NANYANG TECHNOLOGICAL UNIVERSITY

Data Structures & Algorithms in Python: Stacks

Dr. Owen Noel Newton Fernando

College of Computing and Data Science

Linked List

A LinkedList consists of nodes where each node has data and a pointer to the next node, ending with None. It's represented as sequential nodes connected by pointers in memory.

Key Operations:

- **Insert**: Add elements at start or anywhere
- **Delete**: Remove elements from any position
- **Search**: Find elements by traversing through nodes

Types

- Singly Linked List (one direction).
- Doubly Linked List (both directions).
- Circular Linked List (last node points to the first)

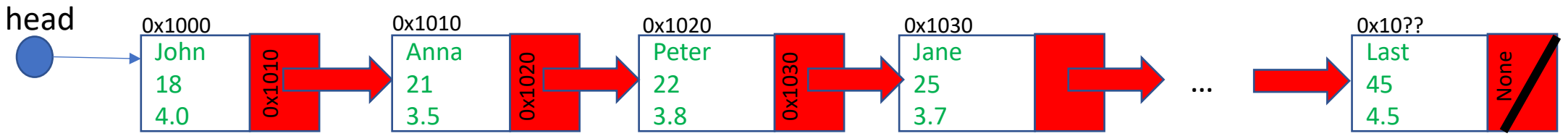
Implementation of a linked list in Python

- A node class is required to store each data and the link to the next node

```
class Node:
    def __init__(self, data, next):
        self.data = data
        self.next = next
```

- `def __init__(self, data, next):` This is the constructor method that initializes a new Node object:
  - `self` refers to the instance being created (Refers to current node instance)
  - `data` is the parameter that will store the node's value
  - `self.next`: Points to next node (Initially set to None and Creates link between nodes)
- Node Components:
  - `self.data = data` → Stores the value in node
  - `self.next = next` → Points to next node (empty at start)

Implementation of a linked list in Python



- A linked list class is used to create and manage a list of nodes.
- The head node is essential to locate the first node in the list.
- Additional pointers like a tail node can improve efficiency by pointing to the last node.
- The class supports operations such as insertion, deletion, and traversal to manage the list effectively.

```
class LinkedList:
    def __init__(self):
        self.head = None
```

Implementation of a linked list in Python

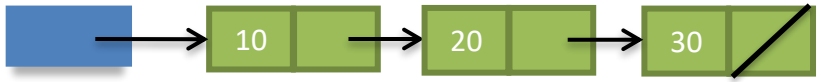
- Just introduce a new member in the linked list class, *size*
- Initialize *size* as zero
- When you add or remove a node, increase or decrease *size* by one accordingly

```
1  class ListNode:
2      def __init__(self, item):
3          self.item = item
4          self.next = None
5
6  class LinkedList:
7      def __init__(self):
8          self.head = None
9          self.size = 0
```

Doubly Linked List

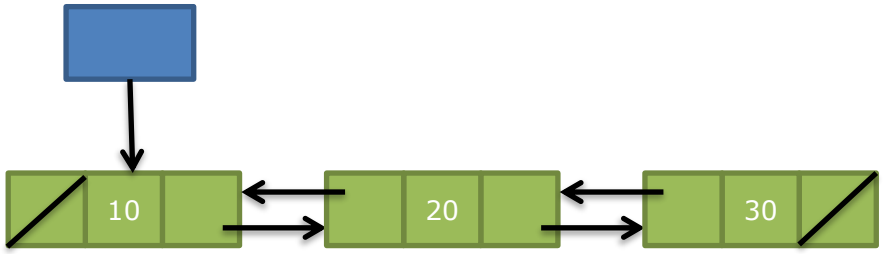
Singly Linked list: Only one link. Traversal of the list is one way only.

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
```



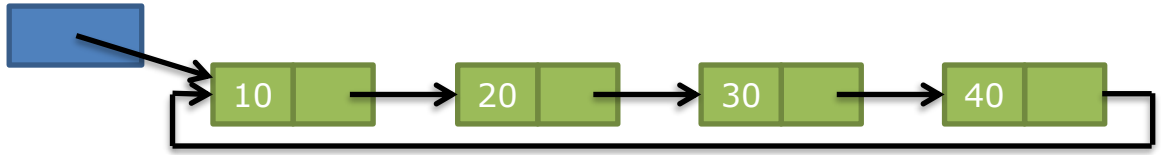
Doubly Linked List: two links in each node. It can search forward and backward.

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
        self.pre = None
```

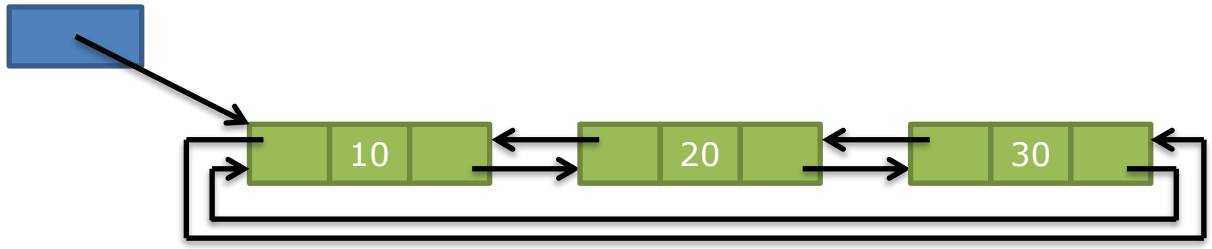


Circular Linked List

- Circular singly linked lists
  - Last node has next pointer pointing to first node

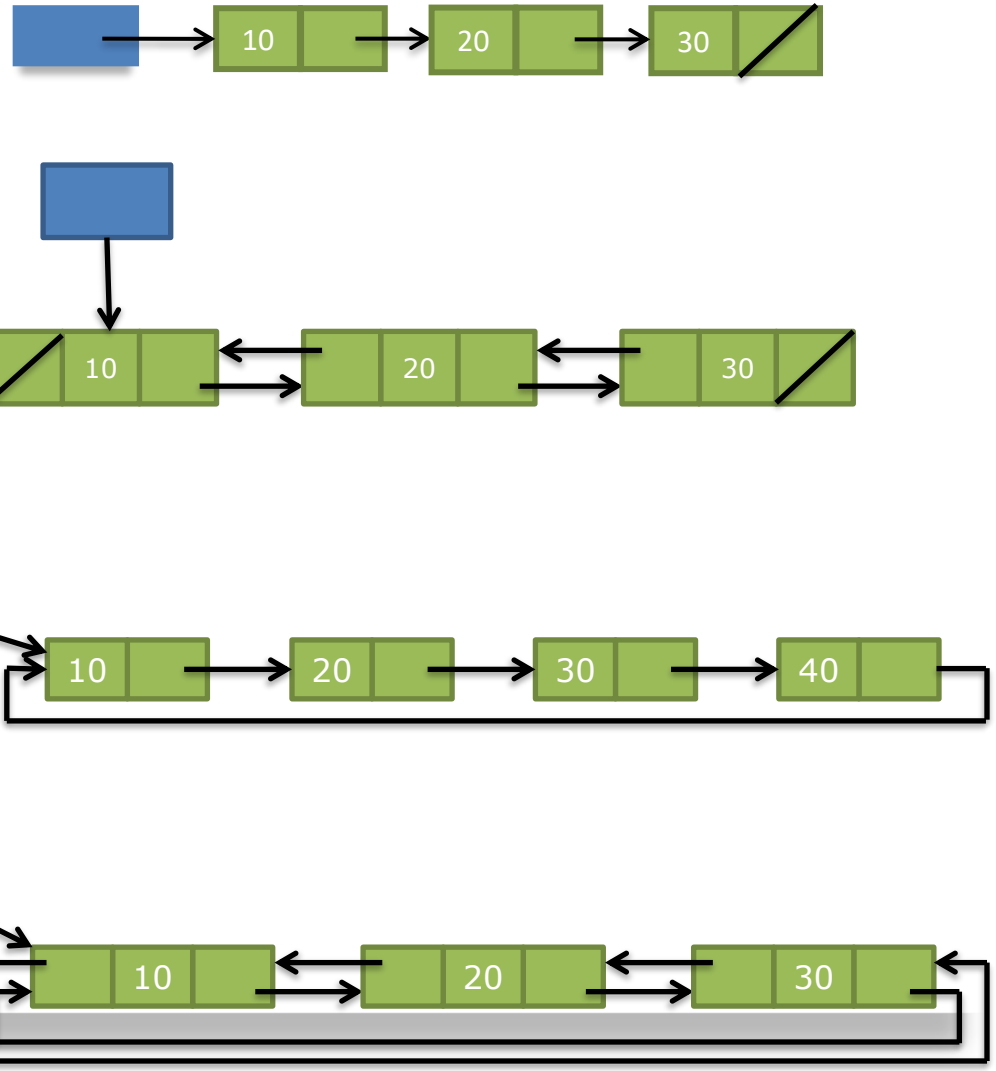


- Circular doubly linked lists
  - Last node has next pointer pointing to first node
  - First node has pre pointer pointing to last node



- **Display, Search, Size**: the last node's link is equal to head instead of **None**. The stop criteria needs to change.
- **Insert** and **Delete**: there is no special case at first or last position. The head node may need to update if the first node is affected.

Summary

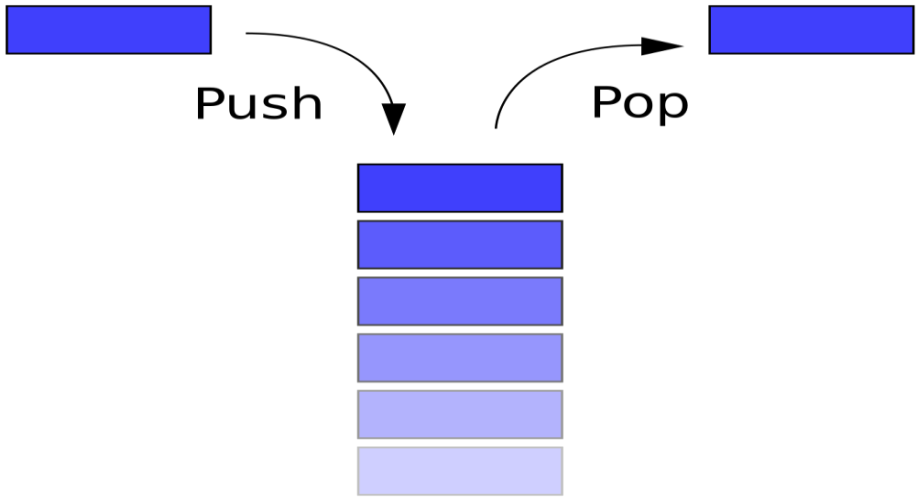


Topics

- Stack
- Queue
- Stacks and Queue Applications

What is a stack?

- A *stack* is a Last In, First Out (LIFO) data structure
- Anything added to the stack goes on the "top" of the stack
- Anything removed from the stack is taken from the "top" of the stack
- Things are removed in the reverse order from that in which they were inserted
- Can be implemented by list or linked list



Implementing a Stack class using Linked List

```
class Node:
    def __init__(self, data):
        self.data = data     # stores the value
        self.next = None     # points to next node
```

```
class Stack:
    def __init__(self):
        self.top = None       # points to top node
        self.size = 0         # tracks number of nodes
```

Core stack operations

`peek()`: Returns the top element without removing it

`push()`: Adds an element to the top of the stack

`pop()`: Removes and returns the top element

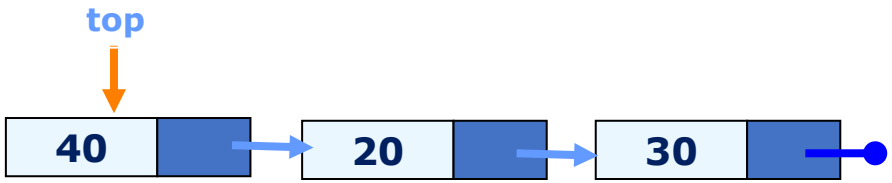
`is_empty()`: Checks if the stack is empty

`get_size()`: Returns the current size of the stack

Core stack operations: peek()

Returns the top element without removing it

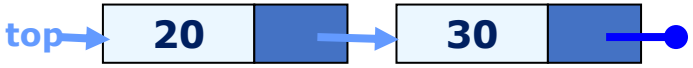
```
def peek(self):
    if self.is_empty():
        raise IndexError("Peek from empty stack")
    return self.top.data
```



- First checks if stack is empty using is_empty()
- If empty, raises an IndexError with a descriptive message
- If not empty, returns the data from the top node without removing it
- IndexError: https://docs.python.org/3/library/exceptions.html

Core stack operations: push()

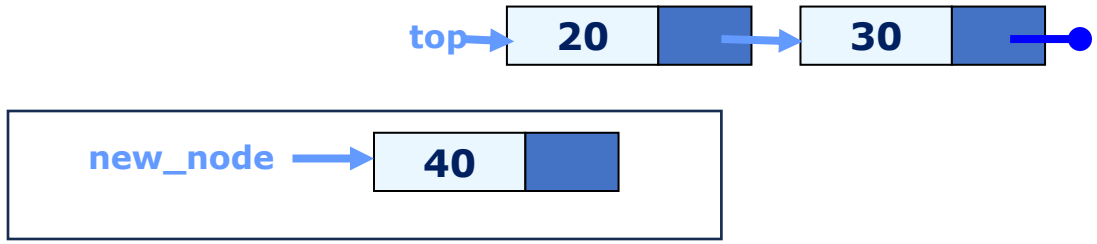
```
def push(self, data):
    new_node = Node(data)
    new_node.next = self.top
    self.top = new_node
    self.size += 1
```



- Step 1: Create new node with the data
- Step 2: Link new node to current top
- Step 3: Make new node the top
- Step 4: Increase size count

Core stack operations: push()

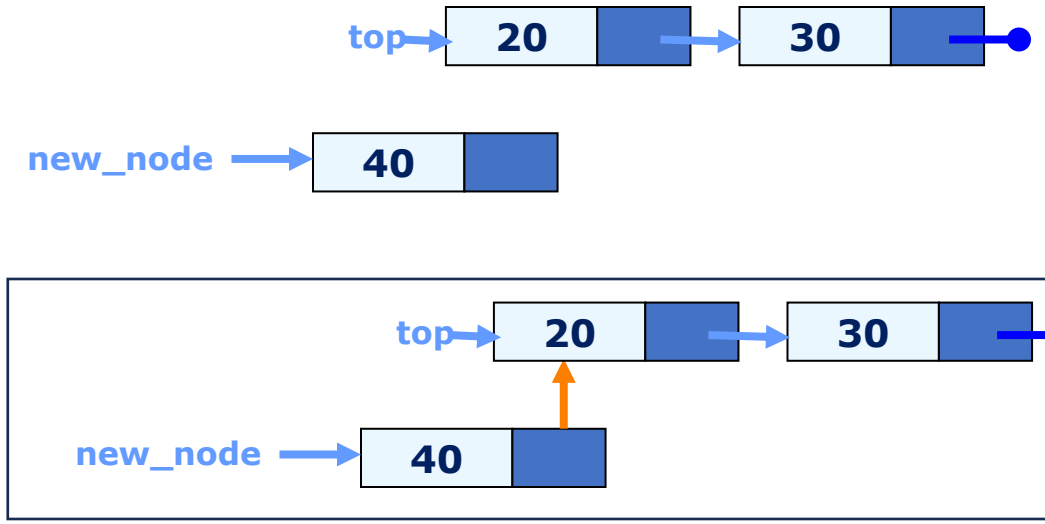
```
def push(self, data):
    new_node = Node(data)
    new_node.next = self.top
    self.top = new_node
    self.size += 1
```



- **Step 1: Create new node with the data**
- Step 2: Link new node to current top
- Step 3: Make new node the top
- Step 4: Increase size count

Core stack operations: push()

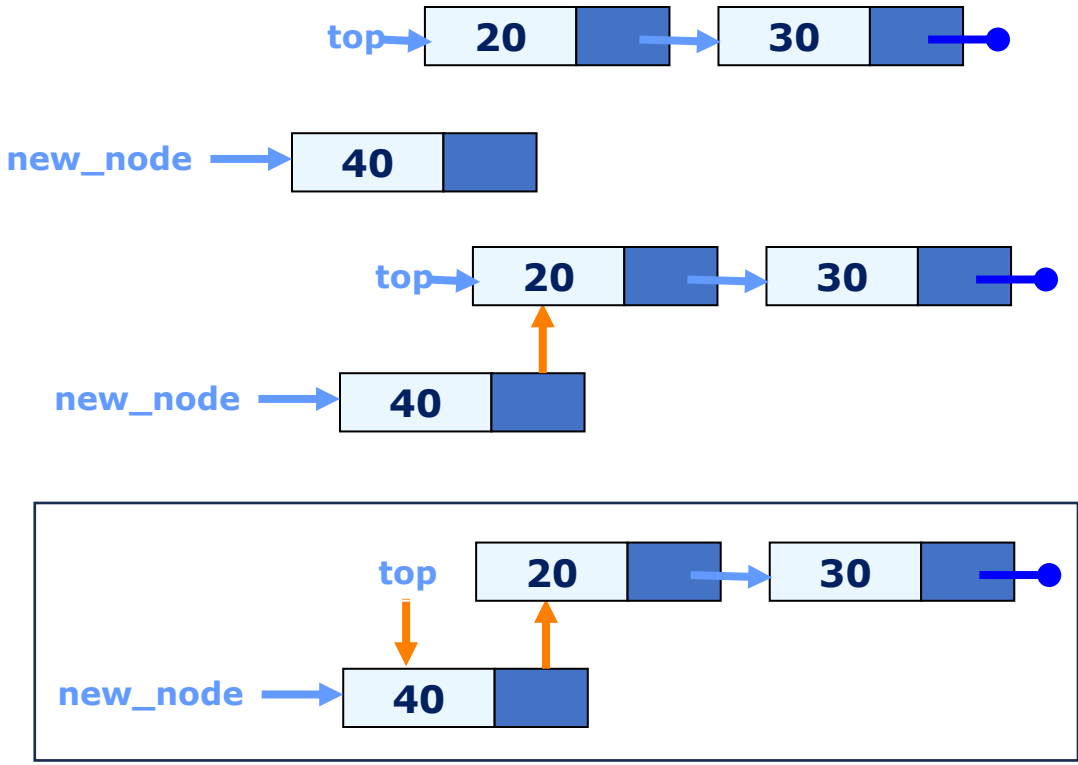
```
def push(self, data):
    new_node = Node(data)
    new_node.next = self.top
    self.top = new_node
    self.size += 1
```



- Step 1: Create new node with the data
- **Step 2: Link new node to current top**
- Step 3: Make new node the top
- Step 4: Increase size count

Core stack operations: push()

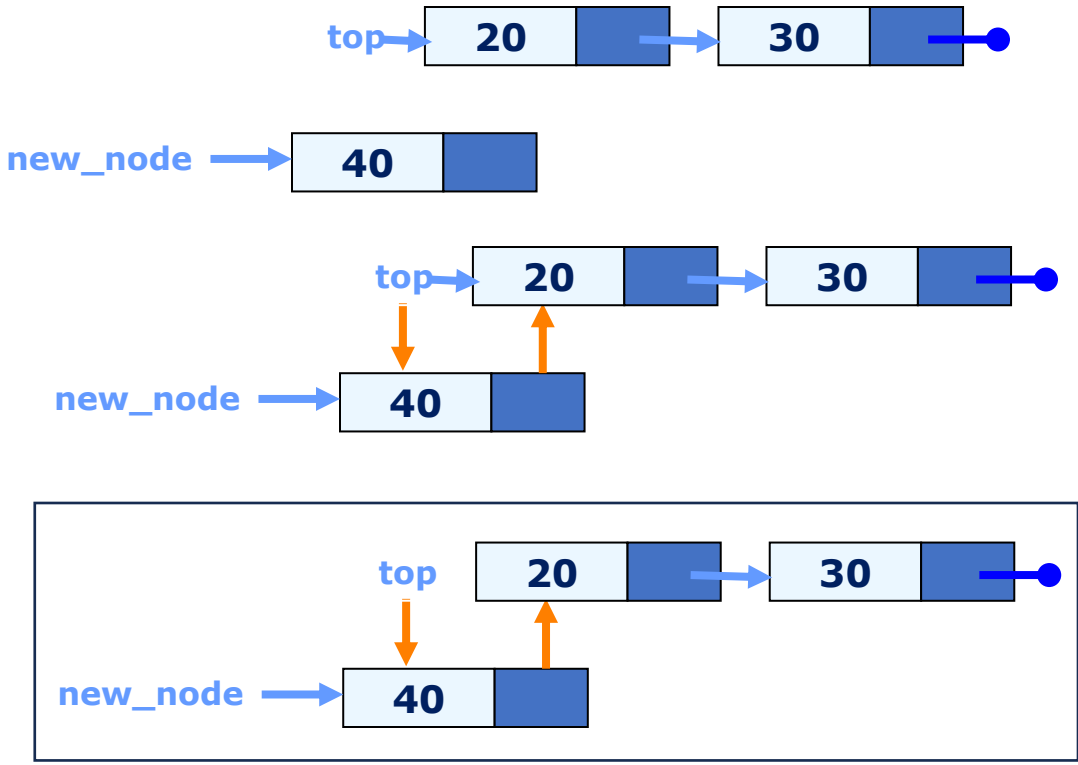
```
def push(self, data):
    new_node = Node(data)
    new_node.next = self.top
    self.top = new_node
    self.size += 1
```



- Step 1: Create new node with the data
- Step 2: Link new node to current top
- **Step 3: Make new node the top**
- Step 4: Increase size count

Core stack operations: push()

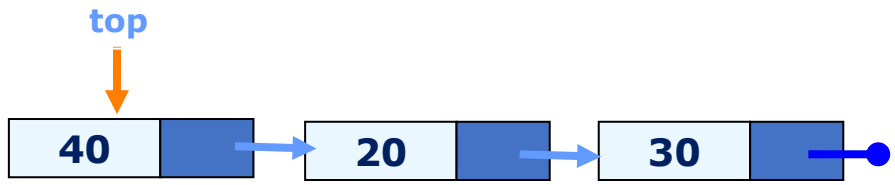
```
def push(self, data):
    new_node = Node(data)
    new_node.next = self.top
    self.top = new_node
    self.size += 1
```



- Step 1: Create new node with the data
- Step 2: Link new node to current top
- Step 3: Make new node the top
- **Step 4: Increase size count**

Core stack operations: pop()

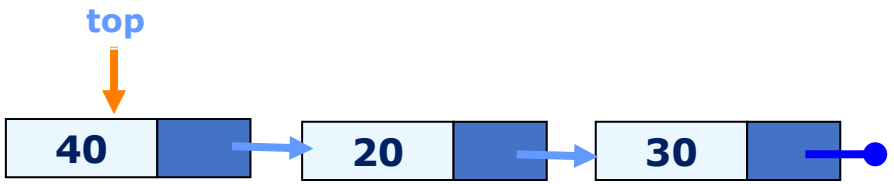
```
def pop(self):
    if self.is_empty():
        raise IndexError("Pop from empty stack")
    popped_node = self.top
    self.top = self.top.next
    self.size -= 1
    return popped_node.data
```

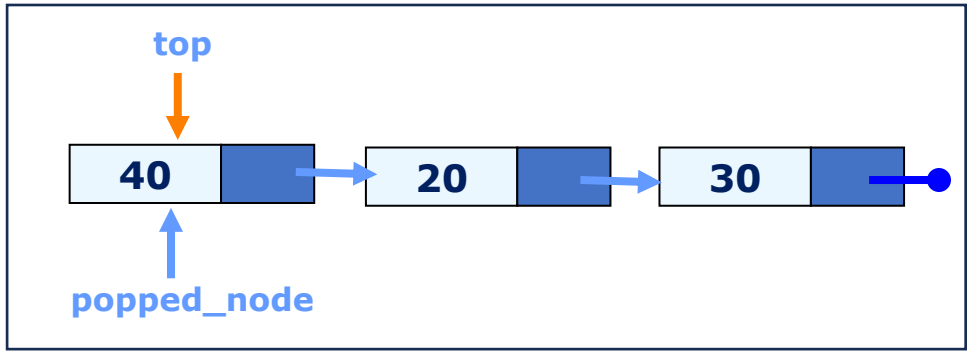


- Step 1: Check if stack is empty
- Step 2: Save current top node
- Step 3: Move top to next node
- Step 4: Decrease size count
- Step 5: Return the data from popped node

Core stack operations: pop()

```
def pop(self):
    if self.is_empty():
        raise IndexError("Pop from empty stack")
    popped_node = self.top
    self.top = self.top.next
    self.size -= 1
    return popped_node.data
```



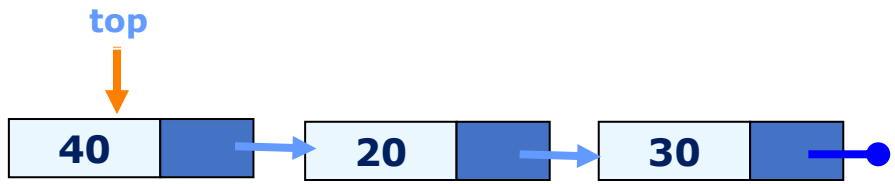


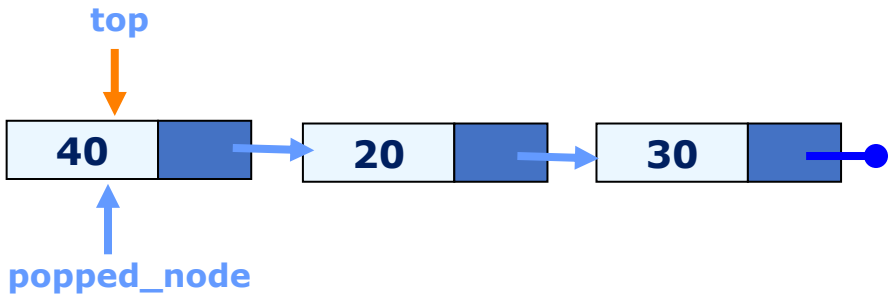
- Step 1: Check if stack is empty
- **Step 2: Save current top node**
- Step 3: Move top to next node
- Step 4: Decrease size count
- Step 5: Return the data from popped node

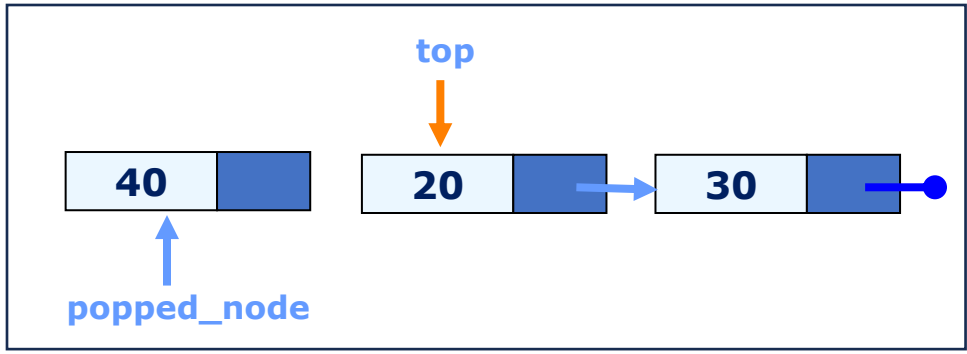
Core stack operations: pop()

```
def pop(self):
    if self.is_empty():
        raise IndexError("Pop from empty stack")
    popped_node = self.top
    self.top = self.top.next
    self.size -= 1
    return popped_node.data
```

- Step 1: Check if stack is empty
- Step 2: Save current top node
- **Step 3: Move top to next node**
- Step 4: Decrease size count
- Step 5: Return the data from popped node



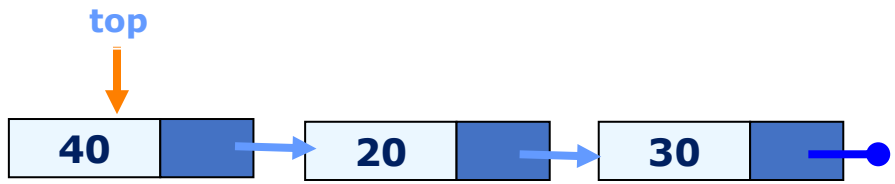


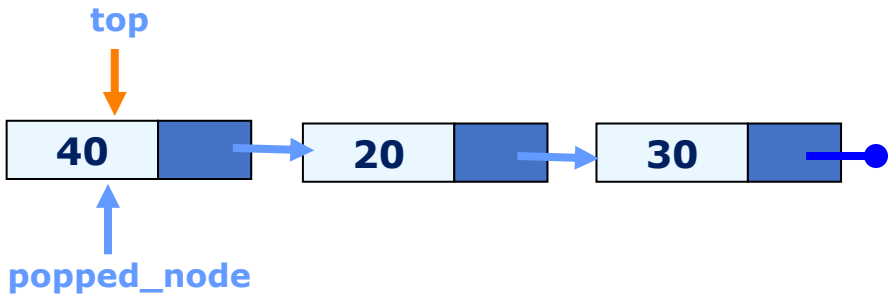


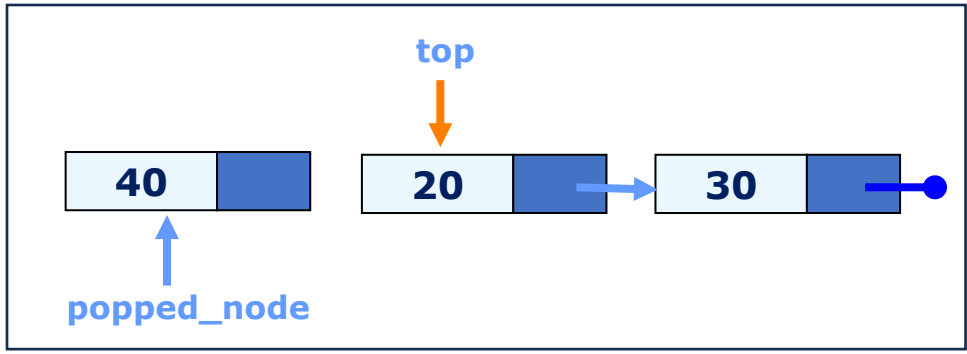
Core stack operations: pop()

```
def pop(self):
    if self.is_empty():
        raise IndexError("Pop from empty stack")
    popped_node = self.top
    self.top = self.top.next
    self.size -= 1
    return popped_node.data
```

- Step 1: Check if stack is empty
- Step 2: Save current top node
- Step 3: Move top to next node
- **Step 4: Decrease size count**
- **Step 5: Return the data from popped node**

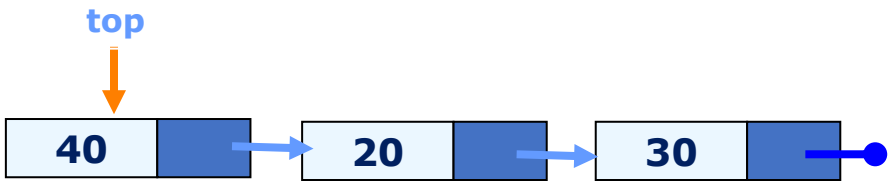






Core stack operations: isEmpty()

```
def is_empty(self):
    return self.top is None
```



- This function just checks if the stack is empty by seeing if top points to None.
  - Stack with multiple items:

    `top -> [40] -> [20] ->[30] ->None`

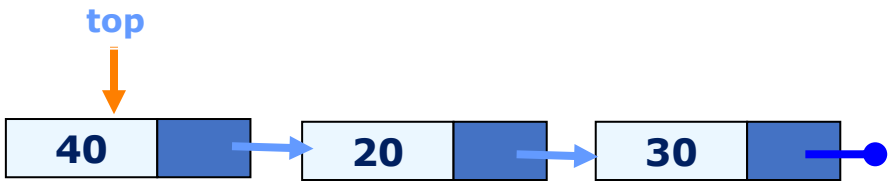
    `is_empty() returns False`
  - Empty Stack:

    `top -> None`

    `is_empty() returns True`

Core stack operations: get_size()

```
def get_size(self):
    return self.size
```



- This function returns the current value of `self.size`, which tracks the number of items in the stack.
  - Stack with multiple items:

    ```
    top -> [40] -> [20] ->[30] ->None
    size = 3
    get_size() returns 3
    ```

  - Empty Stack:

    ```
    top -> None
    size = 0
    get_size() returns 0
    ```

Working Example - 01

```
1 class Node:
2     def __init__(self, data):
3         self.data = data
4         self.next = None
5
6 class Stack:
7     def __init__(self):
8         self.top = None
9         self.size = 0
10
11    def peek(self):
12        if self.is_empty():
13            raise IndexError("Peek: Empty stack")
14        return self.top.data
15
16    def push(self, data):
17        new_node = Node(data)
18        new_node.next = self.top
19        self.top = new_node
20        self.size += 1
```

```
21    def pop(self):
22        if self.is_empty():
23            raise IndexError("Pop: Empty stack")
24        popped_node = self.top
25        self.top = self.top.next
26        self.size -= 1
27        return popped_node.data
28
29    def is_empty(self):
30        return self.top is None
31
32    def get_size(self):
33        return self.size
```

Working Example – 01 (cont.)

```
34 if __name__ == "__main__":
35
35     s = Stack() # Create a new stack
37
38     s.push(10) # Push some elements: 10, 20, and 30
39     s.push(20)
40     s.push(30)
41
42     print("Size:", s.get_size())    # Print the size. Should print 3
43
44     print("Top element:", s.peek())  # Print top element. Should print 30
45
46     print("Popped:", s.pop())       # Should print 30
47     print("Popped:", s.pop())       # Should print 20
48
49     print("New size:", s.get_size()) # Print new size. Should print 1
50
51     print("New top element:", s.peek()) # Print new top element. Should print 10
52
53     print("Is stack empty?", s.is_empty()) # Check if empty. Should print False
54
55     print("Popped:", s.pop())       # Pop last element. Should print 10
56
57     print("Is stack empty now?", s.is_empty()) # Check if empty again. Should print True
```

Working Example – 01 (cont.)

Method Call Translation

- When you call `s.push(10)`:
- Python recognizes you're calling the `push()` method on instance `s` of the Stack class.
- It translates this to `Stack.push(s, 10)`
- `s` becomes the value of the `self` parameter
- `10` becomes the value of the data parameter

```
1 class Node:
2     def __init__(self, data):
3         self.data = data
4         self.next = None
5
6 class Stack:
7     def __init__(self):
8         self.top = None
9         self.size = 0
10
11    def push(self, data):
12        new_node = Node(data)
13        new_node.next = self.top
14        self.top = new_node
15        self.size += 1
16
17 if __name__ == "__main__":
18     # Create a stack instance 's'
19        s = Stack()
20
21        s.push(10)
22        s.push(20)
23        s.push(30)
```

Working Example – 01 (cont.)

Method Call Translation

- When you call `s.push(10):`
- Python recognizes you're calling the `push()` method on instance `s` of the Stack class.
- It translates this to `Stack.push(s, 10)`
- `s` becomes the value of the `self` parameter
- `10` becomes the value of the data parameter

Inside the push Method

- Within the `push` method:
- `self` refers to the stack instance `s`
- `new_node = Node(data)` creates a new node with value `10`
- `self.top` refers to the top node of instance `s`
- `self.size` refers to the size counter of instance `s`

```
1   class Node:
2       def __init__(self, data):
3           self.data = data
4           self.next = None
5
6   class Stack:
7       def __init__(self):
8           self.top = None
9           self.size = 0
10
11      def push(self, data):
12          new_node = Node(data)
13          new_node.next = self.top
14          self.top = new_node
15          self.size += 1
16
17   if __name__ == "__main__":
18      # Create a stack instance 's'
19          s = Stack()
20
21          s.push(10)
22          s.push(20)
23          s.push(30)
```

Working Example – 01 (cont.)

Method Call Translation

- When you call `s.push(10):`
- Python recognizes you're calling the `push()` method on instance `s` of the Stack class.
- It translates this to `Stack.push(s, 10)`
- `s` becomes the value of the `self` parameter
- `10` becomes the value of the data parameter

Inside the push Method

- Within the `push` method:
- `self` refers to the stack instance `s`
- `new_node = Node(data)` creates a new node with value `10`
- `self.top` refers to the top node of instance `s`
- `self.size` refers to the size counter of instance `s`

Operations in push

- The method performs these operations:
- Creates a new node: `new_node = Node(data)` creates `Node(10)`
- Links to current top: `new_node.next = self.top` links to `s`'s current top
- Updates top: `self.top = new_node` updates `s`'s top pointer
- Increments size: `self.size += 1` increases `s`'s size counter

```
1   class Node:
2       def __init__(self, data):
3           self.data = data
4           self.next = None
5
6   class Stack:
7       def __init__(self):
8           self.top = None
9           self.size = 0
10
11      def push(self, data):
12          new_node = Node(data)
13          new_node.next = self.top
14          self.top = new_node
15          self.size += 1
16
17   if __name__ == "__main__":
18      # Create a stack instance 's'
19          s = Stack()
20
21          s.push(10)
22          s.push(20)
23          s.push(30)
```

Working Example – 01 (cont.)

Method Call Translation

- When you call `s.push(10):`
- Python recognizes you're calling the `push()` method on instance `s` of the Stack class.
- It translates this to `Stack.push(s, 10)`
- `s` becomes the value of the `self` parameter
- `10` becomes the value of the data parameter

Inside the push Method

- Within the `push` method:
- `self` refers to the stack instance `s`
- `new_node = Node(data)` creates a new node with value `10`
- `self.top` refers to the top node of instance `s`
- `self.size` refers to the size counter of instance `s`

Operations in push

- The method performs these operations:
- Creates a new node: `new_node = Node(data)` creates `Node(10)`
- Links to current top: `new_node.next = self.top` links to `s`'s current top
- Updates top: `self.top = new_node` updates s's top pointer
- Increments size: `self.size += 1` increases s's size counter

```
1   class Node:
2       def __init__(self, data):
3           self.data = data
4           self.next = None
5
6   class Stack:
7       def __init__(self):
8           self.top = None
9           self.size = 0
10
11      def push(self, data):
12          new_node = Node(data)
13          new_node.next = self.top
14          self.top = new_node
15          self.size += 1
16
17   if __name__ == "__main__":
18      # Create a stack instance 's'
19          s = Stack()
20
21          s.push(10)
22          s.push(20)
23          s.push(30)
```

Implementing a Stack class using Linked List

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
```

```
class LinkedList:
    def __init__(self):
        self.size = 0
        self.head = None
        self.tail = None
```

```
class Stack:
    def __init__(self):
        self.ll = LinkedList()
```

Core stack operations

`peek()`: Returns the top element without removing it

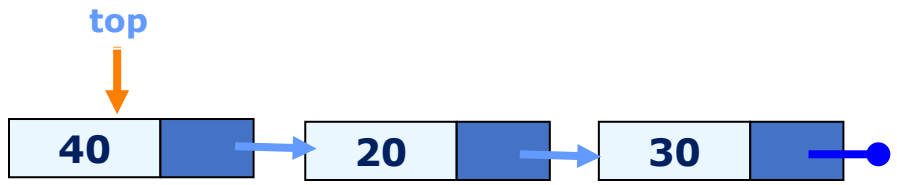
`push()`: Adds an element to the top of the stack

`pop()`: Removes and returns the top element

`is_empty()`: Checks if the stack is empty

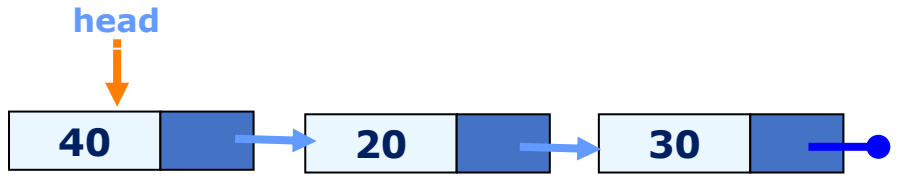
`get_size()`: Returns the current size of the stack

Core stack operations: peek()



Previous version

```
def peek(self):
    if self.is_empty():
        raise IndexError("Peek from empty stack")
    return self.top.data
```

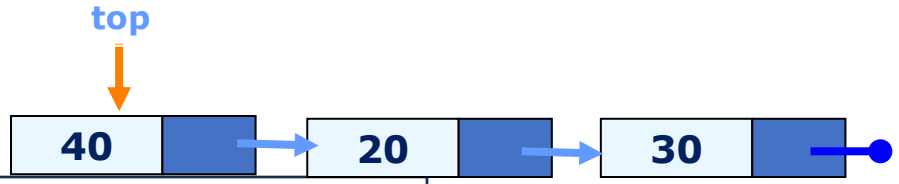


New Version

```
def peek(self):
    if self.is_empty():
        raise IndexError("Peek from empty stack")
    return self.ll.head.data
```

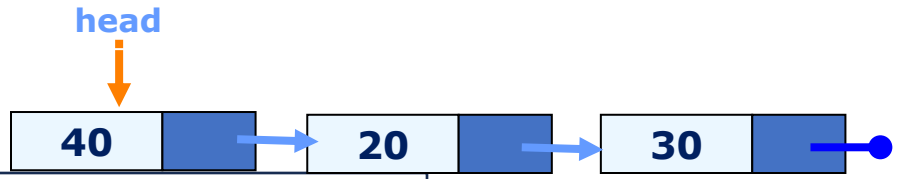
Core stack operations: push()

Previous version




```
def push(self, data):
    new_node = Node(data)
    new_node.next = self.top
    self.top = new_node
    self.size += 1
```

New version

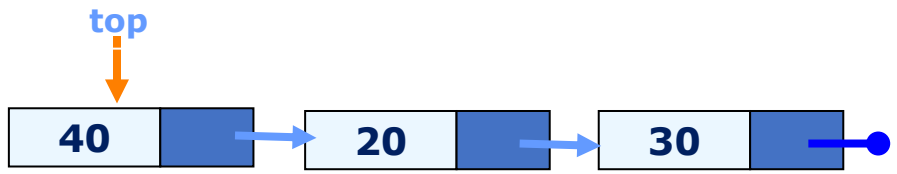



```
def push(self, data):
        self.ll.insert_node(0, data)
```

Core stack operations: pop()

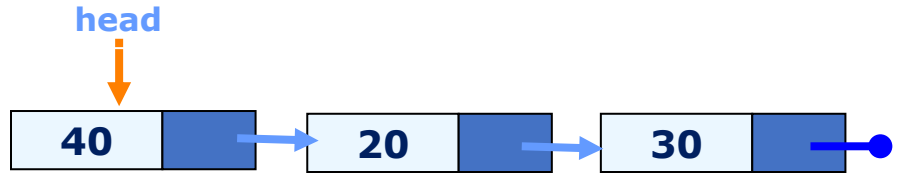
Previous version

```
def pop(self):
    if self.is_empty():
        raise IndexError("Pop from empty stack")
    popped_node = self.top
    self.top = self.top.next
    self.size -= 1
    return popped_node.data
```

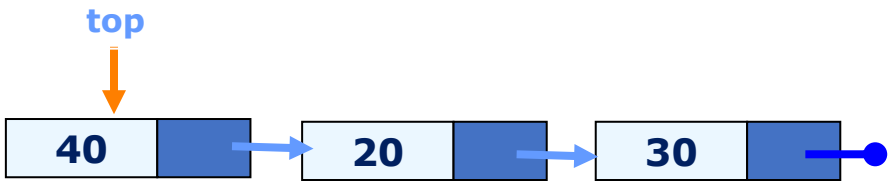


New version

```
def pop(self):
    if self.isEmpty():
        raise IndexError("Pop from empty stack")
    data = self.ll.head.data
    self.ll.remove_node(0)
    return data
```



Core stack operations: isEmpty()



Previous version

```
def is_empty(self):
    return self.top is None
```

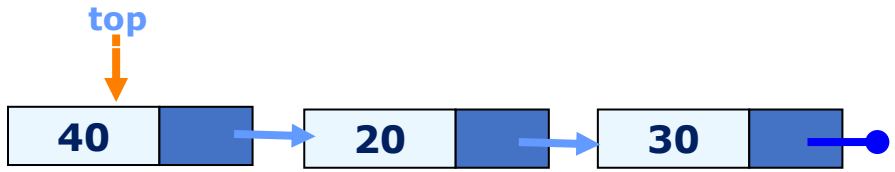
New Version

```
def is_empty(self):
    return self.ll.size == 0
```

Core stack operations: get_size()

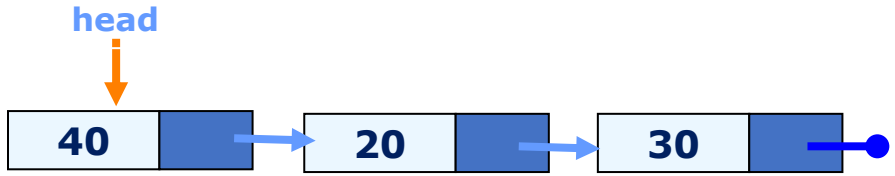
Previous version

```
def get_size(self):
    return self.size
```



New Version

```
def get_size(self):
    return self.ll.size
```



Working Example – 02

```
1  class ListNode:
2      def __init__(self, item):
3          self.item = item
4          self.next = None
5
6  class LinkedList:
7      def __init__(self):
8          self.size = 0
9          self.head = None
10         self.tail = None
11
12     def find_node(self, index):
13         if index < 0 or index >= self.size:
14             return None
15         temp = self.head
16         for _ in range(index):
17             temp = temp.next
18         return temp
19
20     def remove_node(self, index):
21         if index < 0 or index >= self.size:
22             return -1
23         if index == 0:
24             self.head = self.head.next
25             if self.size == 1:
26                 self.tail = None
27         else:
28             prev = self.find_node(index - 1)
29             prev.next = prev.next.next
30             if index == self.size - 1:
31                 self.tail = prev
32         self.size -= 1
33         return 0
```

```
34     def insert_node(self, index, value):
35         if index < 0 or index > self.size:
36             return -1
37         new_node = ListNode(value)
38         if index == 0:
39             new_node.next = self.head
40             self.head = new_node
41             if self.size == 0:
42                 self.tail = new_node
43         elif index == self.size:
44             self.tail.next = new_node
45             self.tail = new_node
46         else:
47             prev = self.find_node(index - 1)
48             new_node.next = prev.next
49             prev.next = new_node
50         self.size += 1
51         return 0
```

Working Example – 02 (cont.)

```
52  class Stack:
53      def __init__(self):
54          self.ll = LinkedList()
55
56      def push(self, item):
57          self.ll.insert_node(0, item)
58
59      def pop(self):
60          if self.is_empty():
61              raise IndexError("Pop from empty stack")
62          item = self.ll.head.item
63          self.ll.remove_node(0)
64          return item
65
66      def peek(self):
67          if self.is_empty():
68              raise IndexError("Pop from empty stack")
69          return self.ll.head.item
70
71      def is_empty(self):
72          return self.ll.size == 0
73
74      def get_size(self):
75          return self.ll.size
```

Working Example – 02 (cont.)

```
76 if __name__ == "__main__":
77
78     s = Stack()  # Create a new stack
79
80     s.push(10)   # Push some elements: 10, 20, and 30
81     s.push(20)
82     s.push(30)
83
84     print("Size:", s.get_size())      # Print the size. Should print 3
85
86     print("Top element:", s.peek()) # Print top element. Should print 30
87
88     print("Popped:", s.pop())      # Should print 30
89     print("Popped:", s.pop())      # Should print 20
90
91     print("New size:", s.get_size())  # Print new size. Should print 1
92
93     print("New top element:", s.peek()) # Print new top element. Should print 10
94
95     print("Is stack empty?", s.is_empty()) # Check if empty. Should print False
96
97     print("Popped:", s.pop())      # Pop last element. Should print 10
98
99     print("Is stack empty now?", s.is_empty()) # Check if empty again. Should print True
```

Stacks in Computer Science

- **Operating Systems:**
  - Function call stack to keep track of active function calls
  - Undo/redo functionality in text editors
  - Memory stack for local variable storage and function execution

- **Programming:**
  - Backtracking algorithms (e.g., solving mazes, recursion, depth-first search)
  - Evaluating expressions (e.g., parsing mathematical expressions using postfix notation)
  - Managing history (e.g., web browser back/forward navigation)

- **Real-World Examples:**
  - A stack of plates in a cafeteria (Last In, First Out - LIFO)
  - Piling books on a table (you take the topmost book first)

Some uses of stacks

1. **Any sort of nesting (such as tracking parentheses)**
   - Stacks help in maintaining nested structures like parentheses, HTML tags, and recursive function calls, ensuring correct opening and closing sequences.

2. **Evaluating arithmetic expressions (and other sorts of expressions)**
   - Stacks are used in expression evaluation (e.g., postfix or infix to postfix conversion), efficiently managing operator precedence and execution order.

3. **Implementing function or method calls (built-in to virtually every programming language)**
   - Programming languages internally use a call stack to manage function calls, storing local variables and returning to the correct execution point after function completion.

Some uses of stacks

4. **Keeping track of choices yet to be made, or steps yet to be done**
   - Stacks assist in decision-making processes, such as depth-first search (DFS) in graphs, where choices for the next step are pushed and popped as needed.

5. **Keeping track of previous choices (as in search backtracking algorithms)**
   - Stacks facilitate backtracking algorithms (e.g., maze solving, Sudoku), allowing the program to revert to previous states when encountering dead ends.