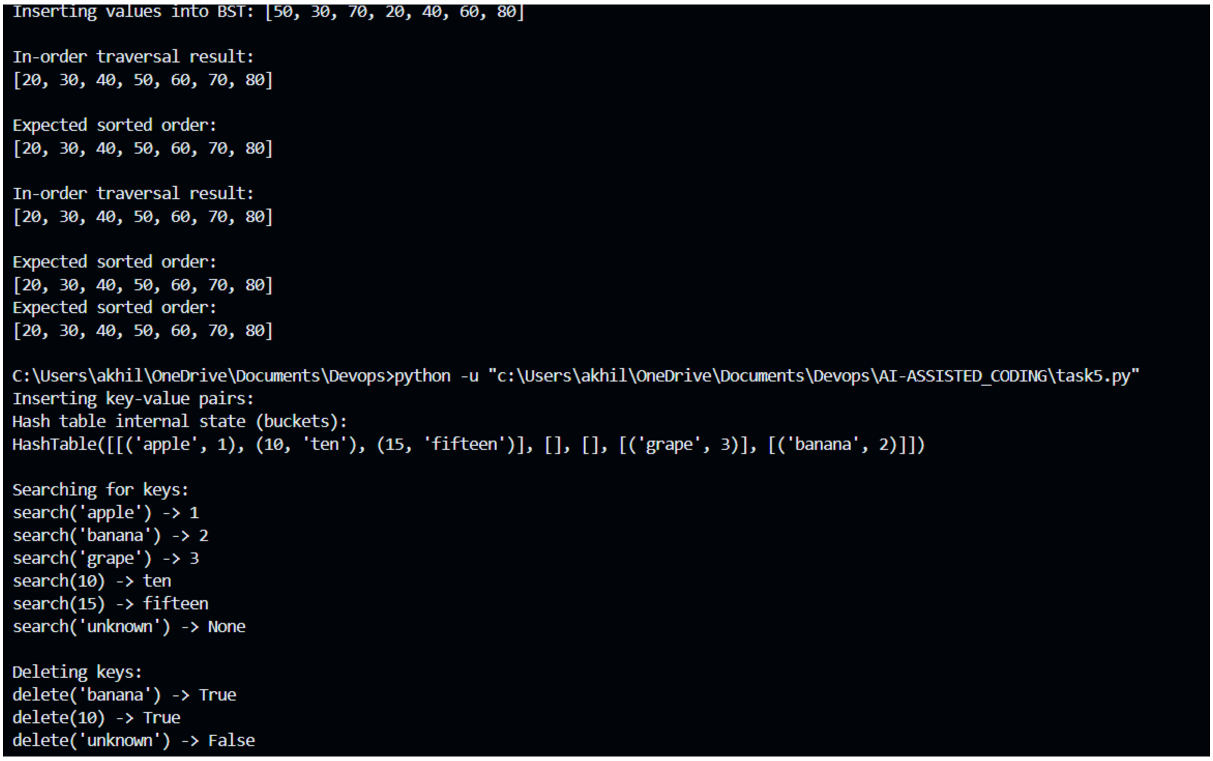
I'll open task5.py and then implement a chained HashTable class with insert/search/delete, comments, example tests, and complexity notes.

✓ Reviewed and updated task5.py

I've implemented your chained hash table and example tests in AI-ASSISTED_CODING/task5.py.

What's implemented

- `HashTable` class with:
    - `__init__(capacity=10)`: creates `capacity` buckets, each an empty list.
    - `_hash(key)`: uses Python `hash(key) % capacity` to pick a bucket index.



```
Inserting values into BST: [50, 30, 70, 20, 40, 60, 80]

In-order traversal result:
[20, 30, 40, 50, 60, 70, 80]

Expected sorted order:
[20, 30, 40, 50, 60, 70, 80]

In-order traversal result:
[20, 30, 40, 50, 60, 70, 80]

Expected sorted order:
[20, 30, 40, 50, 60, 70, 80]
Expected sorted order:
[20, 30, 40, 50, 60, 70, 80]

C:\Users\akhil\OneDrive\Documents\Devops>python -u "c:\Users\akhil\OneDrive\Documents\Devops\AI-ASSISTED_CODING\task5.py"
Inserting key-value pairs:
Hash table internal state (buckets):
HashTable([[('apple', 1), (10, 'ten'), (15, 'fifteen')], [], [], [('grape', 3)], [('banana', 2)]])

Searching for keys:
search('apple') -> 1
search('banana') -> 2
search('grape') -> 3
search(10) -> ten
search(15) -> fifteen
search('unknown') -> None

Deleting keys:
delete('banana') -> True
delete(10) -> True
delete('unknown') -> False
```

- HashTable class with:
    - __init__(capacity=10): creates capacity buckets, each an empty list.
    - _hash(key): uses Python hash(key) % capacity to pick a bucket index.
    - insert(key, value): updates existing key or appends (key, value) into the bucket.
    - search(key): scans the bucket for key, returns the value or None.

  - delete(key): removes (key, value) from the bucket, returns True/False.
- Chaining:
  - self.table is a list of lists (buckets).
  - Each bucket stores multiple (key, value) pairs that share the same index → this is collision handling by chaining.