DSA

QUIZ 2: Linked Lists

2.2 : COMPUTER TECHNOLOGY

NJENGA BEATRICE WAMBUI

**Question one**

**Question (One) 1**

Given head, the head of a linked list, determines if the linked list has a cycle in it. There is a cycle in a linked list if there is some node in the list that can be reached again by continuously following the next pointer. Internally, pos is used to denote the index of the node that tail's next pointer is connected to. Note that pos is not passed as a parameter.

Return true *if there is a cycle in the linked list*. Otherwise, return false.

**To solve this problem, you can use Floyd's Tortoise and Hare algorithm, also known as the slow and fast pointers technique. This algorithm involves using two pointers, one moving at twice the speed of the other.**

class ListNode:

def \_\_init\_\_(self, x):

self.val = x

self.next = None

def hasCycle(head):

if not head or not head.next:

return False

slow = head

fast = head.next

while slow != fast:

if not fast or not fast.next:

return False

slow = slow.next

fast = fast.next.next

return True

# Example usage:

# Create the linked list

head = ListNode(3)

head.next = ListNode(2)

head.next.next = ListNode(0)

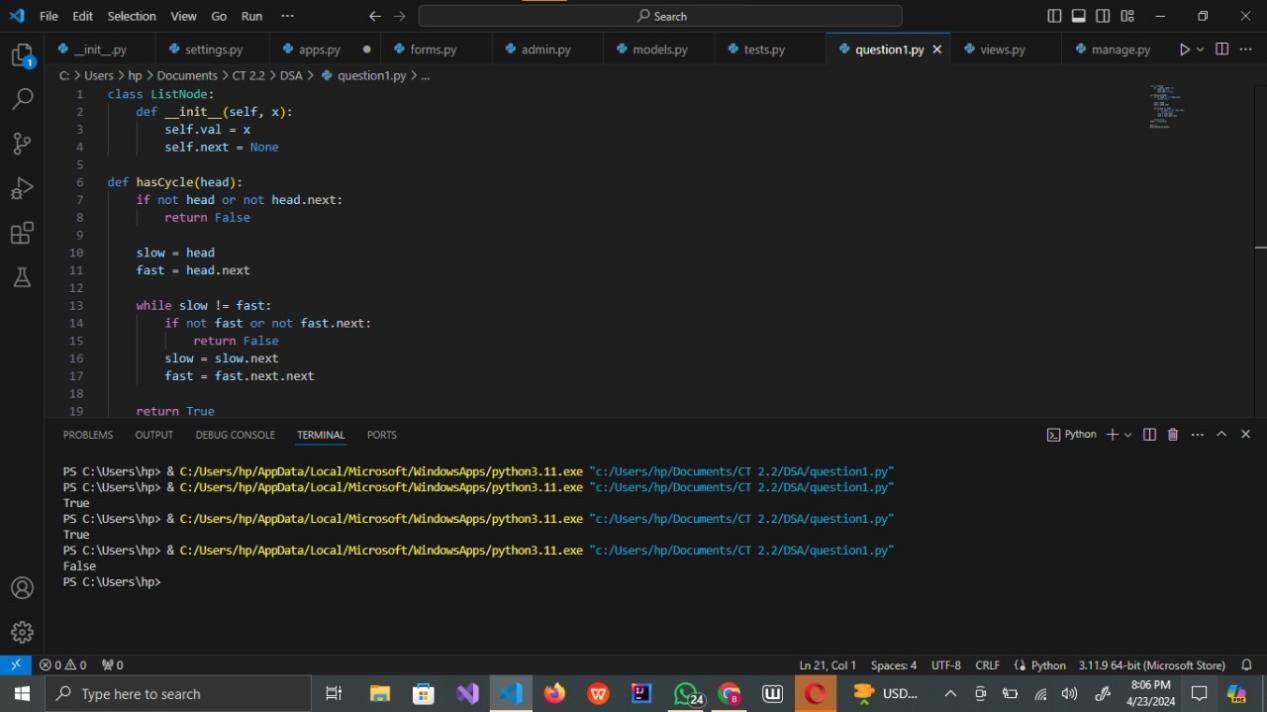
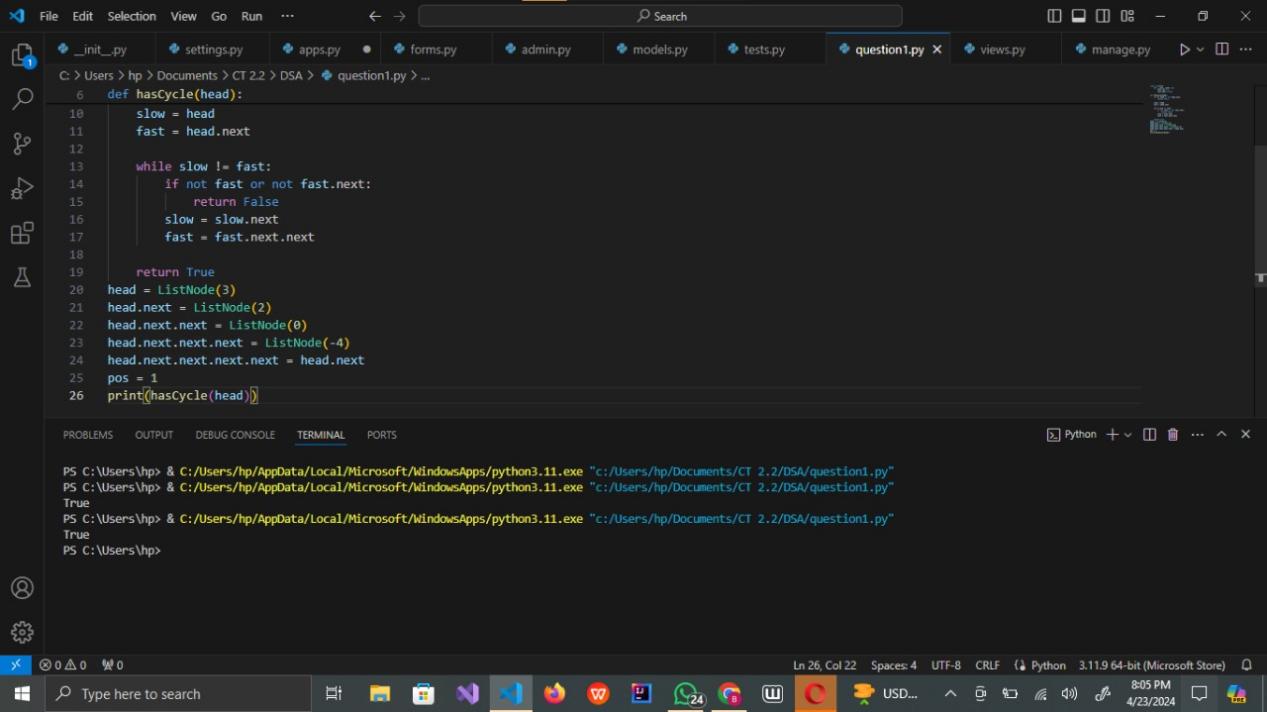
head.next.next.next = ListNode(-4)

head.next.next.next.next = head.next

pos = 1

print(hasCycle(head)) # Output: True

This algorithm has O(1) space complexity since it only uses two pointers regardless of the size of the linked list. It traverses the list once, so the time complexity is O(n), where n is the number of nodes in the linked list.



**Question two**

**QUESTION TWO (2):**

Given the head of a linked list, return *the node where the cycle begins. If there is no cycle, return* null. There is a cycle in a linked list if there is some node in the list that can be reached again by continuously following the next pointer. Internally, pos is used to denote the index of the node that tail's next pointer is connected to (0-indexed). It is -1 if there is no cycle. Note that pos is not passed as a parameter.

**To solve this problem, you can extend the Floyd's Tortoise and Hare algorithm. Once the two pointers meet, you reset one of them to the head of the linked list and move them both at the same speed. The point where they meet again will be the start of the cycle.**

class ListNode:

def \_\_init\_\_(self, x):

self.val = x

self.next = None

def detectCycle(head):

if not head or not head.next:

return None

# Phase 1: Detect cycle

slow = head

fast = head

while fast and fast.next:

slow = slow.next

fast = fast.next.next

if slow == fast:

break

else:

return None # No cycle

# Phase 2: Find start of the cycle

slow = head

while slow != fast:

slow = slow.next

fast = fast.next

return slow

# Example usage:

# Create the linked list

head = ListNode(3)

head.next = ListNode(1)

head.next.next = ListNode(0)

head.next.next.next = ListNode(-4)

head.next.next.next.next = head.next # creating a cycle

cycle\_start = detectCycle(head)

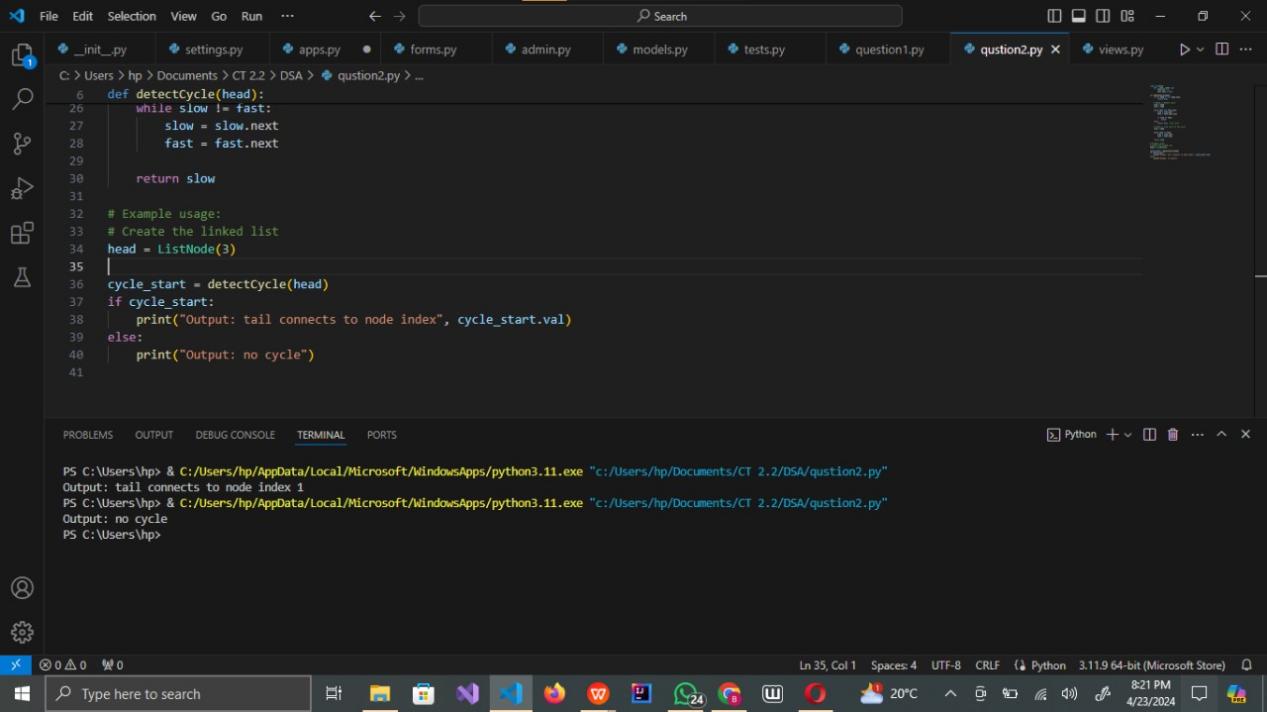
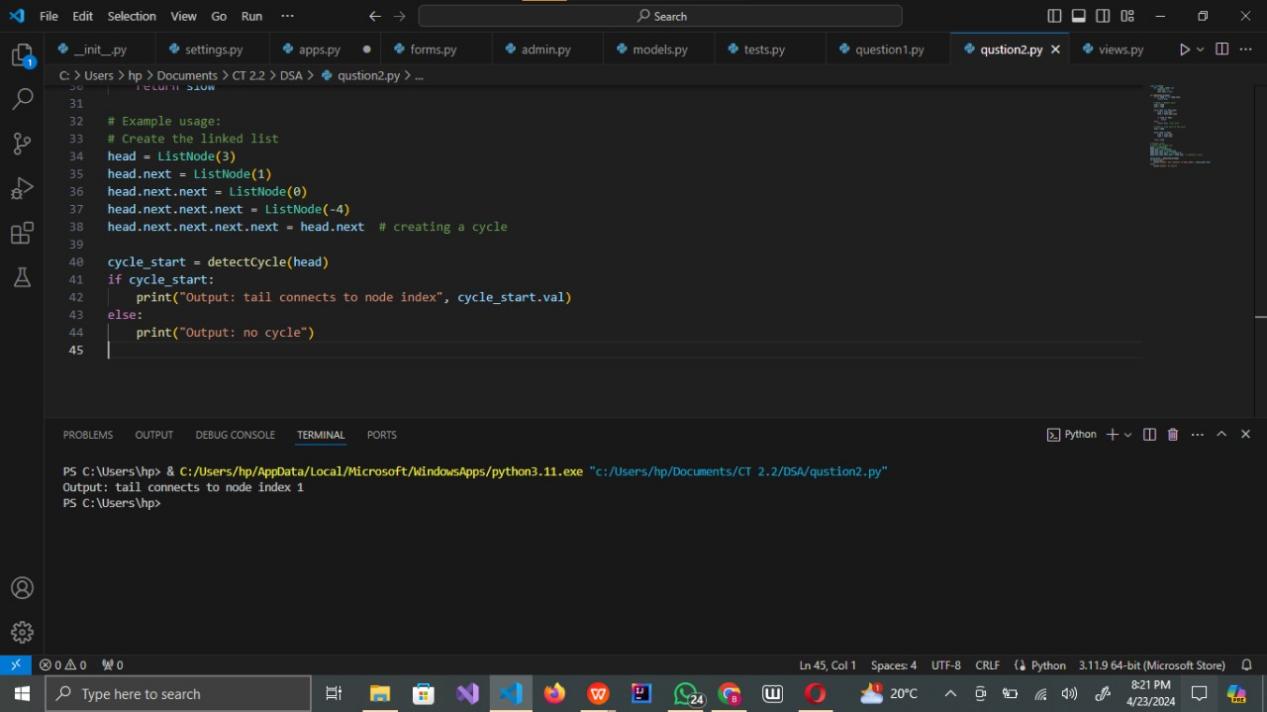
if cycle\_start:

print("Output: tail connects to node index", cycle\_start.val)

else:

print("Output: no cycle")

#output Output: tail connects to node index 1



This algorithm has O(1) space complexity since it only uses two pointers regardless of the size of the linked list. It traverses the list once for detecting the cycle and then once again to find the start of the cycle, so the time complexity is O(n), where n is the number of nodes in the linked list.

**Question three**

Write a function that takes the head of a linked list and returns the reversed list.

class ListNode:

def \_\_init\_\_(self, val=0, next=None):

self.val = val

self.next = next

def reverseLinkedList(head):

prev = None

current = head

while current:

next\_node = current.next

current.next = prev

prev = current

current = next\_node

return prev

# Example usage:

# Create the linked list

head = ListNode(1)

head.next = ListNode(2)

head.next.next = ListNode(3)

head.next.next.next = ListNode(4)

head.next.next.next.next = ListNode(5)

reversed\_head = reverseLinkedList(head)

while reversed\_head:

print(reversed\_head.val, end=" ")

reversed\_head = reversed\_head.next

#output 5 4 3 2 1

This code will print the values of the reversed linked list. The reverseLinkedList function takes the head of the linked list as input and returns the head of the reversed linked list. It iterates through the list, reversing the pointers as it goes, effectively reversing the list.

