# Singly Linked List in Python - Enhanced Notes

This document includes a **professional and educational version** of Singly Linked List (SLL) implementation in Python. It combines clean code with rich context: **real-world analogies, industry relevance**, and **detailed comments** to ensure both learners and professionals benefit.

## 🚀 What is a Singly Linked List?

A **Singly Linked List** is a **linear dynamic data structure** made of nodes, where each node contains: - data: The actual information - next: Reference to the next node in the list

Only the first node (head) is accessible directly. All navigation is done sequentially.

### 🏗️ Real-World Analogy:

Think of a **treasure map** where each clue (node) points to the location of the next clue. You can’t jump to the end—you have to follow the clues one by one.

### 🧠 Why is it useful in the industry?

* **Memory-efficient for dynamic data**: Used when data size is not known beforehand.
* **Web Browsers**: Back and forward button navigation uses linked structures.
* **Music/Playlist Apps**: Playlists where each song leads to the next.
* **Operating Systems**: For task scheduling (ready queue, etc.).

## 🧱 Node Structure

class Node:  
 """  
 A Node in a singly linked list that contains data and reference to the next node.  
 """  
 def \_\_init\_\_(self, data):  
 self.data = data # The value or payload of the node  
 self.next = None # Pointer to the next node in the list

## 🧰 SinglyLinkedList Class - Core API

class SinglyLinkedList:  
 """  
 Implements a singly linked list with operations to manipulate and access data.  
 """  
 def \_\_init\_\_(self):  
 self.head = None # Head points to the first node in the list

## 🔧 Core Operations

### 1. Insert at Beginning

def insert\_at\_beginning(self, data):  
 """  
 Inserts a node at the start of the list.  
 If the list is empty, new node becomes the head.  
 Otherwise, new node points to current head and becomes the new head.  
 """  
 new\_node = Node(data)  
 new\_node.next = self.head  
 self.head = new\_node

### 2. Insert at End

def insert\_at\_end(self, data):  
 """  
 Appends a node at the end of the list.  
 Handles empty list, single-node, and multi-node cases.  
 """  
 new\_node = Node(data)  
  
 if self.head is None:  
 self.head = new\_node  
 return  
  
 current\_node = self.head  
 while current\_node.next is not None:  
 current\_node = current\_node.next  
  
 current\_node.next = new\_node

### 3. Insert at a Specific Position

def insert\_at\_position(self, data, insert\_position):  
 """  
 Inserts a node at a given 1-based index.  
 If position is invalid, prints a message.  
 """  
 if insert\_position <= 0:  
 print("Invalid position")  
 return  
  
 if insert\_position == 1:  
 self.insert\_at\_beginning(data)  
 return  
  
 current\_node = self.head  
 current\_position = 1  
  
 while current\_position < insert\_position - 1 and current\_node is not None:  
 current\_position += 1  
 current\_node = current\_node.next  
  
 if current\_node is None:  
 print("Invalid position, fewer nodes in list")  
 return  
  
 new\_node = Node(data)  
 new\_node.next = current\_node.next  
 current\_node.next = new\_node

### 4. Delete at Beginning

def delete\_at\_beginning(self):  
 """  
 Removes the first node in the list by updating head pointer.  
 Handles empty and single-node list cases.  
 """  
 if self.head is None:  
 return  
  
 self.head = self.head.next

### 5. Delete at End

def delete\_at\_end(self):  
 """  
 Removes the last node of the list by traversing to the second-last node.  
 Edge cases: empty list and single-node list.  
 """  
 if self.head is None:  
 return  
  
 if self.head.next is None:  
 self.head = None  
 return  
  
 current\_node = self.head  
 while current\_node.next.next is not None:  
 current\_node = current\_node.next  
  
 current\_node.next = None

### 6. Delete at a Specific Position

def delete\_at\_position(self, delete\_position):  
 """  
 Deletes the node at the given 1-based position.  
 Verifies the position and edge conditions.  
 """  
 if delete\_position <= 0:  
 print("Invalid position")  
 return  
  
 if delete\_position == 1:  
 self.delete\_at\_beginning()  
 return  
  
 current\_node = self.head  
 current\_position = 1  
  
 while current\_position < delete\_position - 1 and current\_node is not None:  
 current\_position += 1  
 current\_node = current\_node.next  
  
 if current\_node is None or current\_node.next is None:  
 print("Invalid position, fewer nodes in list")  
 return  
  
 current\_node.next = current\_node.next.next

### 7. Search for a Key

def search(self, key):  
 """  
 Searches the list for the specified key.  
 If found, prints confirmation. Otherwise, says not found.  
 """  
 if self.head is None:  
 print("List is empty")  
 return  
  
 current\_node = self.head  
 while current\_node is not None:  
 if key == current\_node.data:  
 print("Key found")  
 return  
 current\_node = current\_node.next  
  
 print("Key not found")

### 8. Display List

def print\_list(self):  
 """  
 Prints the entire linked list in human-readable format.  
 """  
 if self.head is None:  
 print("List is empty")  
 return  
  
 current\_node = self.head  
 while current\_node is not None:  
 print(str(current\_node.data) + " --> ", end='')  
 current\_node = current\_node.next  
 print("None")

## 🧪 Driver Code Examples

def insert\_delete\_at\_given\_position(list):  
 """  
 Demonstrates inserting and deleting at specific positions.  
 Shows valid and invalid examples.  
 """  
 list.insert\_at\_position(10, -1)  
 list.delete\_at\_position(-1)  
  
 list.insert\_at\_position(10, 1)  
 list.delete\_at\_position(1)  
  
 list.insert\_at\_end(10)  
 list.insert\_at\_end(20)  
 list.insert\_at\_end(30)  
  
 list.insert\_at\_position(15, 2)  
 list.insert\_at\_position(40, 4)  
 list.insert\_at\_position(100, 10)

## 📌 Tips

* Always validate positions
* Handle edge cases (empty list, single-node list)
* Use helper functions for testing and debugging

## 📍 Conclusion

Singly Linked Lists are powerful tools for dynamic memory usage, making them highly useful in **real-time systems**, **compilers**, **network packet routing**, and more.