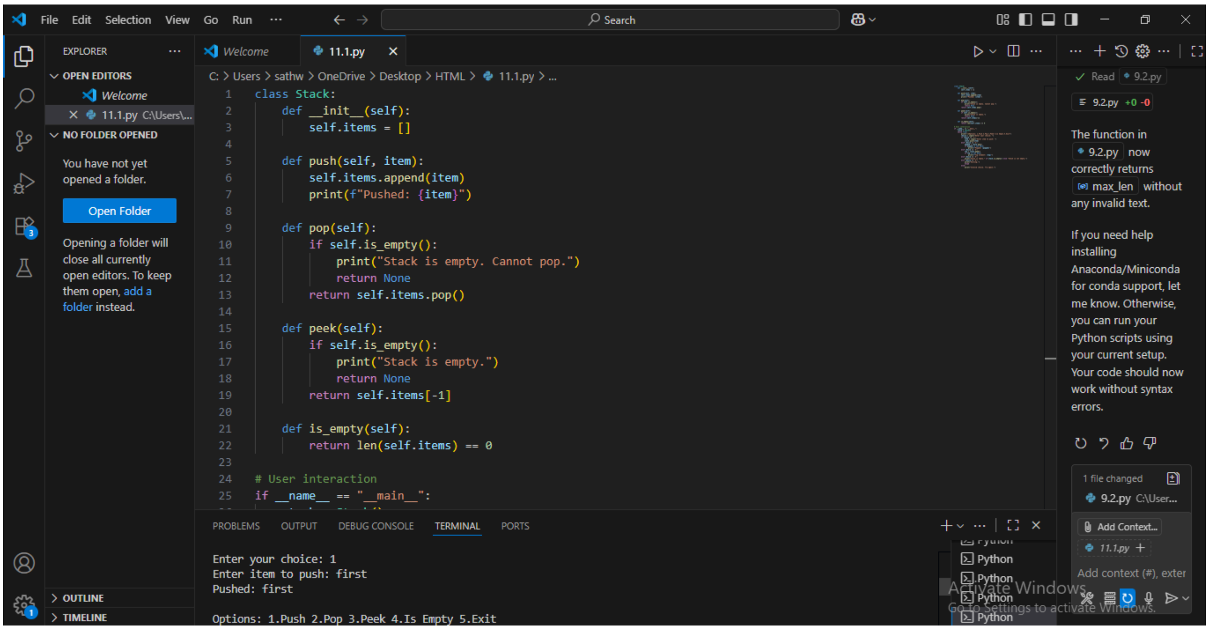
Assignment-11.2

2403a52060

BATCH-03

M.SINDHUJA

TASK-01



You have not yet opened a folder.

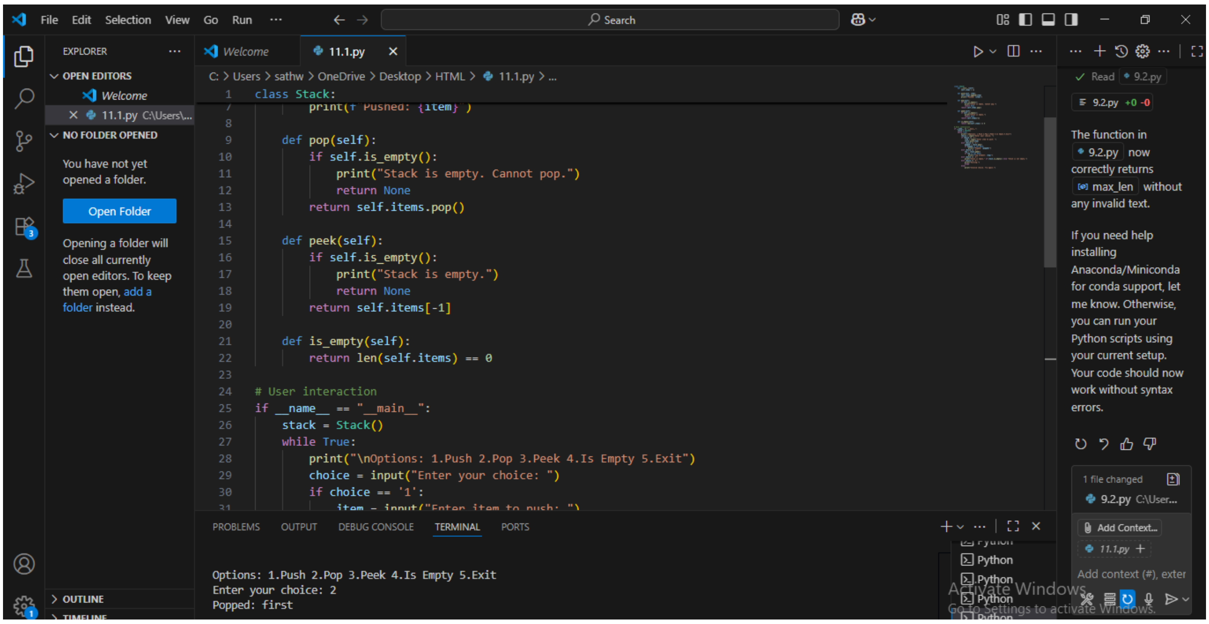
Opening a folder will close all currently open editors. To keep them open, add a folder instead.

```python
class Stack:
    def __init__(self):
        self.items = []

    def push(self, item):
        self.items.append(item)
        print(f"Pushed: {item}")

    def pop(self):
        if self.is_empty():
            print("Stack is empty. Cannot pop.")
            return None
        return self.items.pop()

    def peek(self):
        if self.is_empty():
            print("Stack is empty.")
            return None
        return self.items[-1]

    def is_empty(self):
        return len(self.items) == 0

# User interaction
if __name__ == "__main__":
```

```
Enter your choice: 1
Enter item to push: first
Pushed: first

Options: 1.Push 2.Pop 3.Peek 4.Is Empty 5.Exit
```

The function in 9.2.py now correctly returns max_len without any invalid text.

If you need help installing Anaconda/Miniconda for conda support, let me know. Otherwise, you can run your Python scripts using your current setup. Your code should now work without syntax errors.



You have not yet opened a folder.

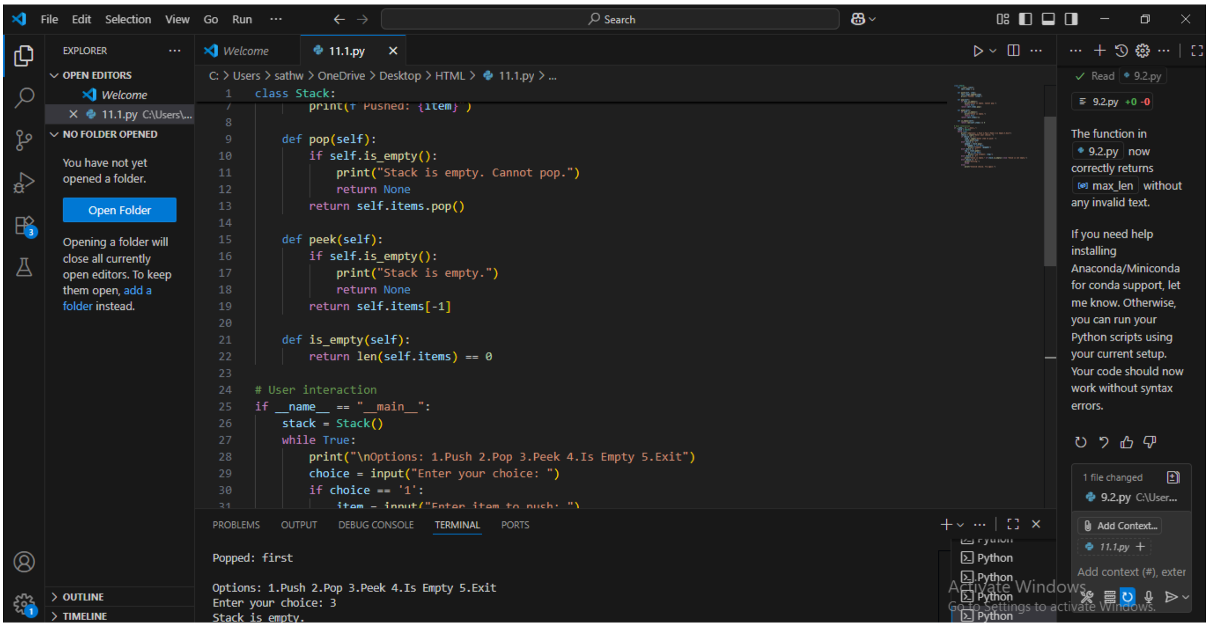
Opening a folder will close all currently open editors. To keep them open, add a folder instead.

```
class Stack:
            print(f"Pushed: {item}")

    def pop(self):
        if self.is_empty():
            print("Stack is empty. Cannot pop.")
            return None
        return self.items.pop()

    def peek(self):
        if self.is_empty():
            print("Stack is empty.")
            return None
        return self.items[-1]

    def is_empty(self):
        return len(self.items) == 0

# User interaction
if __name__ == "__main__":
    stack = Stack()
    while True:
        print("\nOptions: 1.Push 2.Pop 3.Peek 4.Is Empty 5.Exit")
        choice = input("Enter your choice: ")
        if choice == '1':
            item = input("Enter item to push: ")
```

```
Options: 1.Push 2.Pop 3.Peek 4.Is Empty 5.Exit
Enter your choice: 2
Popped: first
```

The function in 9.2.py now correctly returns max_len without any invalid text.

If you need help installing Anaconda/Miniconda for conda support, let me know. Otherwise, you can run your Python scripts using your current setup. Your code should now work without syntax errors.



You have not yet opened a folder.

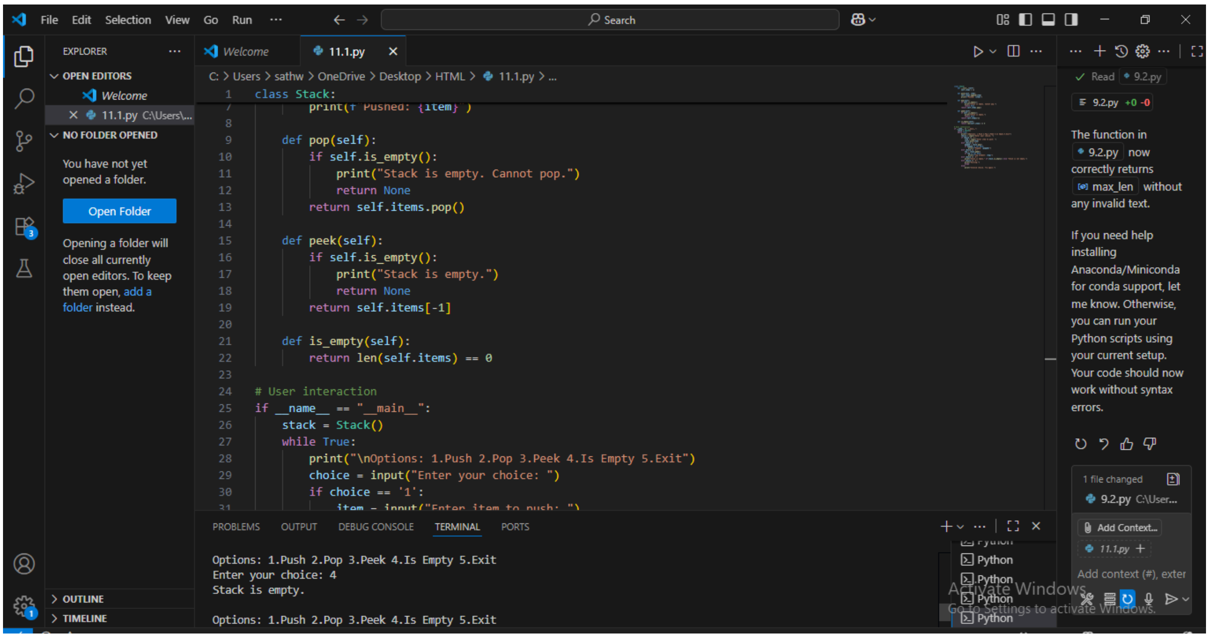
Opening a folder will close all currently open editors. To keep them open, add a folder instead.

```
class Stack:
            print(f"Pushed: {item}")

    def pop(self):
        if self.is_empty():
            print("Stack is empty. Cannot pop.")
            return None
        return self.items.pop()

    def peek(self):
        if self.is_empty():
            print("Stack is empty.")
            return None
        return self.items[-1]

    def is_empty(self):
        return len(self.items) == 0

# User interaction
if __name__ == "__main__":
    stack = Stack()
    while True:
        print("\nOptions: 1.Push 2.Pop 3.Peek 4.Is Empty 5.Exit")
        choice = input("Enter your choice: ")
        if choice == '1':
            item = input("Enter item to push: ")
```

```
Popped: first

Options: 1.Push 2.Pop 3.Peek 4.Is Empty 5.Exit
Enter your choice: 3
Stack is empty.
```

The function in 9.2.py now correctly returns max_len without any invalid text.

If you need help installing Anaconda/Miniconda for conda support, let me know. Otherwise, you can run your Python scripts using your current setup. Your code should now work without syntax errors.



```
class Stack:
            print(f"Pushed: {item}")

    def pop(self):
        if self.is_empty():
            print("Stack is empty. Cannot pop.")
            return None
        return self.items.pop()

    def peek(self):
        if self.is_empty():
            print("Stack is empty.")
            return None
        return self.items[-1]

    def is_empty(self):
        return len(self.items) == 0

# User interaction
if __name__ == "__main__":
    stack = Stack()
    while True:
        print("\nOptions: 1.Push 2.Pop 3.Peek 4.Is Empty 5.Exit")
        choice = input("Enter your choice: ")
        if choice == '1':
            item = input("Enter item to push: ")
```

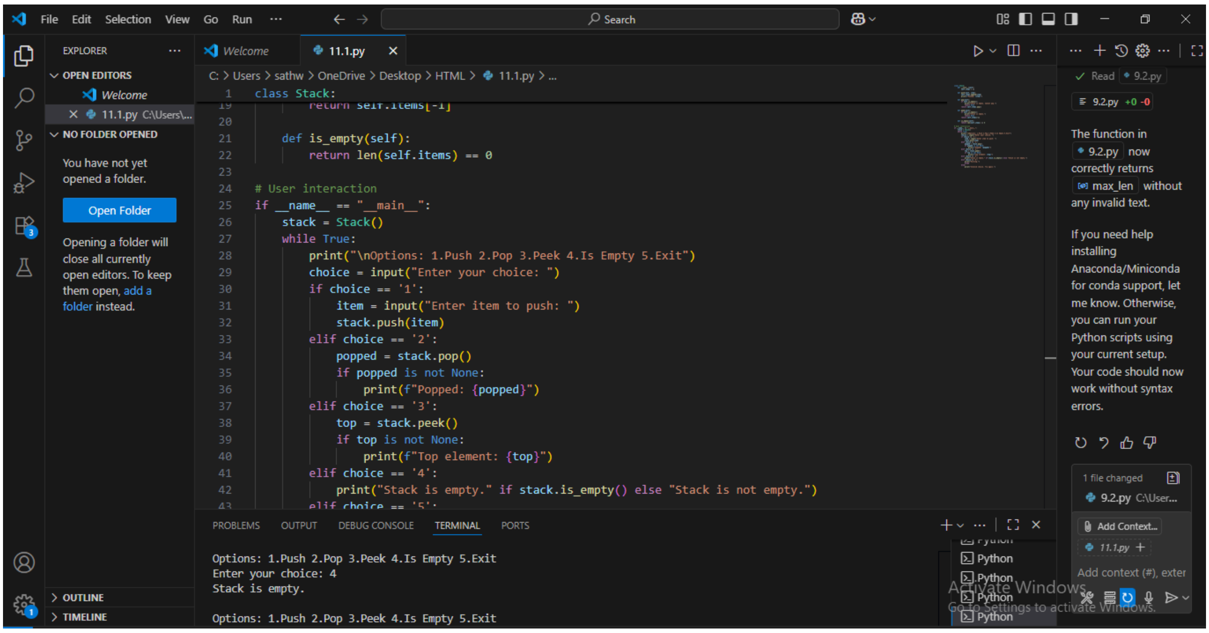
```
Options: 1.Push 2.Pop 3.Peek 4.Is Empty 5.Exit
Enter your choice: 4
Stack is empty.

Options: 1.Push 2.Pop 3.Peek 4.Is Empty 5.Exit
```

You have not yet opened a folder.

Opening a folder will close all currently open editors. To keep them open, add a folder instead.

The function in 9.2.py now correctly returns max_len without any invalid text.

If you need help installing Anaconda/Miniconda for conda support, let me know. Otherwise, you can run your Python scripts using your current setup. Your code should now work without syntax errors.



```
class Stack:
        return self.items[-1]

    def is_empty(self):
        return len(self.items) == 0

# User interaction
if __name__ == "__main__":
    stack = Stack()
    while True:
        print("\nOptions: 1.Push 2.Pop 3.Peek 4.Is Empty 5.Exit")
        choice = input("Enter your choice: ")
        if choice == '1':
            item = input("Enter item to push: ")
            stack.push(item)
        elif choice == '2':
            popped = stack.pop()
            if popped is not None:
                print(f"Popped: {popped}")
        elif choice == '3':
            top = stack.peek()
            if top is not None:
                print(f"Top element: {top}")
        elif choice == '4':
            print("Stack is empty." if stack.is_empty() else "Stack is not empty.")
        elif choice == '5':
```

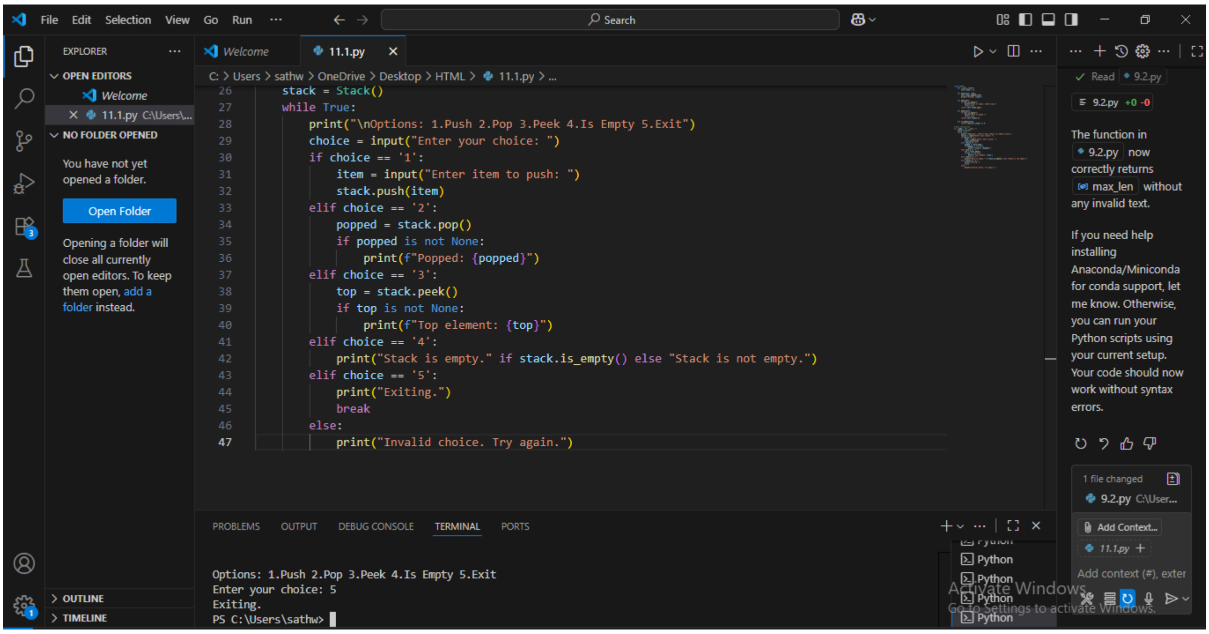
```
Options: 1.Push 2.Pop 3.Peek 4.Is Empty 5.Exit
Enter your choice: 4
Stack is empty.

Options: 1.Push 2.Pop 3.Peek 4.Is Empty 5.Exit
```

You have not yet opened a folder.

Opening a folder will close all currently open editors. To keep them open, add a folder instead.

The function in 9.2.py now correctly returns max_len without any invalid text.

If you need help installing Anaconda/Miniconda for conda support, let me know. Otherwise, you can run your Python scripts using your current setup. Your code should now work without syntax errors.



You have not yet opened a folder.

Opening a folder will close all currently open editors. To keep them open, add a folder instead.

```python
    stack = Stack()
    while True:
        print("\nOptions: 1.Push 2.Pop 3.Peek 4.Is Empty 5.Exit")
        choice = input("Enter your choice: ")
        if choice == '1':
            item = input("Enter item to push: ")
            stack.push(item)
        elif choice == '2':
            popped = stack.pop()
            if popped is not None:
                print(f"Popped: {popped}")
        elif choice == '3':
            top = stack.peek()
            if top is not None:
                print(f"Top element: {top}")
        elif choice == '4':
            print("Stack is empty." if stack.is_empty() else "Stack is not empty.")
        elif choice == '5':
            print("Exiting.")
            break
        else:
            print("Invalid choice. Try again.")
```

```
Options: 1.Push 2.Pop 3.Peek 4.Is Empty 5.Exit
Enter your choice: 5
Exiting.
PS C:\Users\sathw> 
```

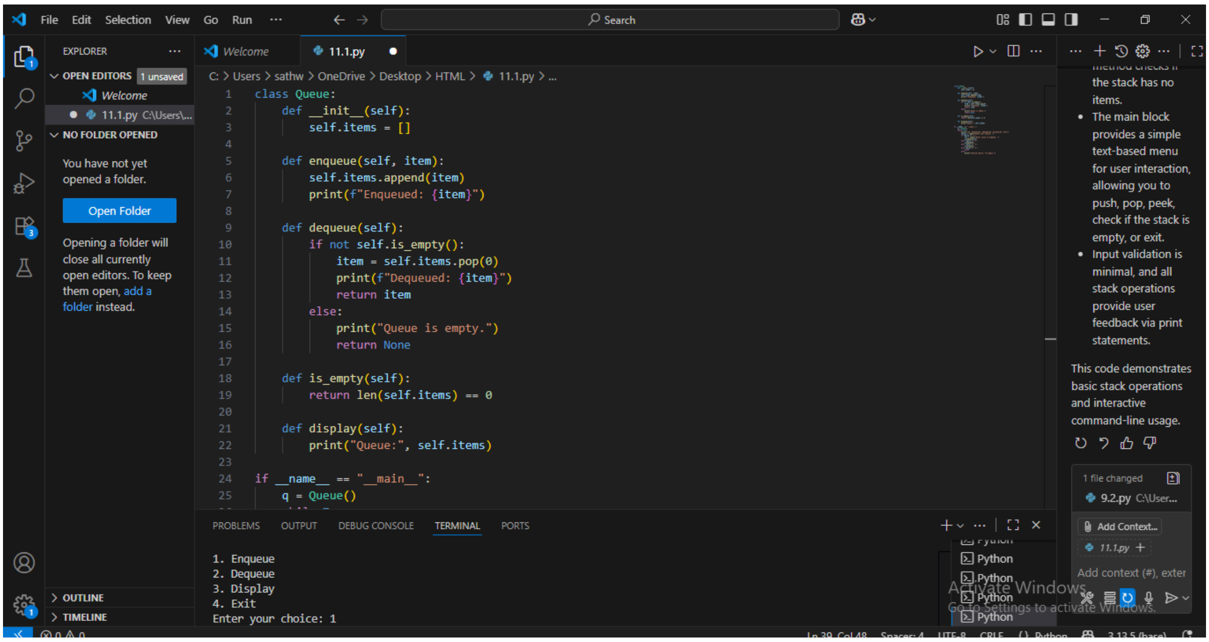
The function in 9.2.py now correctly returns max_len without any invalid text.

If you need help installing Anaconda/Miniconda for conda support, let me know. Otherwise, you can run your Python scripts using your current setup. Your code should now work without syntax errors.

EXPLANATION:

This Python code defines a Stack class and provides a simple command-line interface to interact with a stack data structure. Here's a breakdown:
Stack class:
__init__(self): The constructor initializes an empty list self.items which will be used to store the stack elements.
push(self, item): Adds an item to the top of the stack (the end of the list).
pop(self): Removes and returns the item from the top of the stack. It checks if the stack is empty before attempting to pop.
peek(self): Returns the item at the top of the stack without removing it. It also checks if the stack is empty.
is_empty(self): Returns True if the stack is empty, False otherwise.
User interaction (if **__name__** == "**__main__**":):
This block runs when the script is executed directly.
It creates a Stack object.
It enters a loop that presents the user with options to perform stack operations (push, pop, peek, check if empty, or exit).
Based on the user's input, it calls the corresponding Stack methods.

TASK-02:



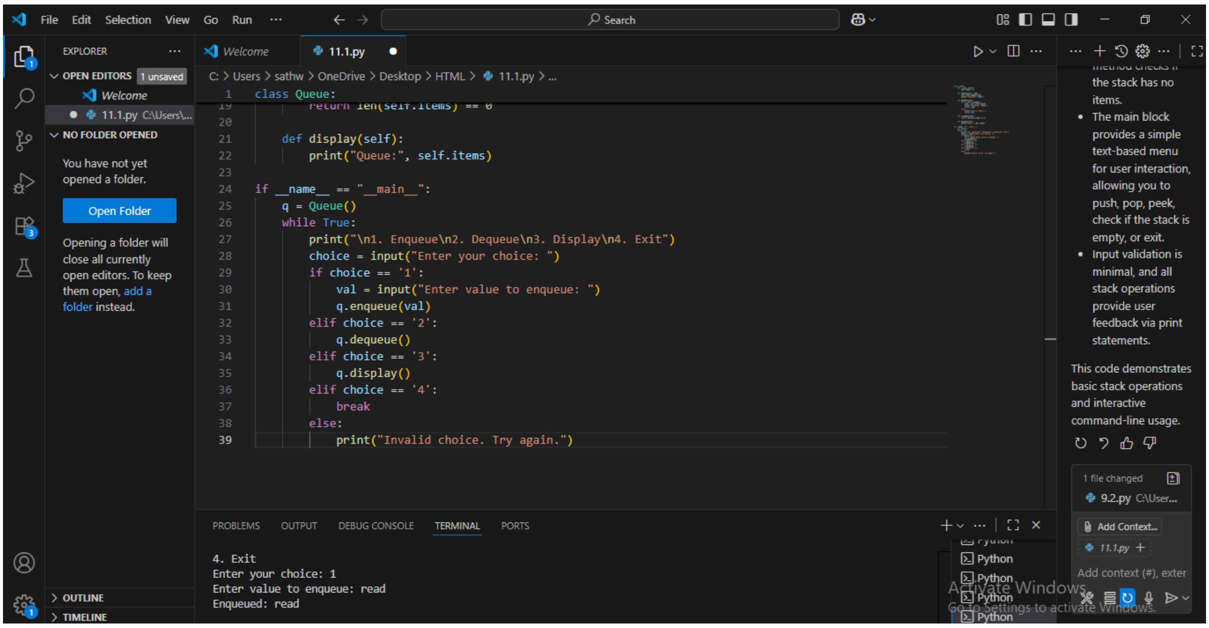
```python
class Queue:
    def __init__(self):
        self.items = []

    def enqueue(self, item):
        self.items.append(item)
        print(f"Enqueued: {item}")

    def dequeue(self):
        if not self.is_empty():
            item = self.items.pop(0)
            print(f"Dequeued: {item}")
            return item
        else:
            print("Queue is empty.")
            return None

    def is_empty(self):
        return len(self.items) == 0

    def display(self):
        print("Queue:", self.items)

if __name__ == "__main__":
    q = Queue()
```

```
1. Enqueue
2. Dequeue
3. Display
4. Exit
Enter your choice: 1
```

the stack has no items.

- The main block provides a simple text-based menu for user interaction, allowing you to push, pop, peek, check if the stack is empty, or exit.
- Input validation is minimal, and all stack operations provide user feedback via print statements.

This code demonstrates basic stack operations and interactive command-line usage.



```python
        return len(self.items) == 0

    def display(self):
        print("Queue:", self.items)

if __name__ == "__main__":
    q = Queue()
    while True:
        print("\n1. Enqueue\n2. Dequeue\n3. Display\n4. Exit")
        choice = input("Enter your choice: ")
        if choice == '1':
            val = input("Enter value to enqueue: ")
            q.enqueue(val)
        elif choice == '2':
            q.dequeue()
        elif choice == '3':
            q.display()
        elif choice == '4':
            break
        else:
            print("Invalid choice. Try again.")
```

```
4. Exit
Enter your choice: 1
Enter value to enqueue: read
Enqueued: read
```

the stack has no items.

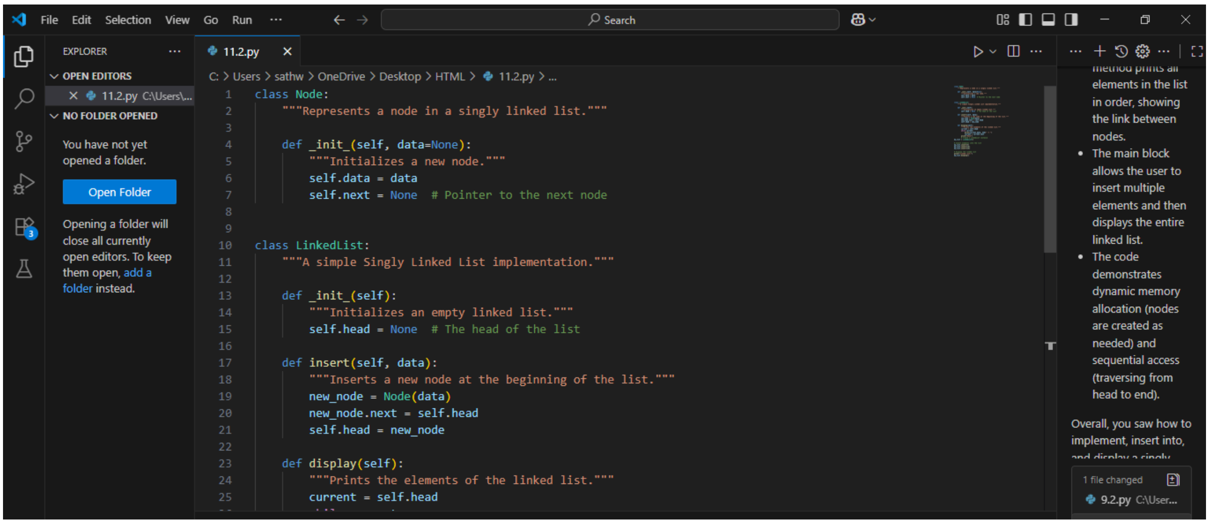
- The main block provides a simple text-based menu for user interaction, allowing you to push, pop, peek, check if the stack is empty, or exit.
- Input validation is minimal, and all stack operations provide user feedback via print statements.

This code demonstrates basic stack operations and interactive command-line usage.

EXPLANATION:

- The Queue class uses a list to store items.
- The enqueue method adds an item to the end of the queue.
- The dequeue method removes and returns the item from the front of the queue, with a message if the queue is empty.
- The `is_empty` method checks if the queue has no items.
- The `display` method prints the current contents of the queue.
- The main block provides a menu-driven interface for the user to enqueue, dequeue, display, or exit.
- Each operation gives feedback to the user, making it easy to understand the queue's state.

TASK-03:



You have not yet opened a folder.

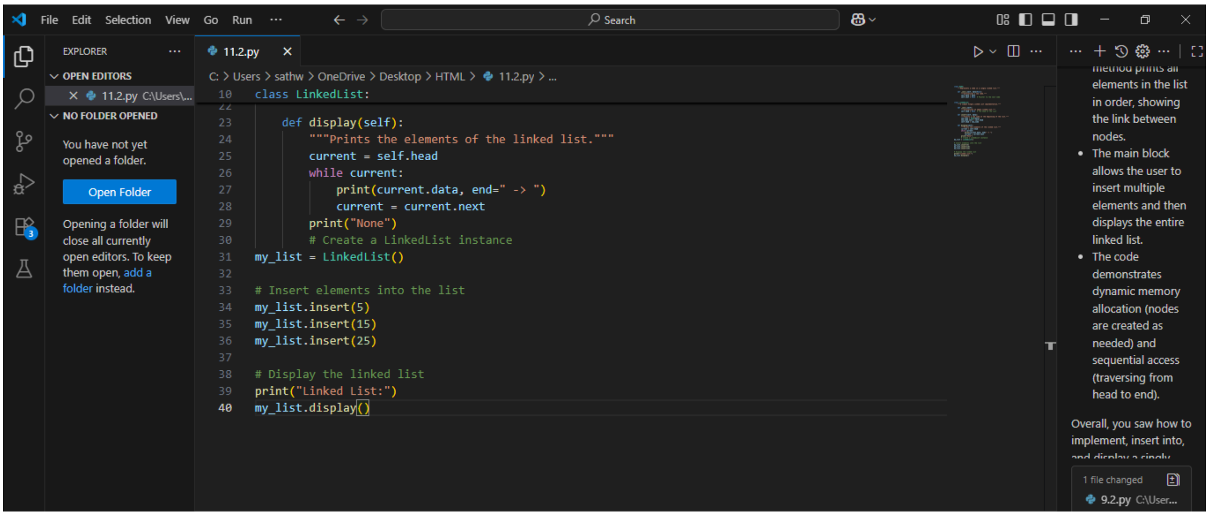
Opening a folder will close all currently open editors. To keep them open, add a folder instead.

```
class Node:
    """Represents a node in a singly linked list."""

    def _init_(self, data=None):
        """Initializes a new node."""
        self.data = data
        self.next = None  # Pointer to the next node


class LinkedList:
    """A simple Singly Linked List implementation."""

    def _init_(self):
        """Initializes an empty linked list."""
        self.head = None  # The head of the list

    def insert(self, data):
        """Inserts a new node at the beginning of the list."""
        new_node = Node(data)
        new_node.next = self.head
        self.head = new_node

    def display(self):
        """Prints the elements of the linked list."""
        current = self.head
```

method prints all elements in the list in order, showing the link between nodes.

- The main block allows the user to insert multiple elements and then displays the entire linked list.
- The code demonstrates dynamic memory allocation (nodes are created as needed) and sequential access (traversing from head to end).

Overall, you saw how to implement, insert into, and display a singly



You have not yet opened a folder.

Opening a folder will close all currently open editors. To keep them open, add a folder instead.

```
class LinkedList:

    def display(self):
        """Prints the elements of the linked list."""
        current = self.head
        while current:
            print(current.data, end=" -> ")
            current = current.next
        print("None")
        # Create a LinkedList instance
my_list = LinkedList()

# Insert elements into the list
my_list.insert(5)
my_list.insert(15)
my_list.insert(25)

# Display the linked list
print("Linked List:")
my_list.display()
```

method prints all elements in the list in order, showing the link between nodes.

- The main block allows the user to insert multiple elements and then displays the entire linked list.
- The code demonstrates dynamic memory allocation (nodes are created as needed) and sequential access (traversing from head to end).

Overall, you saw how to implement, insert into, and display a singly

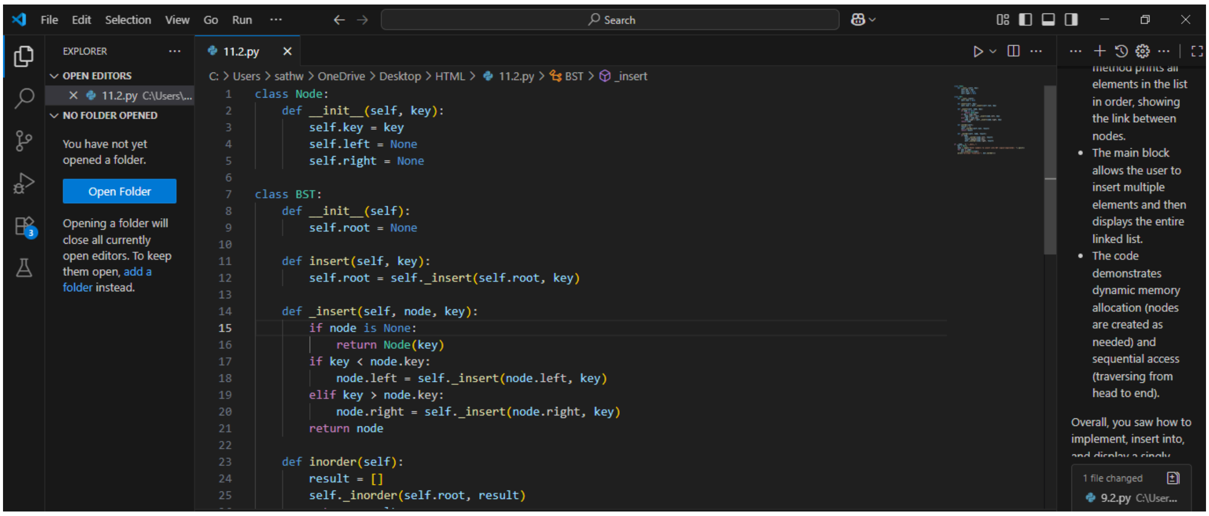
OUTPUT:

Linked List: 25 -> 15 -> 5 -> None

EXPLANATION:

- The `Node` class represents each element in the list, storing data and a reference to the next node.
- The `SinglyLinkedList` class manages the linked list, starting with a `head` node.
- The `insert` method adds new nodes to the end of the list.
- The `display` method prints all elements in the list in order, showing the link between nodes.
- The main block allows the user to insert multiple elements and then displays the entire linked list.
- The code demonstrates dynamic memory allocation (nodes are created as needed) and sequential access (traversing from head to end).
-

TASK-04:



You have not yet opened a folder.

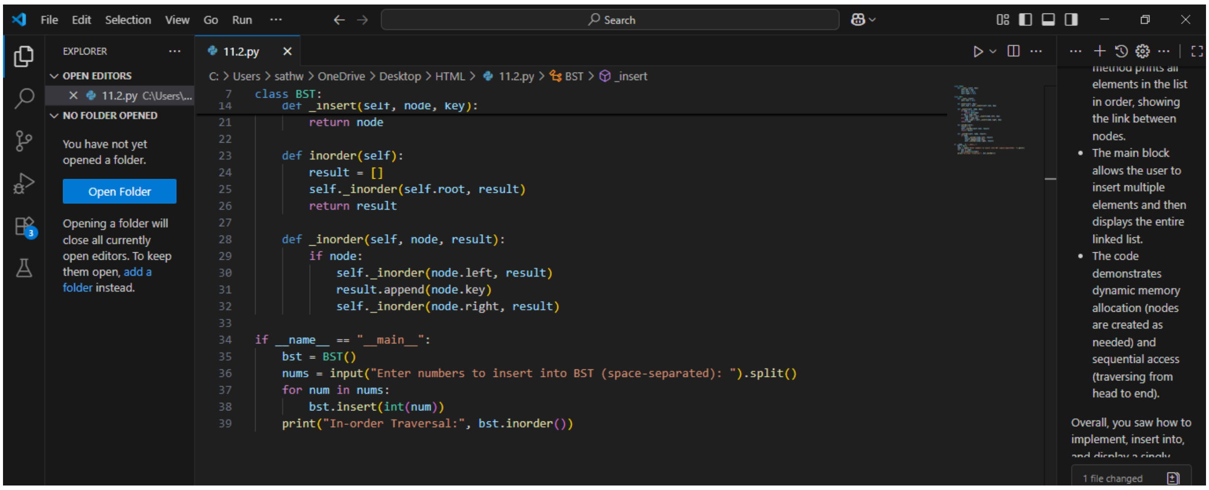
Opening a folder will close all currently open editors. To keep them open, add a folder instead.

```python
class Node:
    def __init__(self, key):
        self.key = key
        self.left = None
        self.right = None

class BST:
    def __init__(self):
        self.root = None

    def insert(self, key):
        self.root = self._insert(self.root, key)

    def _insert(self, node, key):
        if node is None:
            return Node(key)
        if key < node.key:
            node.left = self._insert(node.left, key)
        elif key > node.key:
            node.right = self._insert(node.right, key)
        return node

    def inorder(self):
        result = []
        self._inorder(self.root, result)
```

method prints all elements in the list in order, showing the link between nodes.

- The main block allows the user to insert multiple elements and then displays the entire linked list.
- The code demonstrates dynamic memory allocation (nodes are created as needed) and sequential access (traversing from head to end).

Overall, you saw how to implement, insert into, and display a singly



File Edit Selection View Go Run ···
Search
EXPLORER
OPEN EDITORS
11.2.py C:\Users\...
NO FOLDER OPENED
Open Folder
11.2.py
C: > Users > sathw > OneDrive > Desktop > HTML > 11.2.py > BST > _insert
1 file changed

You have not yet opened a folder.

Opening a folder will close all currently open editors. To keep them open, add a folder instead.

```python
class BST:
    def _insert(self, node, key):
        return node

    def inorder(self):
        result = []
        self._inorder(self.root, result)
        return result

    def _inorder(self, node, result):
        if node:
            self._inorder(node.left, result)
            result.append(node.key)
            self._inorder(node.right, result)

if __name__ == "__main__":
    bst = BST()
    nums = input("Enter numbers to insert into BST (space-separated): ").split()
    for num in nums:
        bst.insert(int(num))
    print("In-order Traversal:", bst.inorder())
```

method prints all elements in the list in order, showing the link between nodes.

- The main block allows the user to insert multiple elements and then displays the entire linked list.
- The code demonstrates dynamic memory allocation (nodes are created as needed) and sequential access (traversing from head to end).

Overall, you saw how to implement, insert into, and display a singly

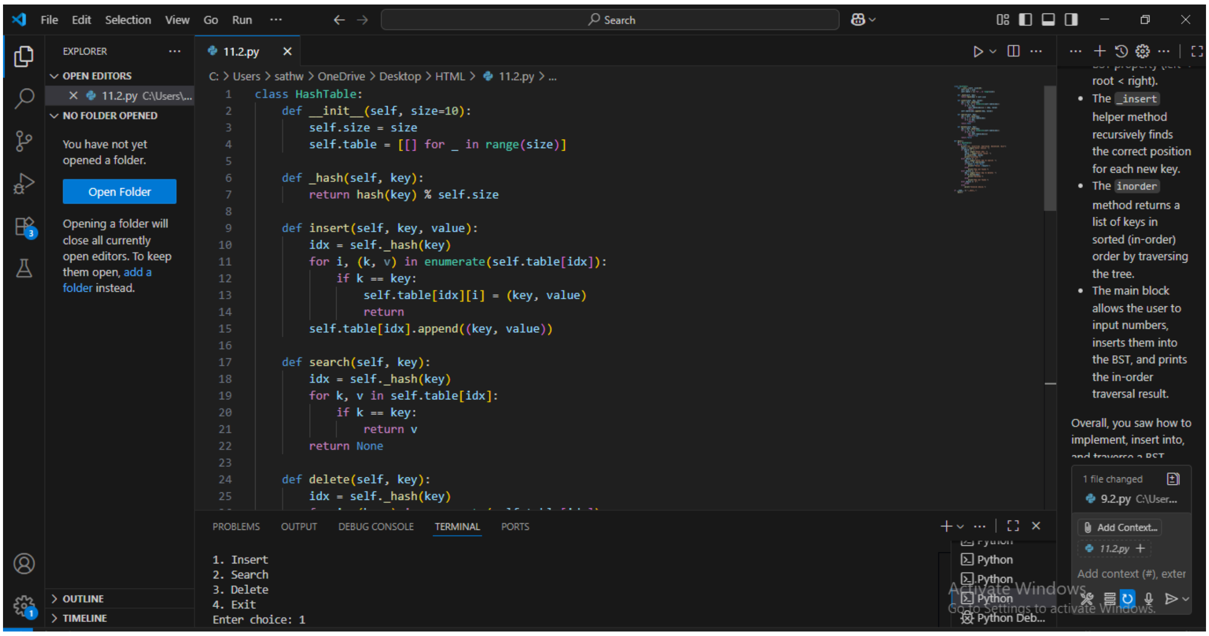
OUTPUT:

```
Enter numbers to insert into BST (space-separated): 5
In-order Traversal: [5]
```

EXPLANATION:

- The Node class represents each node in the tree, storing a key and references to left and right children.
- The BST class manages the tree, starting with a root node.
- The insert method adds new keys to the BST, maintaining the BST property (left < root < right).
- The _insert helper method recursively finds the correct position for each new key.
- The inorder method returns a list of keys in sorted (in-order) order by traversing the tree.
- The main block allows the user to input numbers, inserts them into the BST, and prints the in-order traversal result

TASK-05:



You have not yet opened a folder.

Opening a folder will close all currently open editors. To keep them open, add a folder instead.

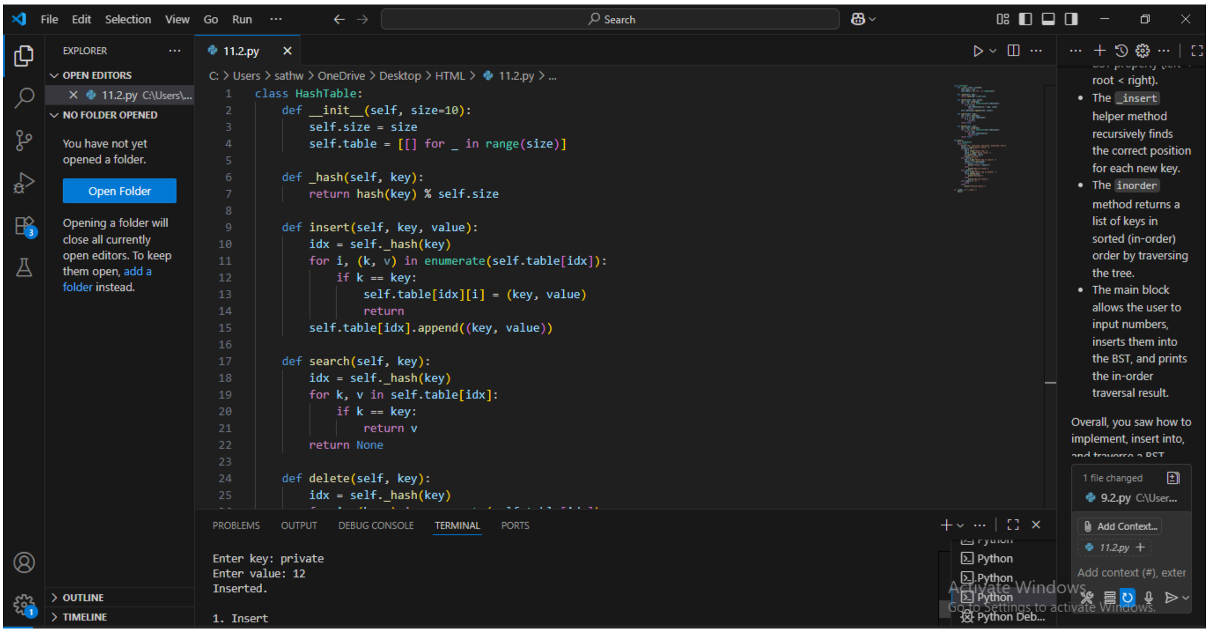
```python
class HashTable:
    def __init__(self, size=10):
        self.size = size
        self.table = [[] for _ in range(size)]

    def _hash(self, key):
        return hash(key) % self.size

    def insert(self, key, value):
        idx = self._hash(key)
        for i, (k, v) in enumerate(self.table[idx]):
            if k == key:
                self.table[idx][i] = (key, value)
                return
        self.table[idx].append((key, value))

    def search(self, key):
        idx = self._hash(key)
        for k, v in self.table[idx]:
            if k == key:
                return v
        return None

    def delete(self, key):
        idx = self._hash(key)
```

```
1. Insert
2. Search
3. Delete
4. Exit
Enter choice: 1
```

root < right).

- The `_insert` helper method recursively finds the correct position for each new key.
- The `inorder` method returns a list of keys in sorted (in-order) order by traversing the tree.
- The main block allows the user to input numbers, inserts them into the BST, and prints the in-order traversal result.

Overall, you saw how to implement, insert into, and traverse a BST



You have not yet opened a folder.

Opening a folder will close all currently open editors. To keep them open, add a folder instead.

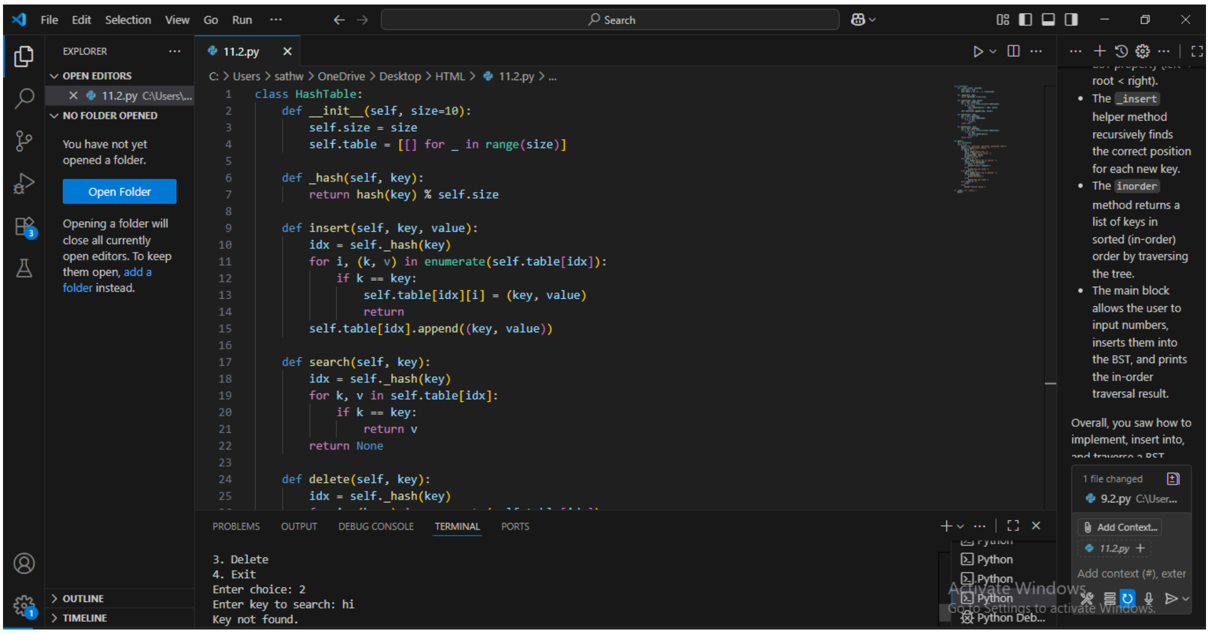
```python
class HashTable:
    def __init__(self, size=10):
        self.size = size
        self.table = [[] for _ in range(size)]

    def _hash(self, key):
        return hash(key) % self.size

    def insert(self, key, value):
        idx = self._hash(key)
        for i, (k, v) in enumerate(self.table[idx]):
            if k == key:
                self.table[idx][i] = (key, value)
                return
        self.table[idx].append((key, value))

    def search(self, key):
        idx = self._hash(key)
        for k, v in self.table[idx]:
            if k == key:
                return v
        return None

    def delete(self, key):
        idx = self._hash(key)
```

```
Enter key: private
Enter value: 12
Inserted.

1. Insert
```

root < right).

- The `_insert` helper method recursively finds the correct position for each new key.
- The `inorder` method returns a list of keys in sorted (in-order) order by traversing the tree.
- The main block allows the user to input numbers, inserts them into the BST, and prints the in-order traversal result.

Overall, you saw how to implement, insert into, and traverse a BST



C: > Users > sathw > OneDrive > Desktop > HTML > 11.2.py > ...

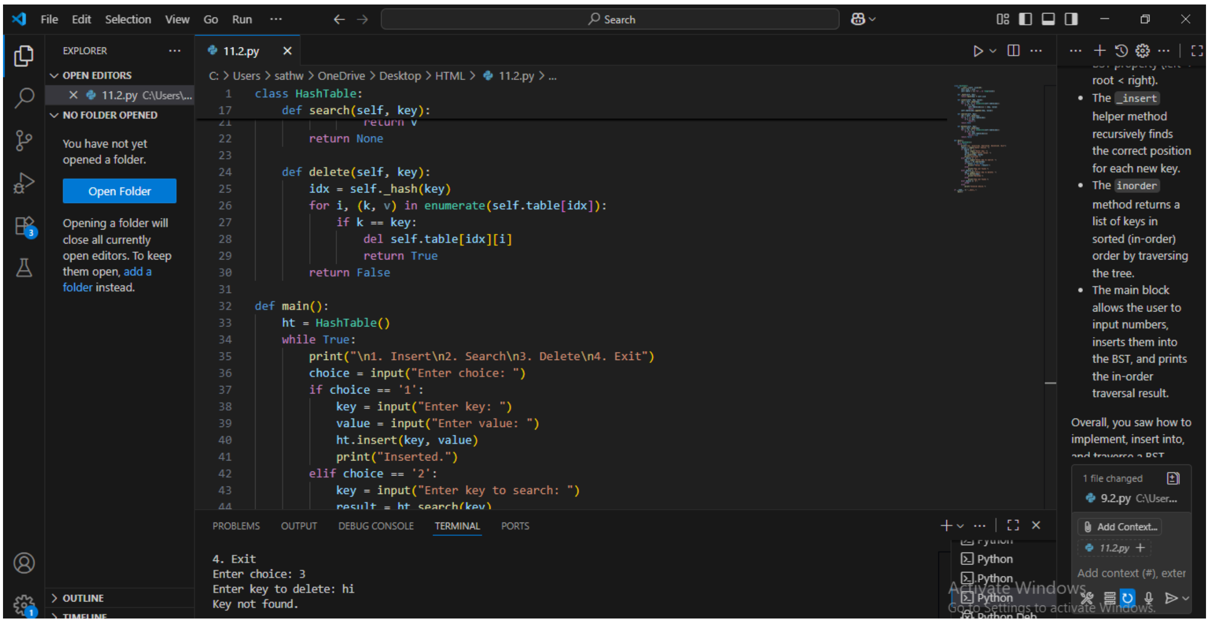
```python
class HashTable:
    def __init__(self, size=10):
        self.size = size
        self.table = [[] for _ in range(size)]

    def _hash(self, key):
        return hash(key) % self.size

    def insert(self, key, value):
        idx = self._hash(key)
        for i, (k, v) in enumerate(self.table[idx]):
            if k == key:
                self.table[idx][i] = (key, value)
                return
        self.table[idx].append((key, value))

    def search(self, key):
        idx = self._hash(key)
        for k, v in self.table[idx]:
            if k == key:
                return v
        return None

    def delete(self, key):
        idx = self._hash(key)
```

```
3. Delete
4. Exit
Enter choice: 2
Enter key to search: hi
Key not found.
```

root < right).

- The `_insert` helper method recursively finds the correct position for each new key.
- The `inorder` method returns a list of keys in sorted (in-order) order by traversing the tree.
- The main block allows the user to input numbers, inserts them into the BST, and prints the in-order traversal result.

Overall, you saw how to implement, insert into,



C: > Users > sathw > OneDrive > Desktop > HTML > 11.2.py > ...

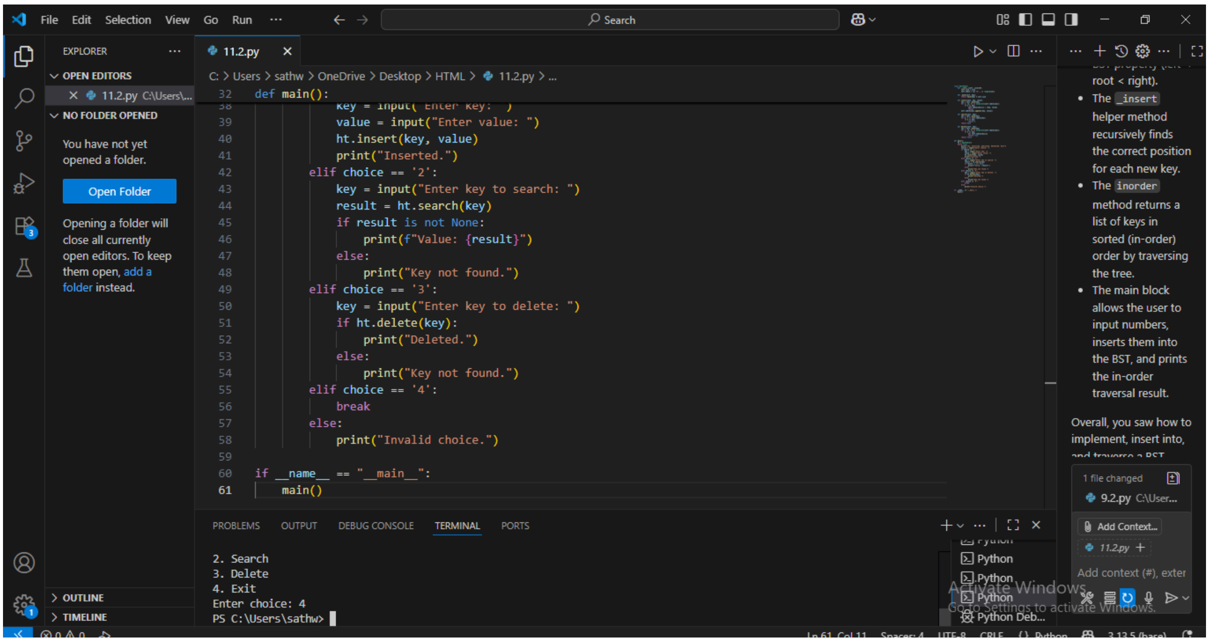
```python
class HashTable:
    def search(self, key):
        return None

    def delete(self, key):
        idx = self._hash(key)
        for i, (k, v) in enumerate(self.table[idx]):
            if k == key:
                del self.table[idx][i]
                return True
        return False

def main():
    ht = HashTable()
    while True:
        print("\n1. Insert\n2. Search\n3. Delete\n4. Exit")
        choice = input("Enter choice: ")
        if choice == '1':
            key = input("Enter key: ")
            value = input("Enter value: ")
            ht.insert(key, value)
            print("Inserted.")
        elif choice == '2':
            key = input("Enter key to search: ")
            result = ht.search(key)
```

```
4. Exit
Enter choice: 3
Enter key to delete: hi
Key not found.
```

root < right).

- The `_insert` helper method recursively finds the correct position for each new key.
- The `inorder` method returns a list of keys in sorted (in-order) order by traversing the tree.
- The main block allows the user to input numbers, inserts them into the BST, and prints the in-order traversal result.

Overall, you saw how to implement, insert into,



You have not yet opened a folder.

Opening a folder will close all currently open editors. To keep them open, add a folder instead.

```
def main():
            key = input("Enter key: ")
            value = input("Enter value: ")
            ht.insert(key, value)
            print("Inserted.")
        elif choice == '2':
            key = input("Enter key to search: ")
            result = ht.search(key)
            if result is not None:
                print(f"Value: {result}")
            else:
                print("Key not found.")
        elif choice == '3':
            key = input("Enter key to delete: ")
            if ht.delete(key):
                print("Deleted.")
            else:
                print("Key not found.")
        elif choice == '4':
            break
        else:
            print("Invalid choice.")

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

```
2. Search
3. Delete
4. Exit
Enter choice: 4
PS C:\Users\sathw>
```

root < right).

- The `_insert` helper method recursively finds the correct position for each new key.
- The `inorder` method returns a list of keys in sorted (in-order) order by traversing the tree.
- The main block allows the user to input numbers, inserts them into the BST, and prints the in-order traversal result.

Overall, you saw how to implement, insert into,

EXPLANATION:

- The HashTable class uses a list of lists (buckets) to handle collisions using chaining.
- The _hash method computes the index for a key using Python's built-in hash() function and modulo operation.
- The insert method adds a key-value pair or updates the value if the key already exists.
- The search method looks for a key and returns its value if found, otherwise returns None.
- The delete method removes a key-value pair if the key exists and returns True; otherwise, it returns False.
- The main function provides a menu-driven interface for inserting, searching, deleting, and exiting.
- User input is used to interact with the hash table, and feedback is given for each operation.