OOPS(object-oriented programming):

Here each object has its own set of features and functions and they can be used in programming, which In turn helps in program reusability.

<u>Example:</u>

imaging doing all the work in a hotel as a single person cooking, room cleaning, and receptionist by one person, now imagine hiring a bunch of people who are experts in particular fields (like chef, housekeeper, receptionist) to work here all of them are doing their individual work which can achieve our task

Here every person works as an object who has his own features and functionalities which helps to achieve our task.

OOPS Concepts:

consider we are creating a virtual hotel:

House keeper: here he has responsible for[2,3] floors and and he does cleaning and notify any damages to supervisor…..

here has refers to the "**Attributes**" (Fancy word of Variable) basically used to store values, its not free floating it is attached to a particular object

Does refers to **"Methods"(**Fancy word of functions) basically refers to the functions he need, here also functions are not free floating it is attached to object

Object = combination of Attibutes and methods(variables and functions), we can create multiple objects from same object like housekeeper1, housekeeper2 here the original blue print is called as **Class** and individual things created from it called as **Objects**

Objects Creating:

Real world objects Scenarios:

you can consider chocolate, mug, and mug as objects → but you can't say both mug's are same because even though they have same color,weight,shape may be one cup behaviour of full is empty and another is filled

Phone: identity-person's phone, attributes – color,shape,size, methods – ring,text,call (these all represents object) each object has its own identity

Non physical things: bank account also be refereed as objects – unique account no, methods: deposit and withdraw

Most programming languages have in built classes which we can refer to create our objects which are useful to us. Most of the time we need to create the functionalities or methods respective to particular object since each obj can be represented as unique in its own way

Syntax = robot1 = RobotBlueprint(),, accessing attribute or method = robot1.battery_level()

Classes:

Imagine a cookies cutters star, circle and all and you make star cookies based on this star cookies in different shapes and tastes based on the cutter, here cutters are the classes and cookies are the objects that we made using specified cutter.

<u>Class components:</u>

[Name: Star cookie, Attributes: weight, color, behaviour(methods): decorate(), consume()]

Method vs functions: it is a function which is specific to particular class only.

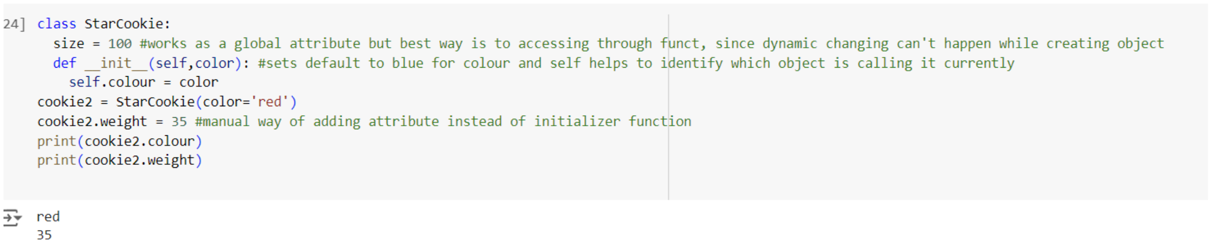
Class declaration: class PascalCase:

PascalCase(popular): every word of class_name should start with capital letter.
camelCase: except first word every subsequent word should start with capital letter.
Snake_case: all words are lowercase but separated with underscore.

Initializer in class: def __init__(self): this used to declare initializer to class generally useful for creation of attributes and self refers to it is refering same object..

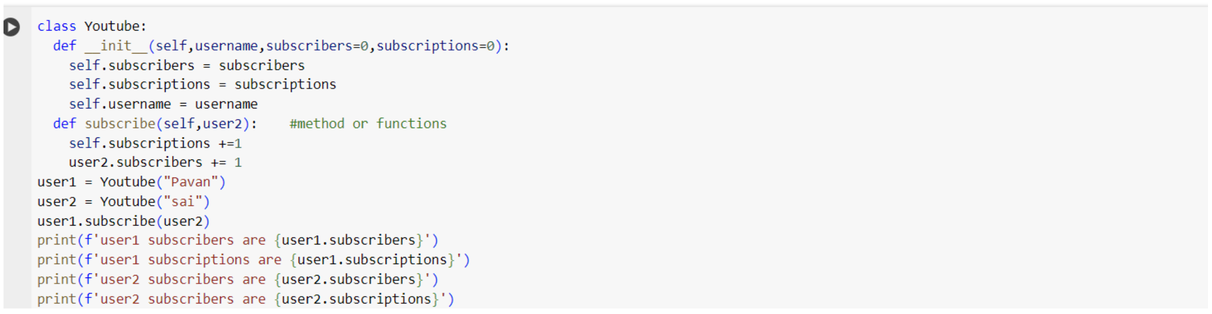
Object is the instance(has its own copy) of class

Self helps us to identify which object is currently calling (very helpful), method calling is simple just use normal .methodname and parameters it will be good



```
24] class StarCookie:
      size = 100 #works as a global attribute but best way is to accessing through funct, since dynamic changing can't happen while creating object
      def __init__(self,color): #sets default to blue for colour and self helps to identify which object is calling it currently
        self.colour = color
    cookie2 = StarCookie(color='red')
    cookie2.weight = 35 #manual way of adding attribute instead of initializer function
    print(cookie2.colour)
    print(cookie2.weight)
```

```
red
35
```



```
class Youtube:
  def __init__(self,username,subscribers=0,subscriptions=0):
    self.subscribers = subscribers
    self.subscriptions = subscriptions
    self.username = username
  def subscribe(self,user2):    #method or functions
    self.subscriptions +=1
    user2.subscribers += 1
user1 = Youtube("Pavan")
user2 = Youtube("sai")
user1.subscribe(user2)
print(f'user1 subscribers are {user1.subscribers}')
print(f'user1 subscriptions are {user1.subscriptions}')
print(f'user2 subscribers are {user2.subscribers}')
print(f'user2 subscribers are {user2.subscriptions}')
```

Note:

In python there are hundreds of pre defined classes basics things like arrays,strings are often included in those classes which are readily available to use.

Linked List:

Real life example:

Consider a train where it has engine(head) and tail(last compartments) and has many comparte are linked each other using links and each compartment is independent of one another and if we need to go to tail we need to travel from head to tail while train is moving

Linked list behaves exactly it has nodes which is independent of one another and each node has data and address(physical location of next node) and they are interlinked to one another by reference next node address that they have, head(points to first node) helps us to find starting point which has next node address and tail(points to last node) consists of last node physical loca

List vs Linked List:

In List the elements are located continuously and can be accessed through indexing, but here elements are dispersed as nodes and they are using pointers to point to next node where head points starting node and tail points ending node

Types:

Single linked List: normal linked list where node has data and points to next node
Circular Linked List: here the last node has location of starting node (ex: chess player turns)(unidire)
Double Linked List: here the node stores both previous and next node memory locations (first node previous is null and last lode next is null) since they are not refers to anything(ex: songs playing)
Circular Doubly Linked List: here last node stores address of first node and first node address of last and nodes are interconnected using doubly linked list(bi directional)(ex: photos looping to first and first to last)

Memory allocation: elements are allocated randomly and are interconnected to each other by referencing to next node, this helps the size to be dynamic, drawback to access particular element we need to traverse from starting point.

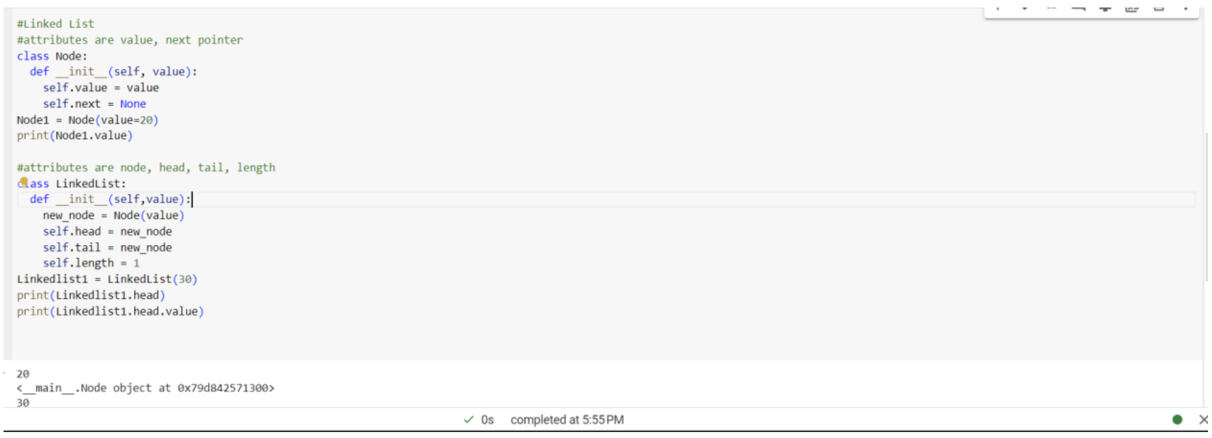
Operations:

Node class:

We can create node class as self.value, self.next = None , since while creating we don’t know next node address and each node class is different from other so using class is the good way which make it more unique.

Linked List Class:

Attributes are head, tail, node,length



```
#Linked List
#attributes are value, next pointer
class Node:
  def __init__(self, value):
    self.value = value
    self.next = None
Node1 = Node(value=20)
print(Node1.value)

#attributes are node, head, tail, length
class LinkedList:
  def __init__(self,value):
    new_node = Node(value)
    self.head = new_node
    self.tail = new_node
    self.length = 1
Linkedlist1 = LinkedList(30)
print(Linkedlist1.head)
print(Linkedlist1.head.value)
```

```
20
<__main__.Node object at 0x79d842571300>
30
```

✓ 0s completed at 5:55 PM

And Rest all basic operations like Insertion, Traversing, get, Set Value, Append, Pop ,Remove, and delete all nodes can be done as below

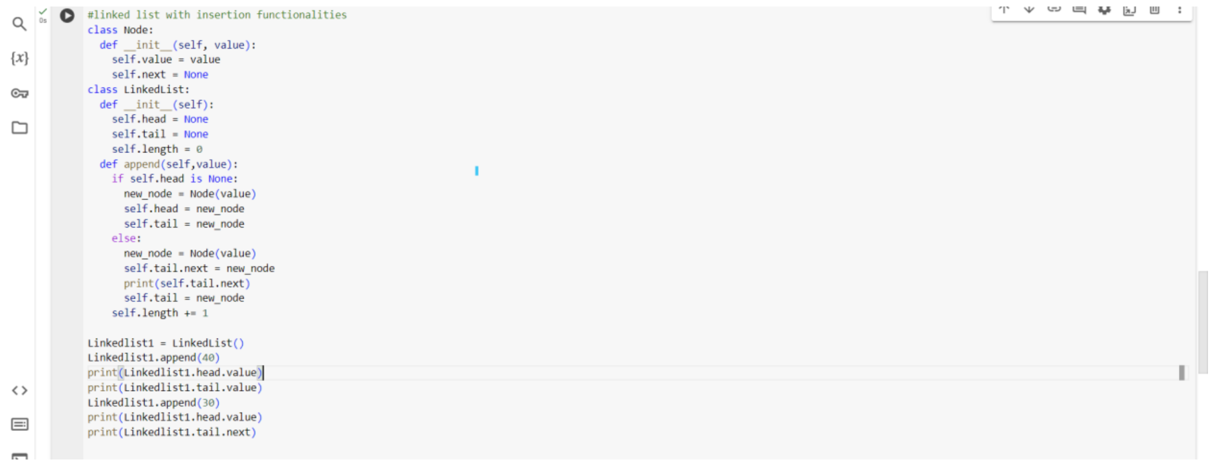
Insertion:

Beginning: create a node and add its reference to the header and take starting our node should refer to the first node

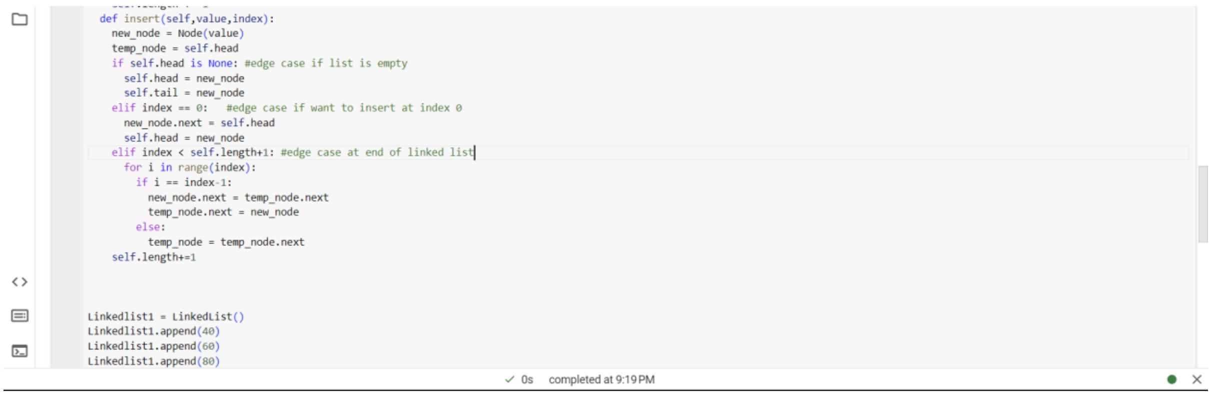
Middle: traverse and after the desired element add next reference to us and we is refered to next element in the hierarchy

Last: travel all and add our element as a reference to last element and tail is a reference to us

Append and insert implementation:



```
#linked list with insertion functionalities
class Node:
  def __init__(self, value):
    self.value = value
    self.next = None
class LinkedList:
  def __init__(self):
    self.head = None
    self.tail = None
    self.length = 0
  def append(self,value):
    if self.head is None:
      new_node = Node(value)
      self.head = new_node
      self.tail = new_node
    else:
      new_node = Node(value)
      self.tail.next = new_node
      print(self.tail.next)
      self.tail = new_node
    self.length += 1

Linkedlist1 = LinkedList()
Linkedlist1.append(40)
print(Linkedlist1.head.value)
print(Linkedlist1.tail.value)
Linkedlist1.append(30)
print(Linkedlist1.head.value)
print(Linkedlist1.tail.next)
```



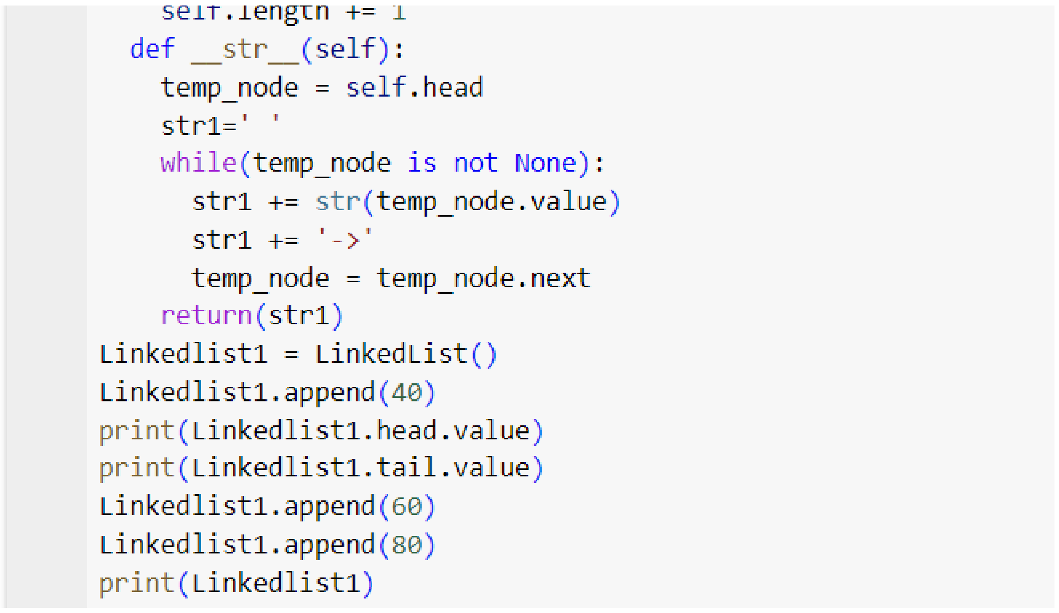
```
  def insert(self,value,index):
    new_node = Node(value)
    temp_node = self.head
    if self.head is None: #edge case if list is empty
      self.head = new_node
      self.tail = new_node
    elif index == 0:   #edge case if want to insert at index 0
      new_node.next = self.head
      self.head = new_node
    elif index < self.length+1: #edge case at end of linked list
      for i in range(index):
        if i == index-1:
          new_node.next = temp_node.next
          temp_node.next = new_node
        else:
          temp_node = temp_node.next
    self.length+=1


Linkedlist1 = LinkedList()
Linkedlist1.append(40)
Linkedlist1.append(60)
Linkedlist1.append(80)
```

✓ 0s completed at 9:19 PM

Adding at begging also is almost same as append method functionality

Traversing:

__str__ method is useful or returns the meaning full representation as output for printing the class, if we don't use __str__ we will get object at xx as the output.



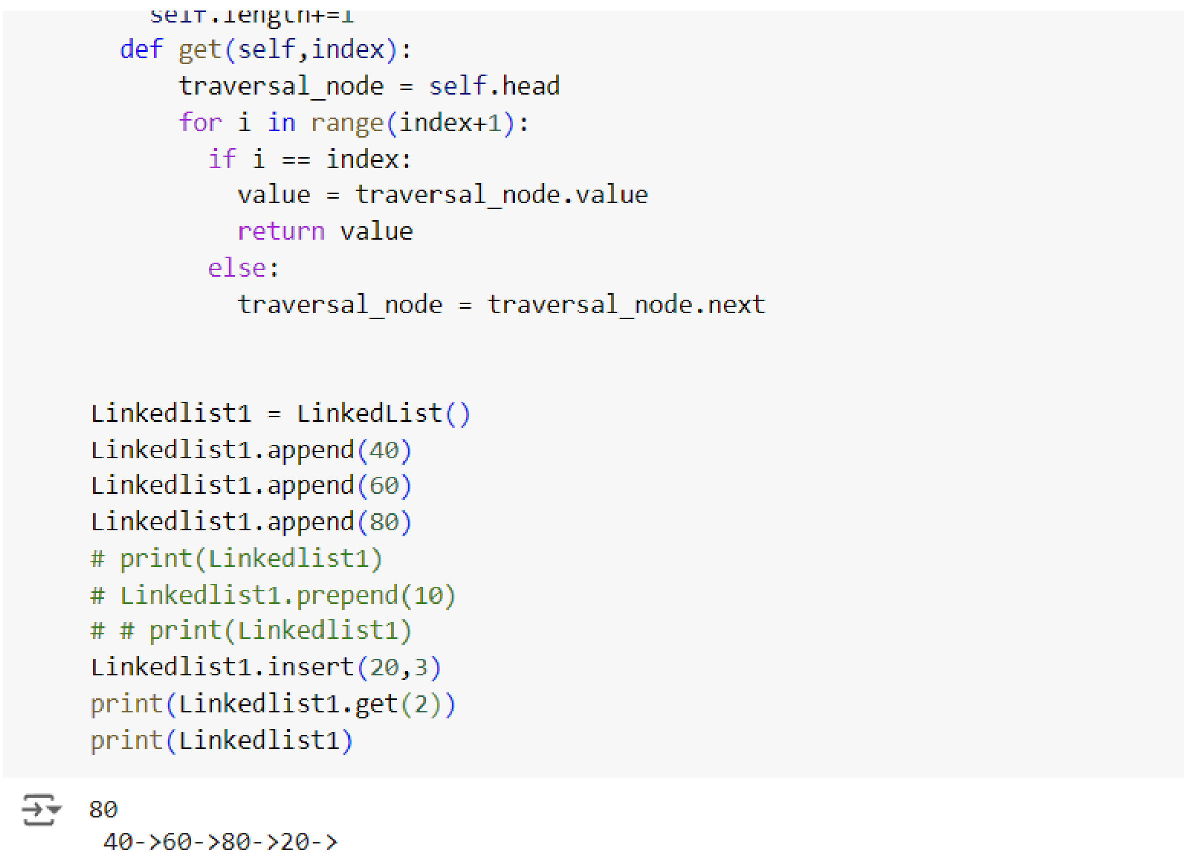
```
    self.length += 1
  def __str__(self):
    temp_node = self.head
    str1=' '
    while(temp_node is not None):
      str1 += str(temp_node.value)
      str1 += '->'
      temp_node = temp_node.next
    return(str1)
Linkedlist1 = LinkedList()
Linkedlist1.append(40)
print(Linkedlist1.head.value)
print(Linkedlist1.tail.value)
Linkedlist1.append(60)
Linkedlist1.append(80)
print(Linkedlist1)
```

Search:

Same just traversal and check the value in node whether it is similar to out target node

Get Method:

Getting the value of node associated with the passing node index



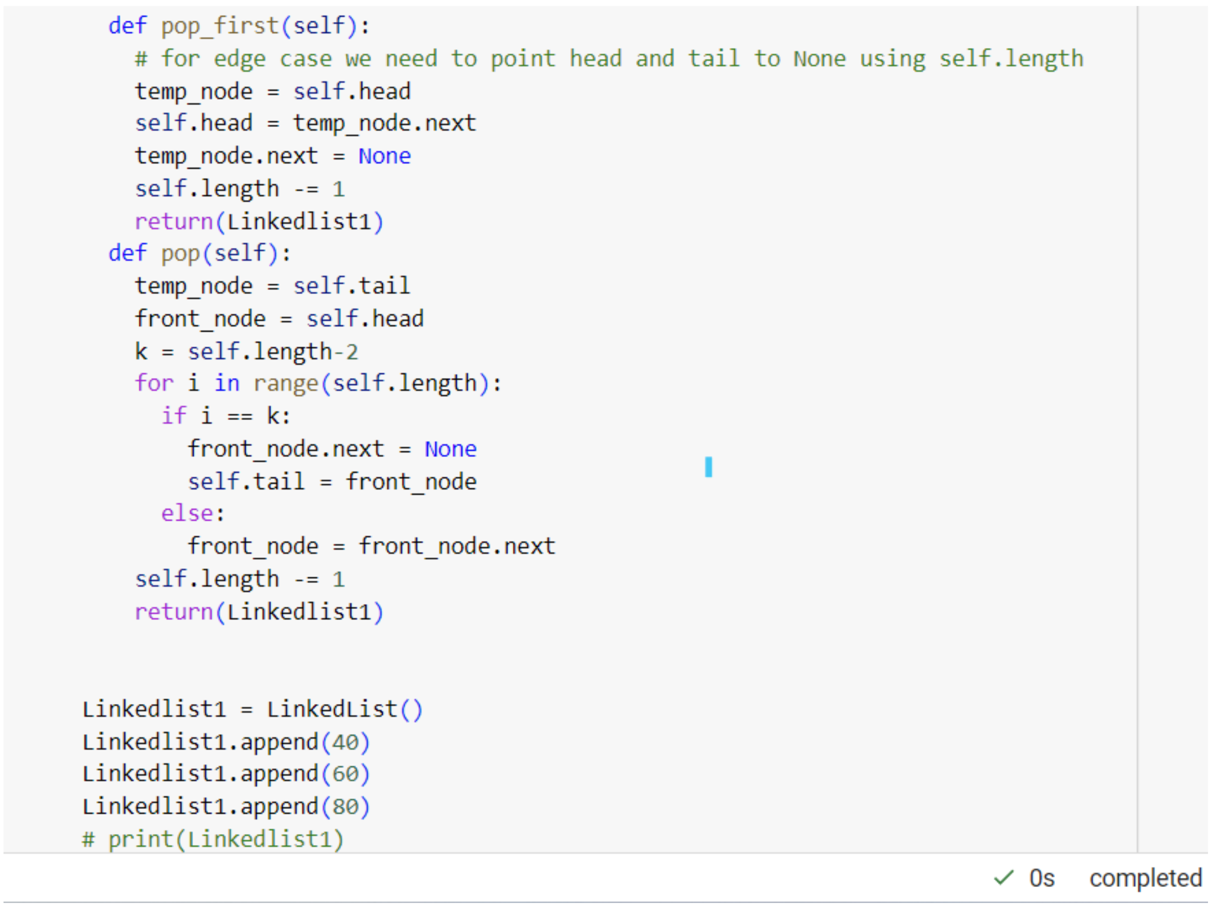
```
    self.length+=1
  def get(self,index):
      traversal_node = self.head
      for i in range(index+1):
        if i == index:
          value = traversal_node.value
          return value
        else:
          traversal_node = traversal_node.next


Linkedlist1 = LinkedList()
Linkedlist1.append(40)
Linkedlist1.append(60)
Linkedlist1.append(80)
# print(Linkedlist1)
# Linkedlist1.prepend(10)
# # print(Linkedlist1)
Linkedlist1.insert(20,3)
print(Linkedlist1.get(2))
print(Linkedlist1)
```

```
80
 40->60->80->20->
```

SET Method: Just access the index and change the current value to the target value similar to search method

Pop First and Pop Method: in Pop First just remove first node(making .next to None) using head referencing and change head point to next node this works for pop first,

Similarly in pop method move tail to 2nd last node and update 2nd last node reference to none(logic)

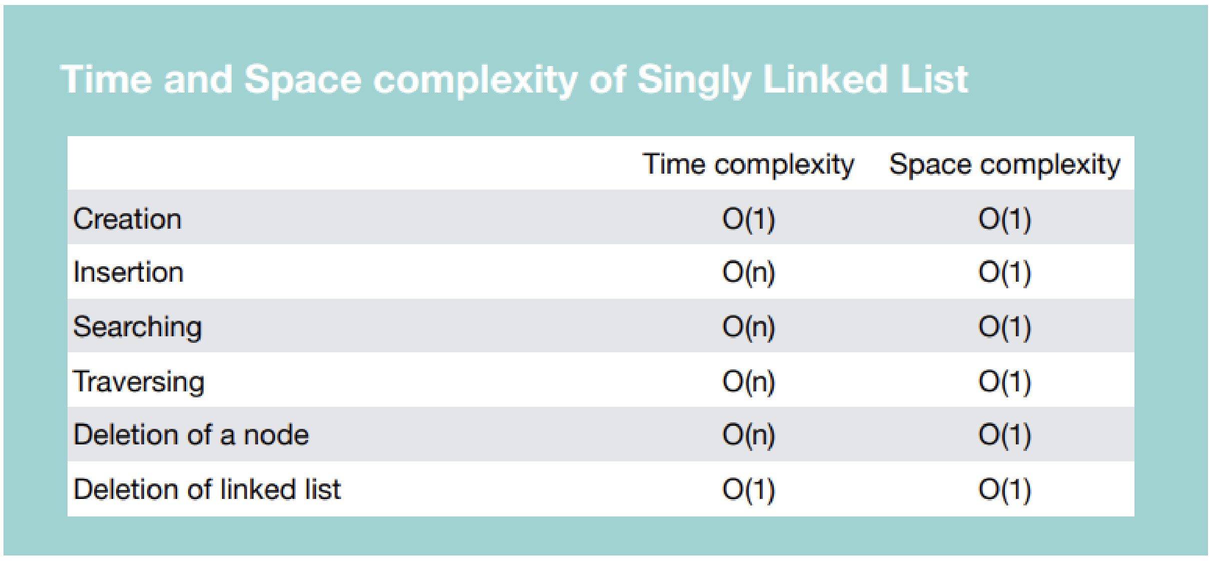


```
  def pop_first(self):
    # for edge case we need to point head and tail to None using self.length
    temp_node = self.head
    self.head = temp_node.next
    temp_node.next = None
    self.length -= 1
    return(Linkedlist1)
  def pop(self):
    temp_node = self.tail
    front_node = self.head
    k = self.length-2
    for i in range(self.length):
      if i == k:
        front_node.next = None
        self.tail = front_node
      else:
        front_node = front_node.next
    self.length -= 1
    return(Linkedlist1)


Linkedlist1 = LinkedList()
Linkedlist1.append(40)
Linkedlist1.append(60)
Linkedlist1.append(80)
# print(Linkedlist1)
```

Remove method:

use 2 pointers one for actual element and one for previous allocate previous.next = actual.next and then make actual.next to None

Delete Linked list: make self.head and self.tail to None main link is destroyed.



Time and Space complexity of Singly Linked List

	Time complexity	Space complexity
Creation	O(1)	O(1)
Insertion	O(n)	O(1)
Searching	O(n)	O(1)
Traversing	O(n)	O(1)
Deletion of a node	O(n)	O(1)
Deletion of linked list	O(1)	O(1)

Reversing Method:

Remember address storing in the node is the one which represents the direction try to change address direction will be automatically changed

Note:

In range(-1) the loop never performs since -1 is greater than 0 the loop never going to work

always remember the edge case like empty list, dealing with head and tail

Circular Single Linked List:

It is same as single linked list but tail next reference or last node next reference is pointed to the head node

Class node: will be same just contain value and next attributes

Linked list: (Empty) head and tail points to none and length is zero, for single element node.next = node self pointing and head and tail should points to the single element

Operations:

Most of all operations remain same in SLL and CSLL

<u>Append:</u> same as SLL just add new node at end and change tail and also next reference(New Node) to head

<u>Prepend:</u> same as SLL create new node and add next of it to current head pointing node update head and also update tail to the new node

<u>Traversing:</u> same as SSL traverse via node = node.next but remember to break after tail node

<u>Searching:</u> same do Traversing but compare current node with target and returns true if present

<u>Get:</u> access the index and return the current node of that index

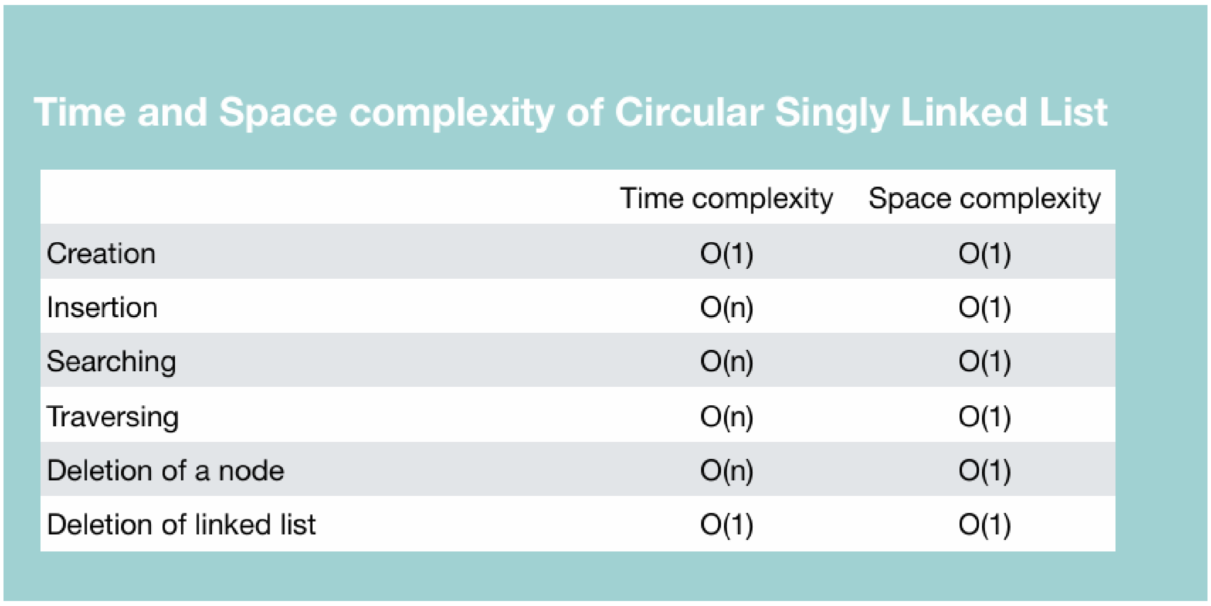
<u>Set:</u> just access the index and update the value with given value same as SLL

<u>Pop First:</u> make head point to next reference and update tail and make initial.next = None

<u>Pop:</u> traverse up to less than length-1 update tail and make reference to head update last node next=None

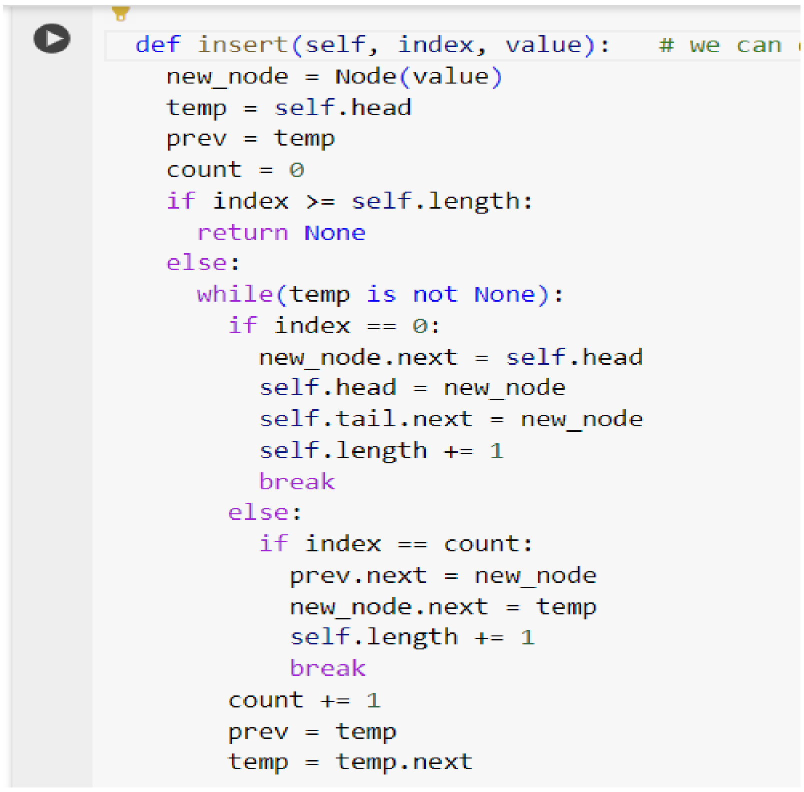
<u>Remove:</u> traverse using current and prev node and just remove current link current-next with pre nex

<u>Delete all nodes:</u> update head and tail and last node next to none connection will be breaked

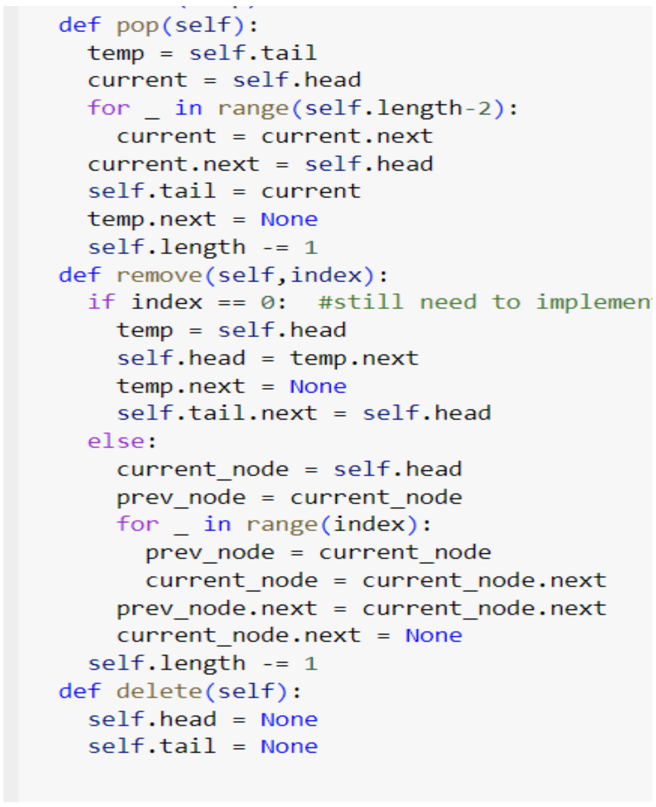


Time and Space complexity of Circular Singly Linked List

	Time complexity	Space complexity
Creation	O(1)	O(1)
Insertion	O(n)	O(1)
Searching	O(n)	O(1)
Traversing	O(n)	O(1)
Deletion of a node	O(n)	O(1)
Deletion of linked list	O(1)	O(1)



```python
def insert(self, index, value):   # we can
  new_node = Node(value)
  temp = self.head
  prev = temp
  count = 0
  if index >= self.length:
    return None
  else:
    while(temp is not None):
      if index == 0:
        new_node.next = self.head
        self.head = new_node
        self.tail.next = new_node
        self.length += 1
        break
      else:
        if index == count:
          prev.next = new_node
          new_node.next = temp
          self.length += 1
          break
      count += 1
      prev = temp
      temp = temp.next
```



```
def pop(self):
  temp = self.tail
  current = self.head
  for _ in range(self.length-2):
    current = current.next
  current.next = self.head
  self.tail = current
  temp.next = None
  self.length -= 1
def remove(self,index):
  if index == 0:  #still need to implemen
    temp = self.head
    self.head = temp.next
    temp.next = None
    self.tail.next = self.head
  else:
    current_node = self.head
    prev_node = current_node
    for _ in range(index):
      prev_node = current_node
      current_node = current_node.next
    prev_node.next = current_node.next
    current_node.next = None
  self.length -= 1
def delete(self):
  self.head = None
  self.tail = None
```

Note:

In python if you remove all references to the object it is eligible for garbage collection

Double Linked List:

Incase of single linked list we can’t go to previous node it is unidirectional, but double linked list resolves that issue here each node points to the previous and next node.

Node class: attributes → value, next reference, prev reference(just one addition to SLL Node class)

Linked List: has same attributed head, tail, length

Note:

Always consider edge cases like empty linked list and single element

Operations:

Append: access self.tail and reference tail next to new_node and new_node prev to tail and finally move tail to new_node

Prepend: create node and just append it at front using head accessing of node next and head prev reference

Traversal: just access self.head and traverse using .next references, you can do reverse traversal also using access from last(self.tail) and prev references

Search: traverse and compare node value with target that's it

Get: here we have an advantage of getting node by index, if index is less than /2 we can start from head else we can start from tail traversing for accessing that index node

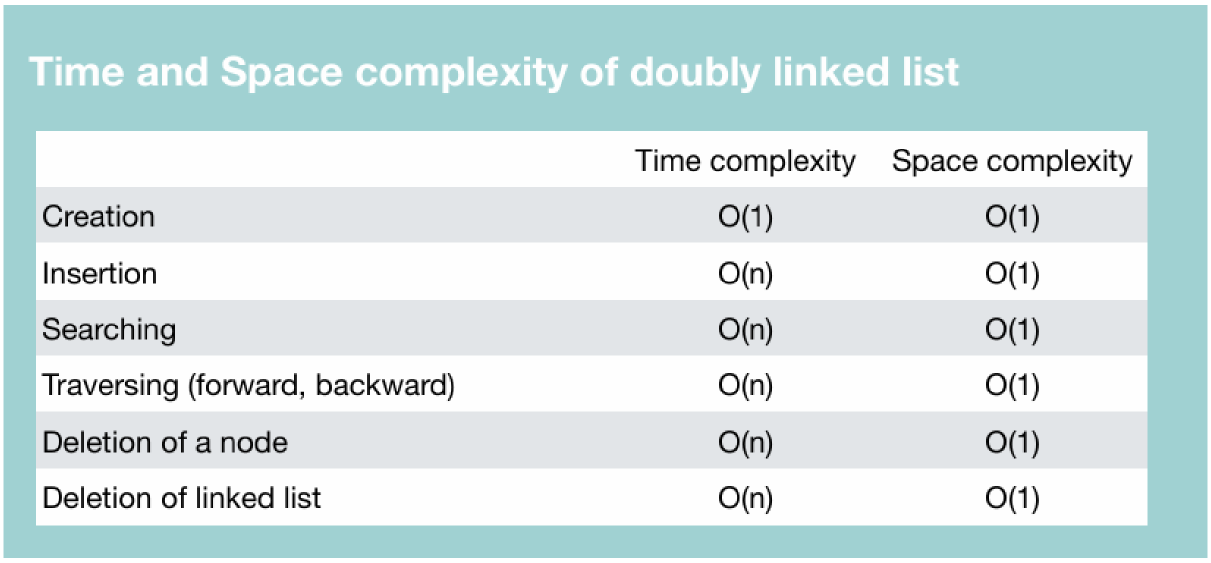
Set Value: same as get operation for accessing index and update the value with target node

Insert: access up to prev index and here create new node and update prev and next references of new_node with prev index and prev index next also change the references of prev and next nodes

Pop: access self.tail and refer to self.tail.prev and change tail and next reference to None, kind of easy as we are accessing through tail

Remove: create connection between prev and next node of current index and then make prev and next reference of current node to None

Deletion all nodes: head and tail and length is set to 0.


Time and Space complexity of doubly linked list

	Time complexity	Space complexity
Creation	O(1)	O(1)
Insertion	O(n)	O(1)
Searching	O(n)	O(1)
Traversing (forward, backward)	O(n)	O(1)
Deletion of a node	O(n)	O(1)
Deletion of linked list	O(n)	O(1)

Circular Double Linked List:

Here Last node points to the first node and first node points to the last node i.e, head to tail, tail to head

Node class : attributes value, prev, next same as double linked list

Linked List: attributes head, tail, length same as double linked list

Operations:

append: same as double linked list access last node using tail add node and remember to link to self.head

Prepend: accessing head and add in front and remember to link to tail

Traverse forward and reverse: we can traverse both forward and backward since we have both prev and next attributes

Search: just traverse and search the target value

Get: traversing and returning the particular index node

Set: Traverse and set the particular index node

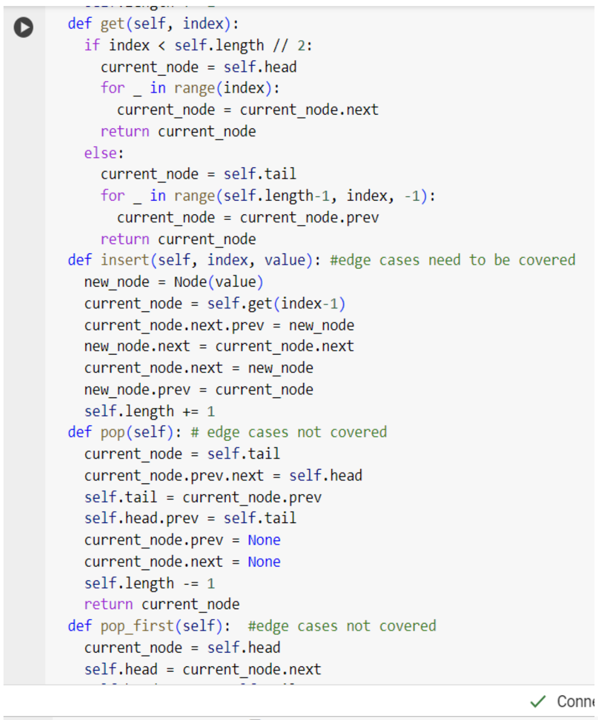
Insert: same as DLL traverse up to previous index and link the new_node prev and next with current index

Pop first: remove the first element as as DLL but link with self.tail

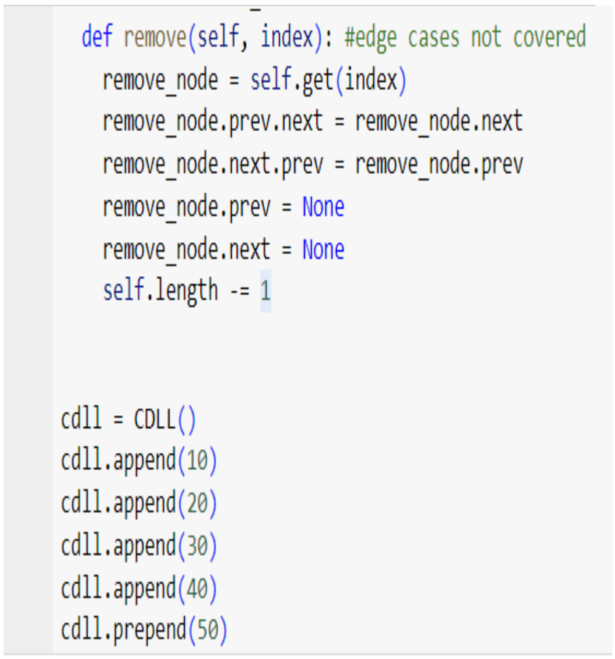
Pop: remove last node same as DLL but remember to reference to self.head

Remove: same as DLL match prev and next nodes together of the current node

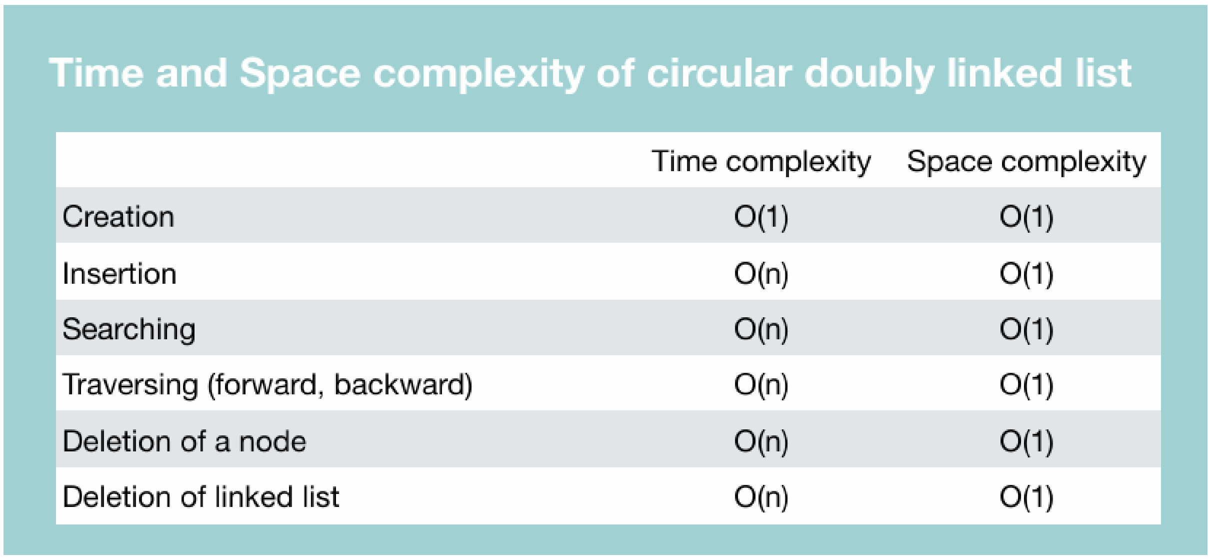
Delete CD Linked list: head, tail, length to 0 since nothing external is pointing the object it was treated as a garbage by the compiler



```
  def get(self, index):
    if index < self.length // 2:
      current_node = self.head
      for _ in range(index):
        current_node = current_node.next
      return current_node
    else:
      current_node = self.tail
      for _ in range(self.length-1, index, -1):
        current_node = current_node.prev
      return current_node
  def insert(self, index, value): #edge cases need to be covered
    new_node = Node(value)
    current_node = self.get(index-1)
    current_node.next.prev = new_node
    new_node.next = current_node.next
    current_node.next = new_node
    new_node.prev = current_node
    self.length += 1
  def pop(self): # edge cases not covered
    current_node = self.tail
    current_node.prev.next = self.head
    self.tail = current_node.prev
    self.head.prev = self.tail
    current_node.prev = None
    current_node.next = None
    self.length -= 1
    return current_node
  def pop_first(self):  #edge cases not covered
    current_node = self.head
    self.head = current_node.next
```



```
  def remove(self, index): #edge cases not covered
    remove_node = self.get(index)
    remove_node.prev.next = remove_node.next
    remove_node.next.prev = remove_node.prev
    remove_node.prev = None
    remove_node.next = None
    self.length -= 1


cdll = CDLL()
cdll.append(10)
cdll.append(20)
cdll.append(30)
cdll.append(40)
cdll.prepend(50)
```



Time and Space complexity of circular doubly linked list

	Time complexity	Space complexity
Creation	O(1)	O(1)
Insertion	O(n)	O(1)
Searching	O(n)	O(1)
Traversing (forward, backward)	O(n)	O(1)
Deletion of a node	O(n)	O(1)
Deletion of linked list	O(n)	O(1)

STACK

Stack is a data structure that stores the data in last-in/ first-out manner

Real life examples: Stacking books, stacking plates, back button in chrome

Operations:

Create_stack: creating a new stack using linked list and python list

Push: insertion into the stack --> stack_name.push()

Pop: remove last element from the stack --> stack_name .pop() (if stack is empty raise error)

Peek: returns the last element from the stack but doesn't remove the last element (stack_name.peek())

isEmpty: returns False if not empty --> stack_name.isEmpty()

isFull: return False if not Full will be useful when we created using size limit stack_name.isFull()

Delete: deletes the stack from the memory ---> stack_name.delete()

Creation and operations using list:

Python linked list implemention is easy for stack but disadvantage is system may be slowed because elements are stored In contiguous manner

Stack class for creation: just “two” initial attribute empty list and maxlen (self.list = [])

isEmpty we can create a function in stack class that just checks whether list is empty or not

isFull same creating function in stack to check whether list size is = max length

push function is implemented using append method In list but remember edge case max len

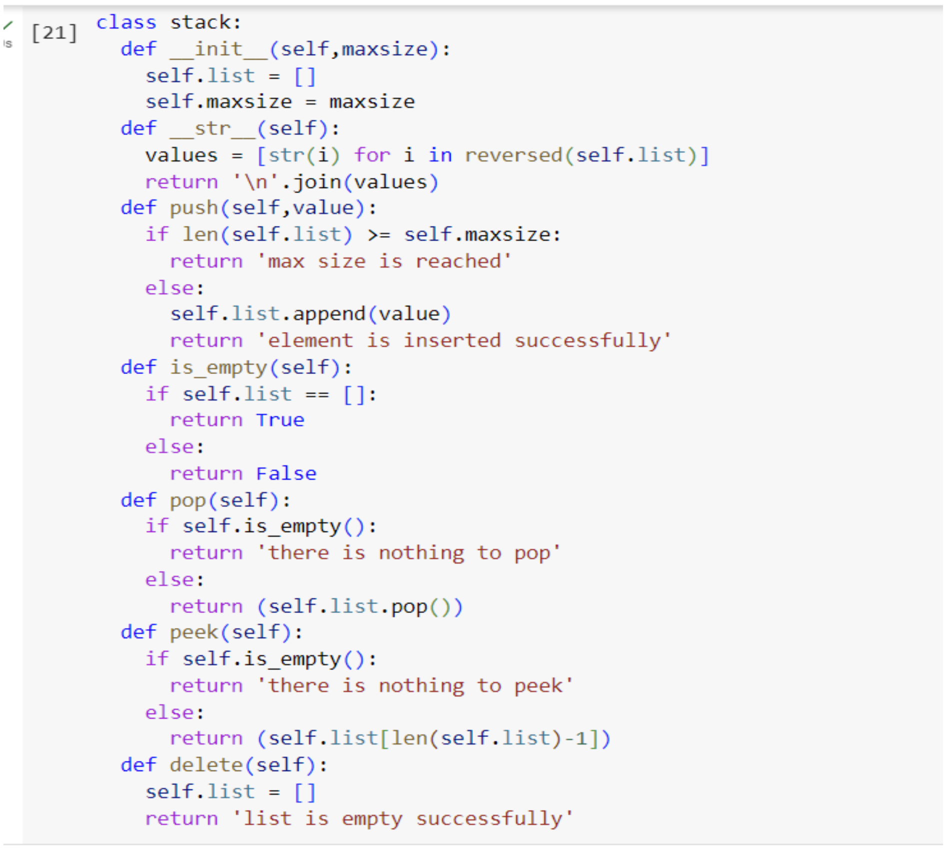
pop function is implemented using .pop() method

peek function is implemented using index accessing of list of last element

Delete function just set the list to empty

Note:

Maxsize is not a mandatory attribute



```
[21] class stack:
       def __init__(self,maxsize):
         self.list = []
         self.maxsize = maxsize
       def __str__(self):
         values = [str(i) for i in reversed(self.list)]
         return '\n'.join(values)
       def push(self,value):
         if len(self.list) >= self.maxsize:
           return 'max size is reached'
         else:
           self.list.append(value)
           return 'element is inserted successfully'
       def is_empty(self):
         if self.list == []:
           return True
         else:
           return False
       def pop(self):
         if self.is_empty():
           return 'there is nothing to pop'
         else:
           return (self.list.pop())
       def peek(self):
         if self.is_empty():
           return 'there is nothing to peek'
         else:
           return (self.list[len(self.list)-1])
       def delete(self):
         self.list = []
         return 'list is empty successfully'
```

Stack Creations and Operations using Linked List:

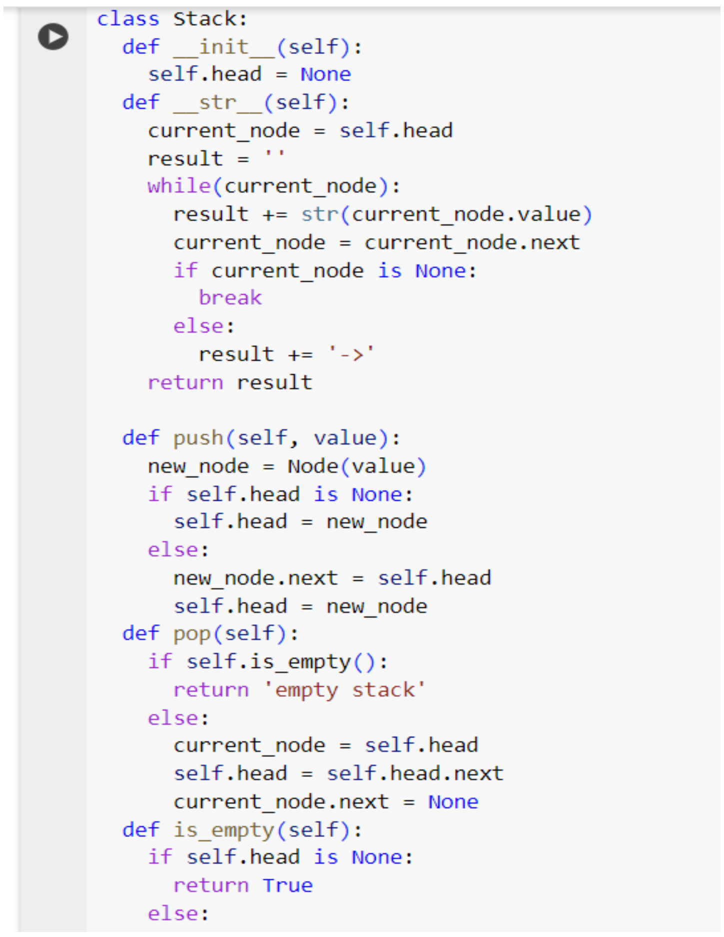
Stack is nothing but a single linked list with only head reference

And pop and push operations are pop_first and prepend methods in single linked list,

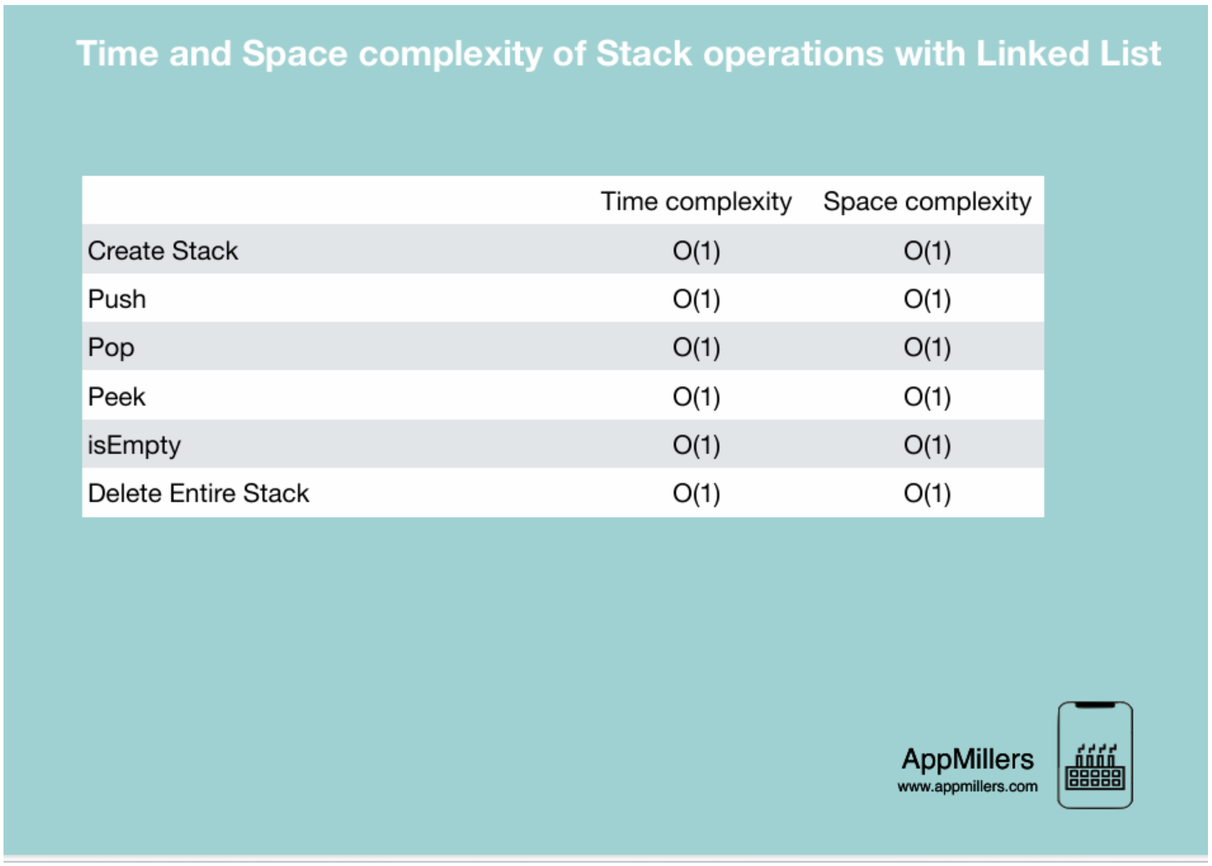
peek is the first element that head is pointing to, and delete is making head pointer to None

Advantages and disadvantages:

Can be used for LIFO functionality and chances of data corruption is minimal since accessing the lowest element is tough need to traverse all back down(Disadvantage)



```
class Stack:
  def __init__(self):
    self.head = None
  def __str__(self):
    current_node = self.head
    result = ''
    while(current_node):
      result += str(current_node.value)
      current_node = current_node.next
      if current_node is None:
        break
      else:
        result += '->'
    return result

  def push(self, value):
    new_node = Node(value)
    if self.head is None:
      self.head = new_node
    else:
      new_node.next = self.head
      self.head = new_node
  def pop(self):
    if self.is_empty():
      return 'empty stack'
    else:
      current_node = self.head
      self.head = self.head.next
      current_node.next = None
  def is_empty(self):
    if self.head is None:
      return True
    else:
```



Time and Space complexity of Stack operations with Linked List

	Time complexity	Space complexity
Create Stack	O(1)	O(1)
Push	O(1)	O(1)
Pop	O(1)	O(1)
Peek	O(1)	O(1)
isEmpty	O(1)	O(1)
Delete Entire Stack	O(1)	O(1)

AppMillers
www.appmillers.com

Problems:

LinkedList Cycle → tortoise and hare (hare moves towards by 2, and tortoise try to move away be 1 here difference is 1 so there are bound to reach each other when difference is zero)

Finding starting node of cycle → lets assume slow is at starting cycle and distance covered is L1 and fast is away from slow and it is at distance d to slow (total loop length is L1+d), now in order to collide each other slow must travel distance d, and fast 2d since difference is 1 for every jump now at the point they collide distance from head is 2L so slow is moved to head and now slow and fast are given one jump so the point they collide is starting node of cycle