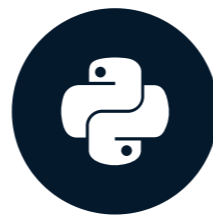


Welcome!

DATA STRUCTURES AND ALGORITHMS IN PYTHON



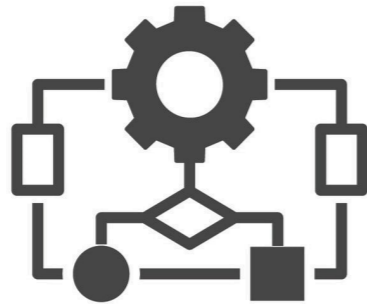
Miriam Antona
Software Engineer

The importance of algorithms and data structures

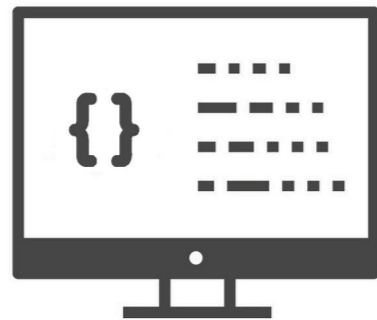
- Data structures and algorithms enable us to
 - solve everyday problems
 - using efficient code
- The course could be taught in any programming language

Algorithms and data structures

- **Algorithm:** set of instructions that solve a problem
 1. *Design*
- **Data structures:** hold and manipulate data when we execute an algorithm
 - **Advanced data structures:** linked lists, stacks, queues...

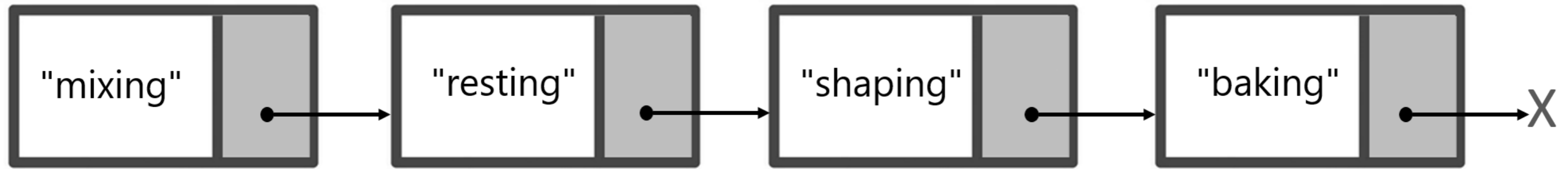


2. *Code*



Linked lists

bread_steps



- Sequence of data connected through links

Linked lists - structure



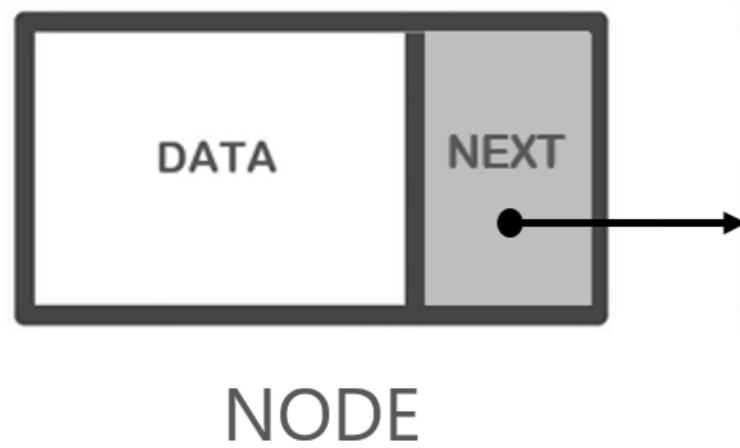
NODE

Linked lists - structure

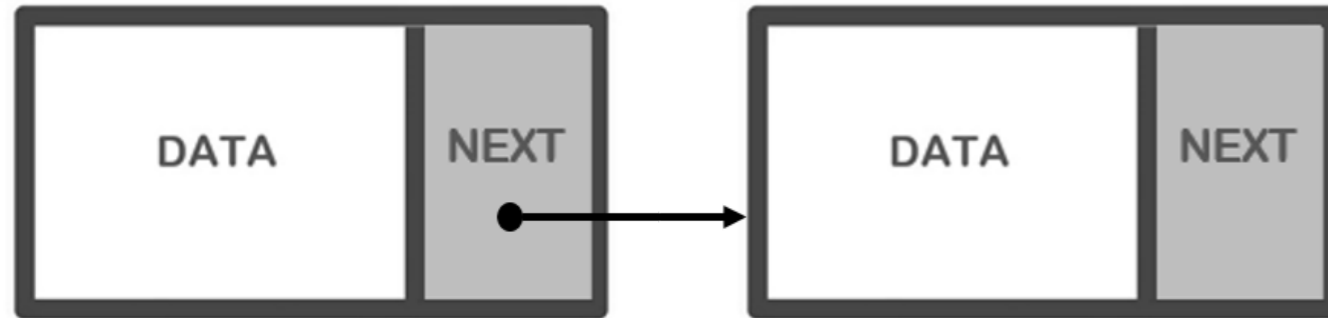


NODE

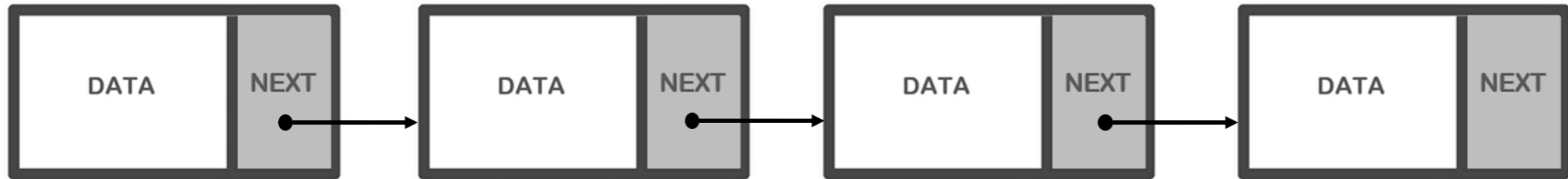
Linked lists - structure



Linked lists - structure



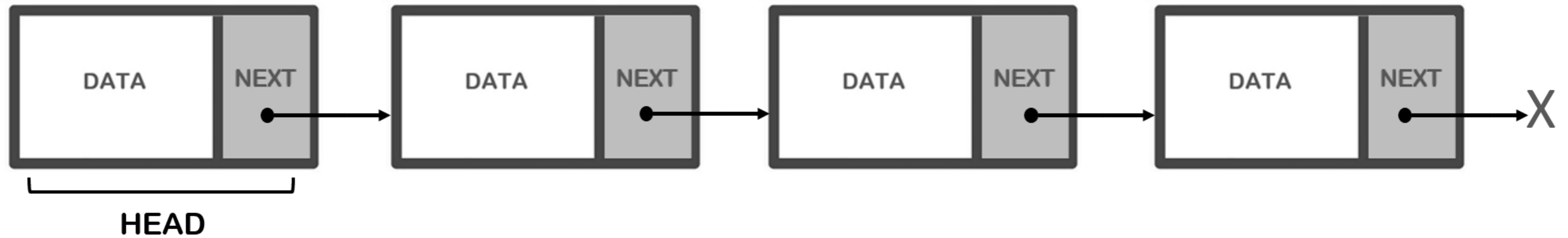
Linked lists - structure



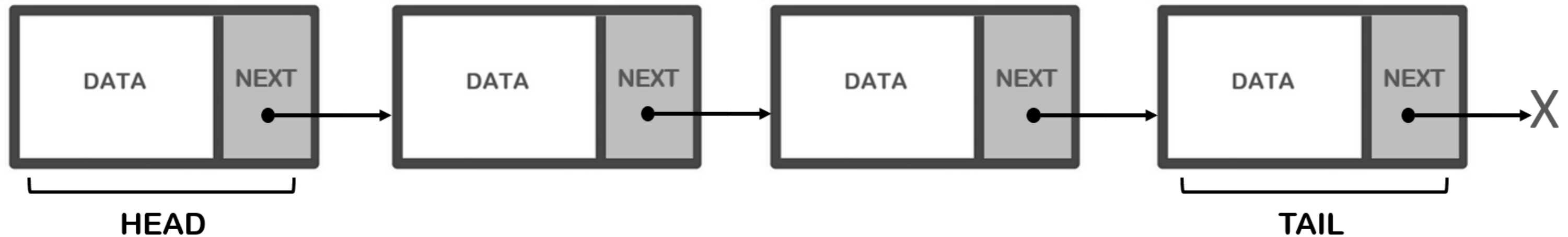
Linked lists - structure



Linked lists - structure



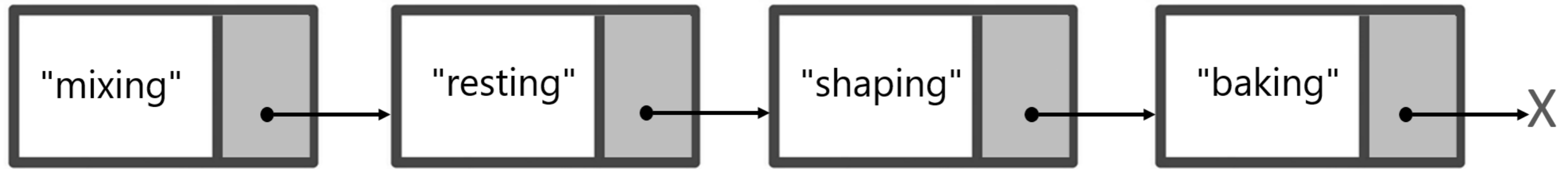
Linked lists - structure



- Data doesn't need to be stored in contiguous blocks of memory
- Data can be located in any available memory address

Singly linked lists

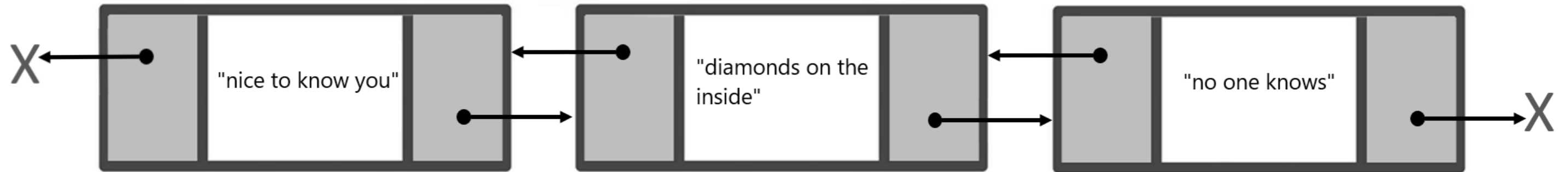
bread_steps



- One link: singly linked list

Doubly linked lists

my_favourite_playlist



- Two links in either direction: **doubly linked list**

Linked lists - real uses

- Implement other data structures:
 - stacks
 - queues
 - graphs
- Access information by navigating backward and forward
 - web browser
 - music playlist

Linked lists - Node class

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

Linked lists - LinkedList class

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

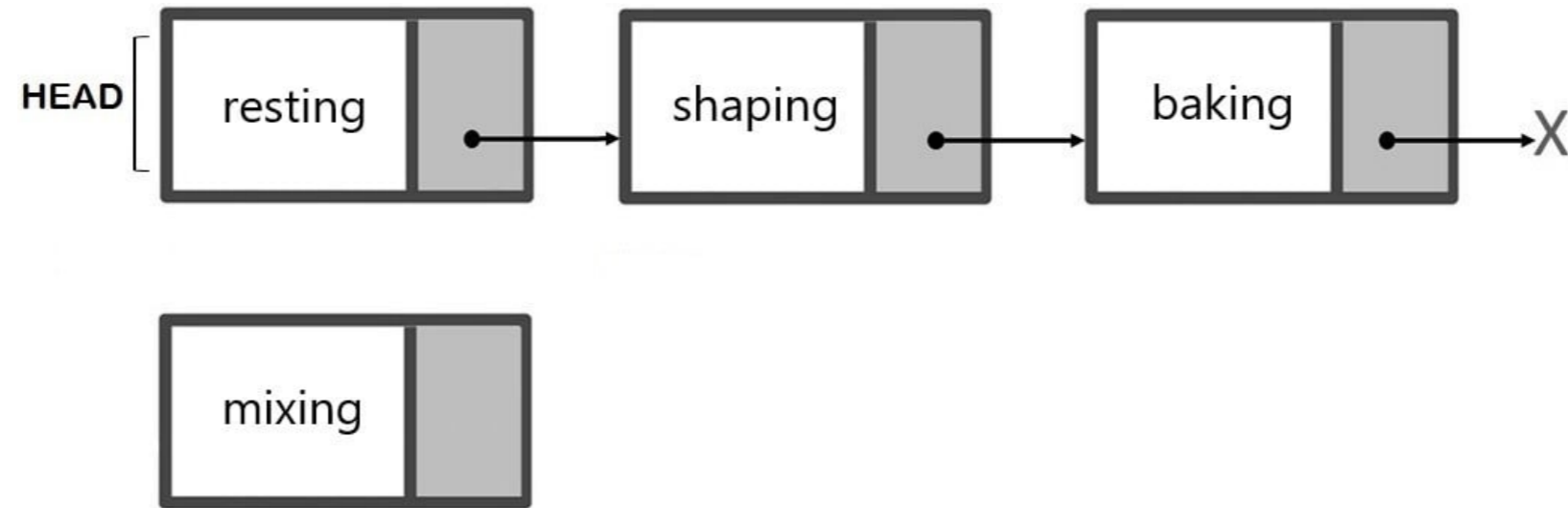
Linked lists - methods

- `insert_at_beginning()`
- `remove_at_beginning()`
- `insert_at_end()`
- `remove_at_end()`
- `insert_at()`
- `remove_at()`
- `search()`
- ...

Linked lists - insert_at_beginning

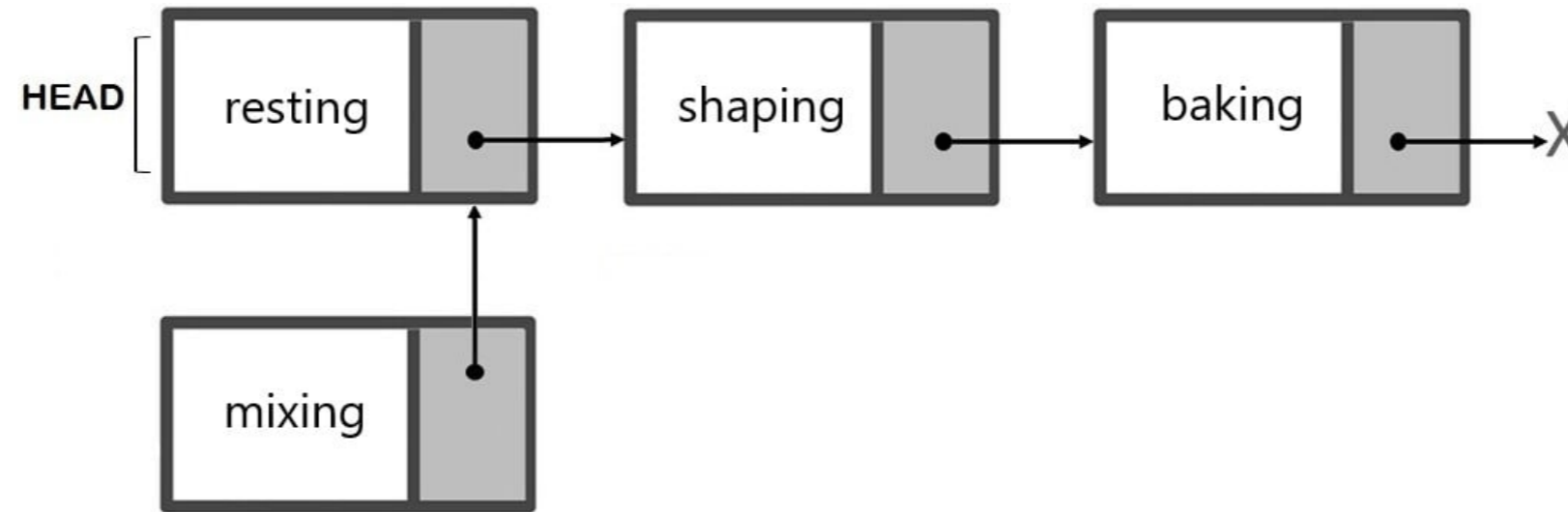


Linked lists - insert_at_beginning



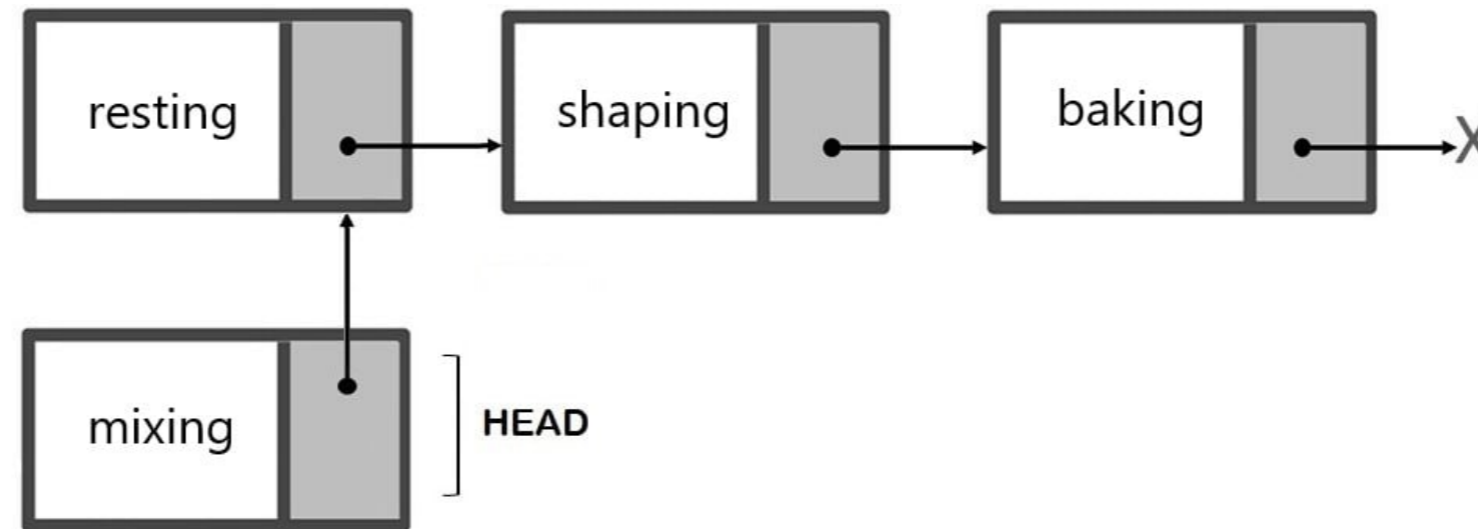
```
def insert_at_beginning(self, data):  
    new_node = Node(data)  
    if self.head:
```

Linked lists - insert_at_beginning



```
def insert_at_beginning(self, data):  
    new_node = Node(data)  
    if self.head:  
        new_node.next = self.head
```

Linked lists - insert_at_beginning



```
def insert_at_beginning(self, data):  
    new_node = Node(data)  
    if self.head:  
        new_node.next = self.head  
        self.head = new_node
```

Linked lists - insert_at_beginning



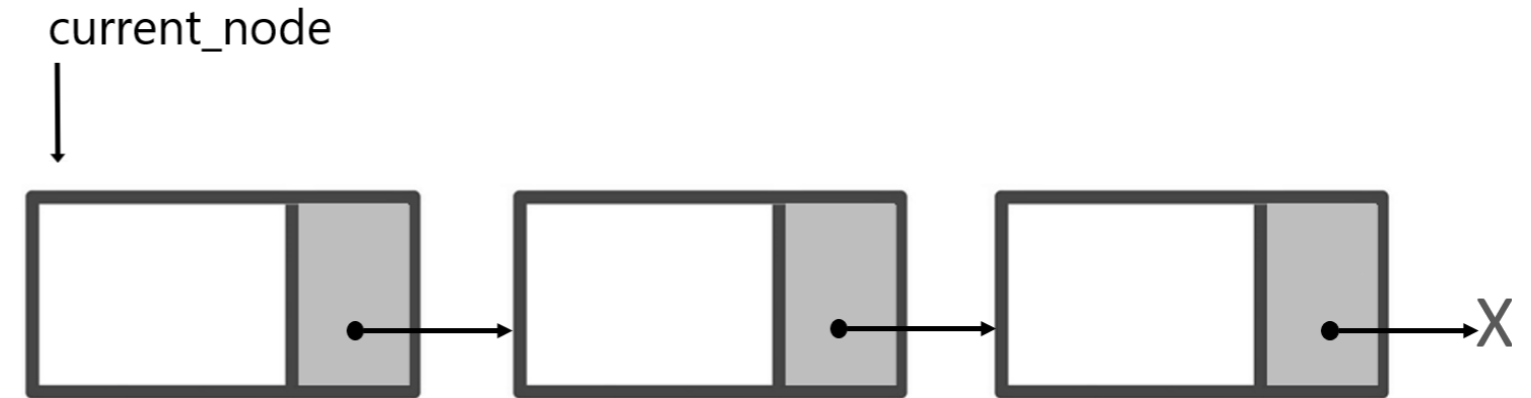
```
def insert_at_beginning(self, data):  
    new_node = Node(data)  
    if self.head:  
        new_node.next = self.head  
        self.head = new_node  
    else:  
        self.tail = new_node  
        self.head = new_node
```

Linked lists - insert_at_end

```
def insert_at_end(self, data):  
    new_node = Node(data)  
    if self.head:  
        self.tail.next = new_node  
        self.tail = new_node  
    else:  
        self.head = new_node  
        self.tail = new_node
```

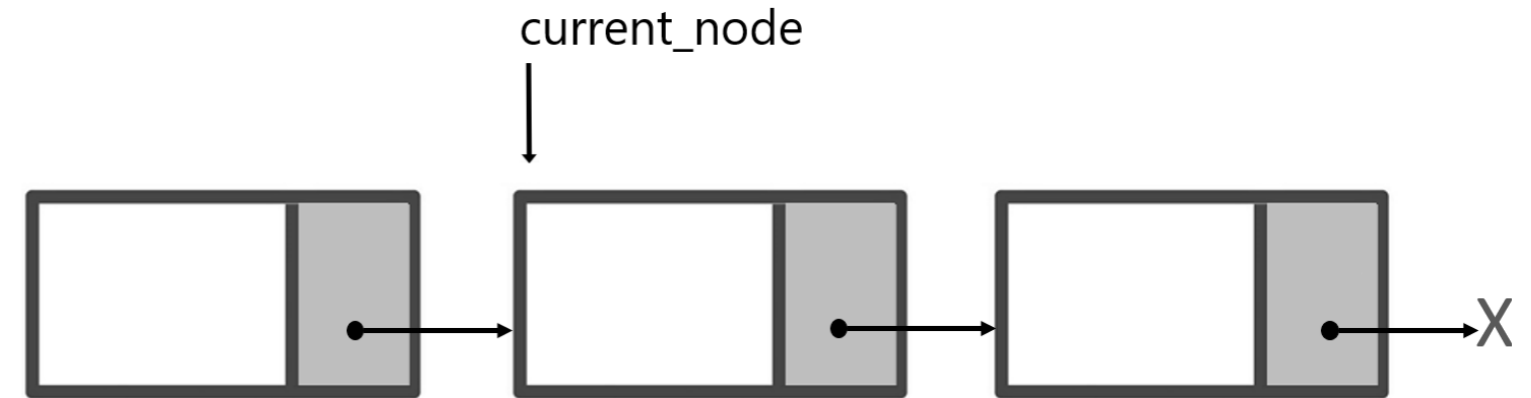
Linked lists - search

```
def search(self, data):  
    current_node = self.head  
    while current_node:  
        if current_node.data == data:  
            return True
```



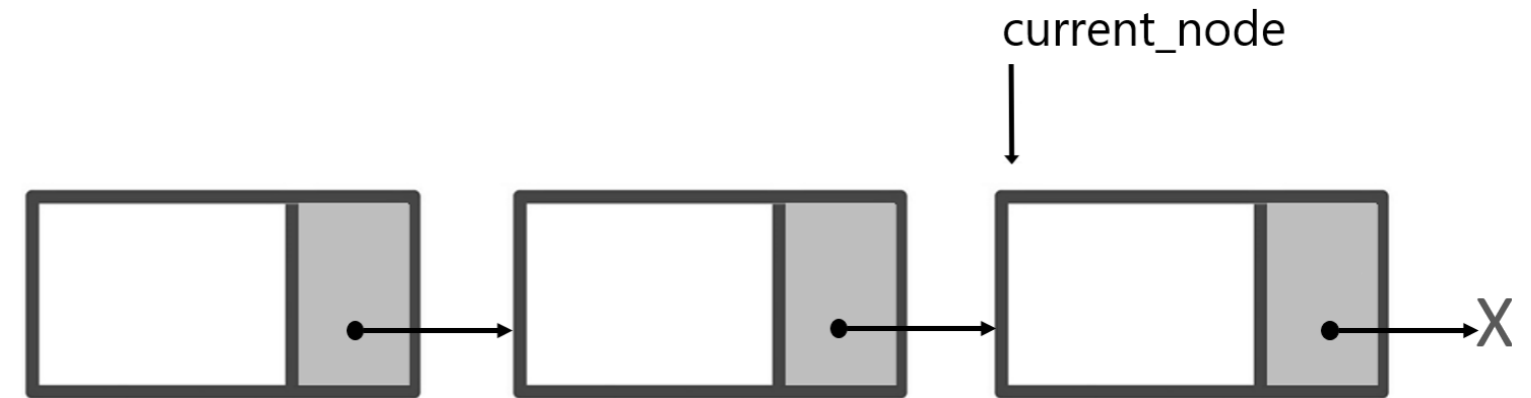
Linked lists - search

```
def search(self, data):  
    current_node = self.head  
    while current_node:  
        if current_node.data == data:  
            return True  
        else:  
            current_node = current_node.next
```

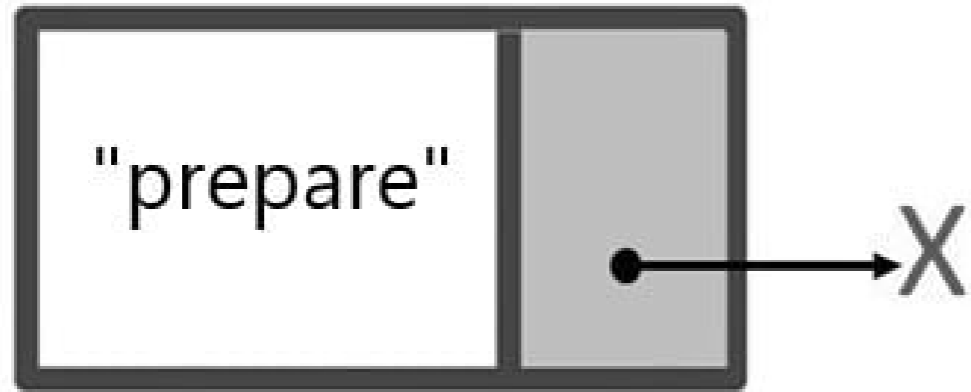


Linked lists - search

```
def search(self, data):  
    current_node = self.head  
    while current_node:  
        if current_node.data == data:  
            return True  
        else:  
            current_node = current_node.next  
    return False
```

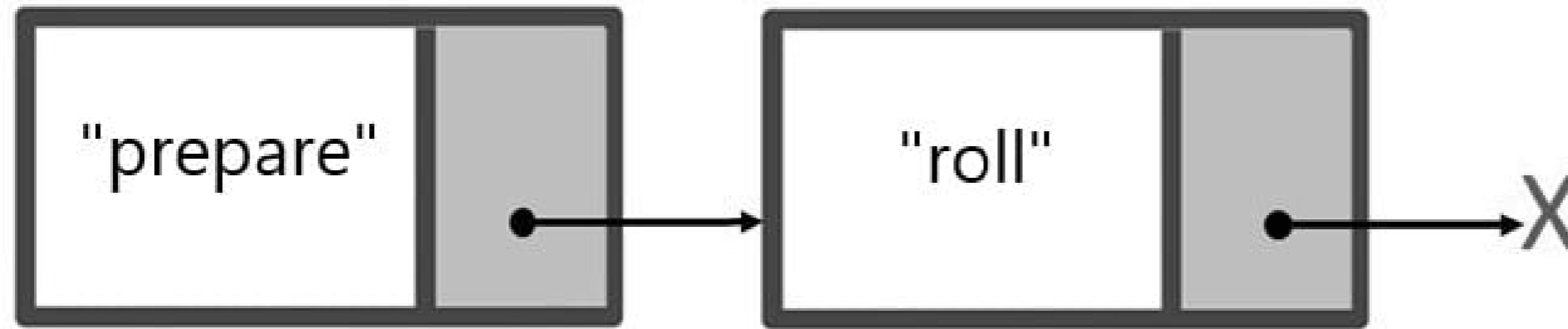


Linked lists - example



```
sushi_preparation = LinkedList()  
sushi_preparation.insert_at_end("prepare")
```

Linked lists - example



```
sushi_preparation = LinkedList()
sushi_preparation.insert_at_end("prepare")
sushi_preparation.insert_at_end("roll")
```

Linked lists - example



```
sushi_preparation = LinkedList()
sushi_preparation.insert_at_end("prepare")
sushi_preparation.insert_at_end("roll")
sushi_preparation.insert_at_beginning("assemble")
```

Linked lists - example



```
sushi_preparation.search("roll")
```

True

```
sushi_preparation.search("mixing")
```

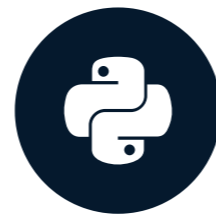
False

Let's practice!

DATA STRUCTURES AND ALGORITHMS IN PYTHON

Understanding Big O Notation

DATA STRUCTURES AND ALGORITHMS IN PYTHON

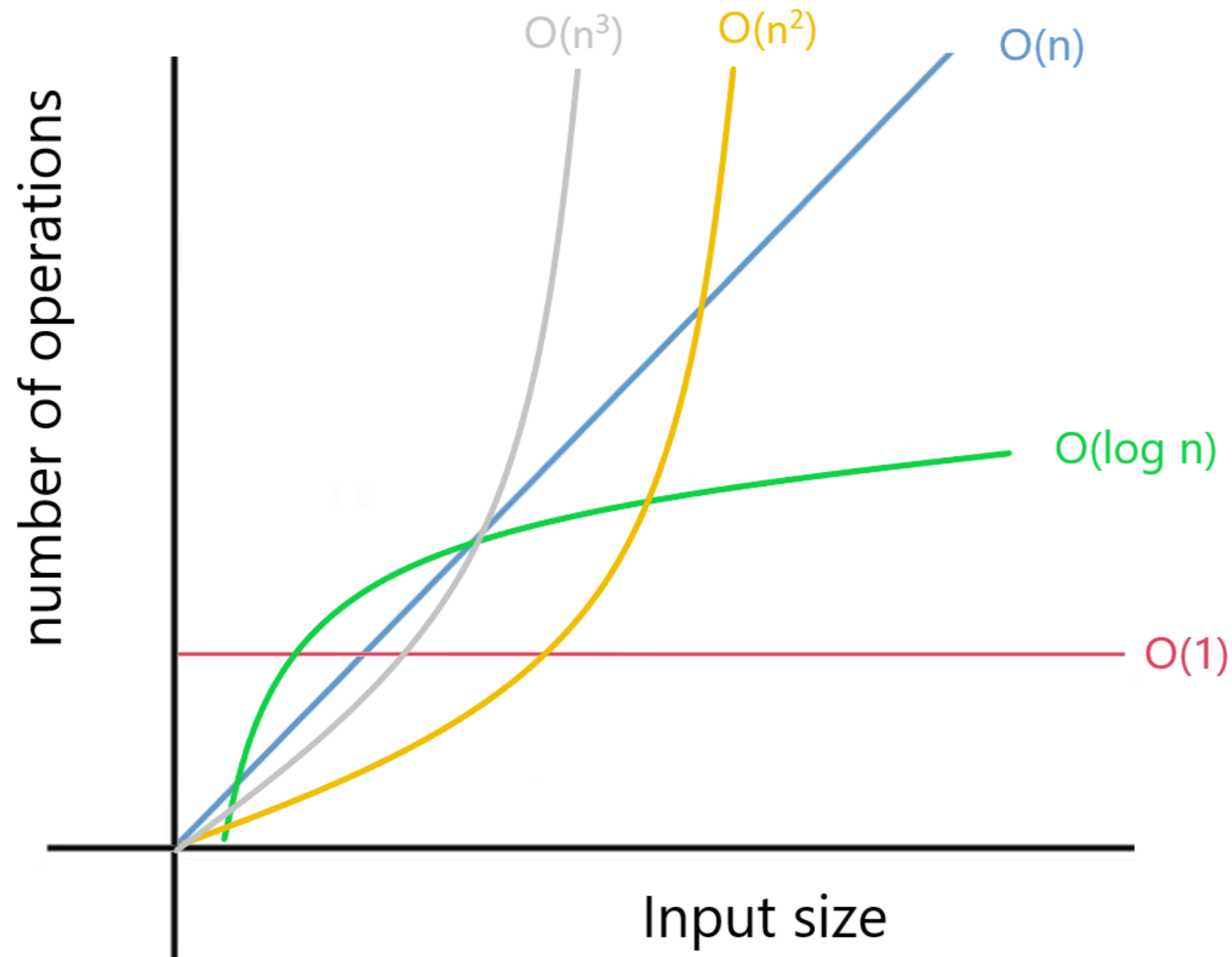


Miriam Antona
Software Engineer

Big O Notation

- Measures the **worst-case complexity** of an algorithm
 - **Time complexity**: time taken to run completely
 - **Space complexity**: extra memory space
- Doesn't use seconds/bytes
 - Different results depending on the hardware
- **Mathematical expressions**: $O(1)$, $O(n)$, $O(n^2)$...

Big O Notation



$O(1)$

```
colors = ['green', 'yellow', 'blue', 'pink']
```

```
def constant(colors):  
    print(colors[2])
```

```
constant(colors)
```

```
blue
```

$O(1)$

```
colors = ['green', 'yellow', 'blue', 'pink', 'black', 'white', 'purple', 'red']
```

```
def constant(colors):  
    print(colors[2]) #  $O(1)$ 
```

```
constant(colors)
```

blue

$O(n)$

```
colors = ['green', 'yellow', 'blue', 'pink']
```

```
def linear(colors):  
    for color in colors:  
        print(color)
```

```
linear(colors)
```

```
green  
yellow  
blue  
pink
```

$O(n)$

```
colors = ['green', 'yellow', 'blue', 'pink'] # n=4
```

```
def linear(colors):  
    for color in colors:  
        print(color)    # O(4)
```

```
linear(colors)
```

- `n=4` : 4 operations

$O(n)$

```
colors = ['green', 'yellow', 'blue', 'pink', 'black', 'white', 'purple'] # n=7

def linear(colors):
    for color in colors:
        print(color)    # O(7)

linear(colors)
```

- `n=4` : 4 operations
- `n=7` : 7 operations
- `n=100` : 100 operations
- ...
- $O(n)$ complexity

$O(n^2)$

```
colors = ['green', 'yellow', 'blue']
```

```
def quadratic(colors):  
    for first in colors:  
        for second in colors:  
            print(first, second)
```

```
quadratic(colors)
```

```
green green  
green yellow  
green blue  
yellow green  
yellow yellow  
yellow blue  
blue green  
blue yellow  
blue blue
```

- `n=3` : (3 x 3) 9 operations
- `n=100` : (100 x 100) 10,000 operations
- quadratic pattern
- $O(n^2)$ complexity

$O(n^3)$

```
colors = ['green', 'yellow', 'blue']

def cubic(colors):
    for color1 in colors:
        for color2 in colors:
            for color3 in colors:
                print(color1, color2, color3)

cubic(colors)
```

- `n=3` : (3 x 3 x 3) **27 operations**
- `n=10` : (10 x 10 x 10) **1,000 operations**
- cubic pattern
- $O(n^3)$ **complexity**

Calculating Big O Notation

```
colors = ['green', 'yellow', 'blue', 'pink', 'black', 'white', 'purple']
other_colors = ['orange', 'brown']

def complex_algorithm(colors):
    color_count = 0

    for color in colors:
        print(color)
        color_count += 1

    for other_color in other_colors:
        print(other_color)
        color_count += 1

    print(color_count)

complex_algorithm(colors)
```

Calculating Big O Notation

```
colors = ['green', 'yellow', 'blue', 'pink', 'black', 'white', 'purple'] # 0(1)
other_colors = ['orange', 'brown'] # 0(1)

def complex_algorithm(colors):
    color_count = 0 # 0(1)

    for color in colors:
        print(color) # 0(n)
        color_count += 1 # 0(n)

    for other_color in other_colors:
        print(other_color) # 0(m)
        color_count += 1 # 0(m)

    print(color_count) # 0(1)

complex_algorithm(colors) # 0(4)
```

Calculating Big O Notation

```
colors = ['green', 'yellow', 'blue', 'pink', 'black', 'white', 'purple'] # 0(1)
other_colors = ['orange', 'brown'] # 0(1)

def complex_algorithm(colors):
    color_count = 0 # 0(1)

    for color in colors:
        print(color) # 0(n)
        color_count += 1 # 0(n)

    for other_color in other_colors:
        print(other_color) # 0(m)
        color_count += 1 # 0(m)

    print(color_count) # 0(1)

complex_algorithm(colors) # 0(4 + 2n)
```

Calculating Big O Notation

```
colors = ['green', 'yellow', 'blue', 'pink', 'black', 'white', 'purple'] # 0(1)
other_colors = ['orange', 'brown'] # 0(1)

def complex_algorithm(colors):
    color_count = 0 # 0(1)

    for color in colors:
        print(color) # 0(n)
        color_count += 1 # 0(n)

    for other_color in other_colors:
        print(other_color) # 0(m)
        color_count += 1 # 0(m)

    print(color_count) # 0(1)

complex_algorithm(colors) # 0(4 + 2n + 2m)
```

Simplifying Big O Notation

1. Remove constants
 - $O(4 + 2n + 2m) \rightarrow O(n + m)$
2. Different variables for different inputs
 - $O(n + m)$
3. Remove smaller terms
 - $O(n + n^2)$

Simplifying Big O Notation

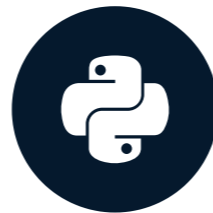
1. Remove constants
 - $O(4 + 2n + 2m) \rightarrow O(n + m)$
2. Different variables for different inputs
 - $O(n + m)$
3. Remove smaller terms
 - $O(n + n^2) \rightarrow O(n^2)$

Let's practice!

DATA STRUCTURES AND ALGORITHMS IN PYTHON

Working with stacks

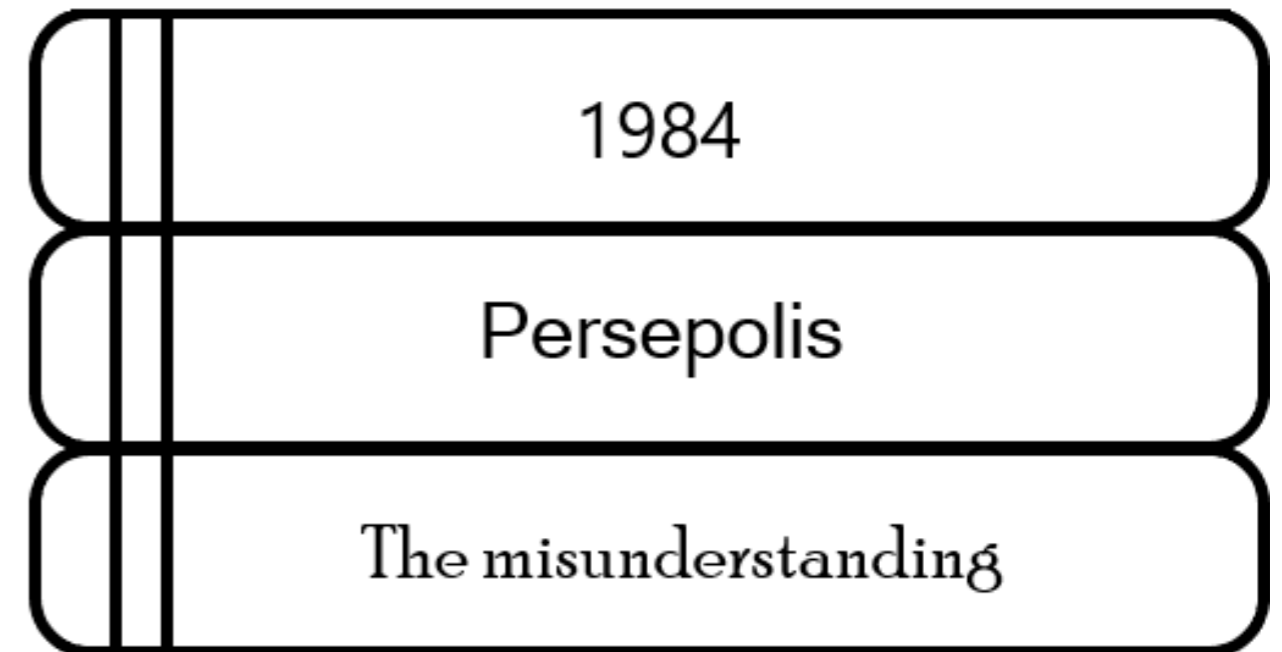
DATA STRUCTURES AND ALGORITHMS IN PYTHON



Miriam Antona
Software Engineer

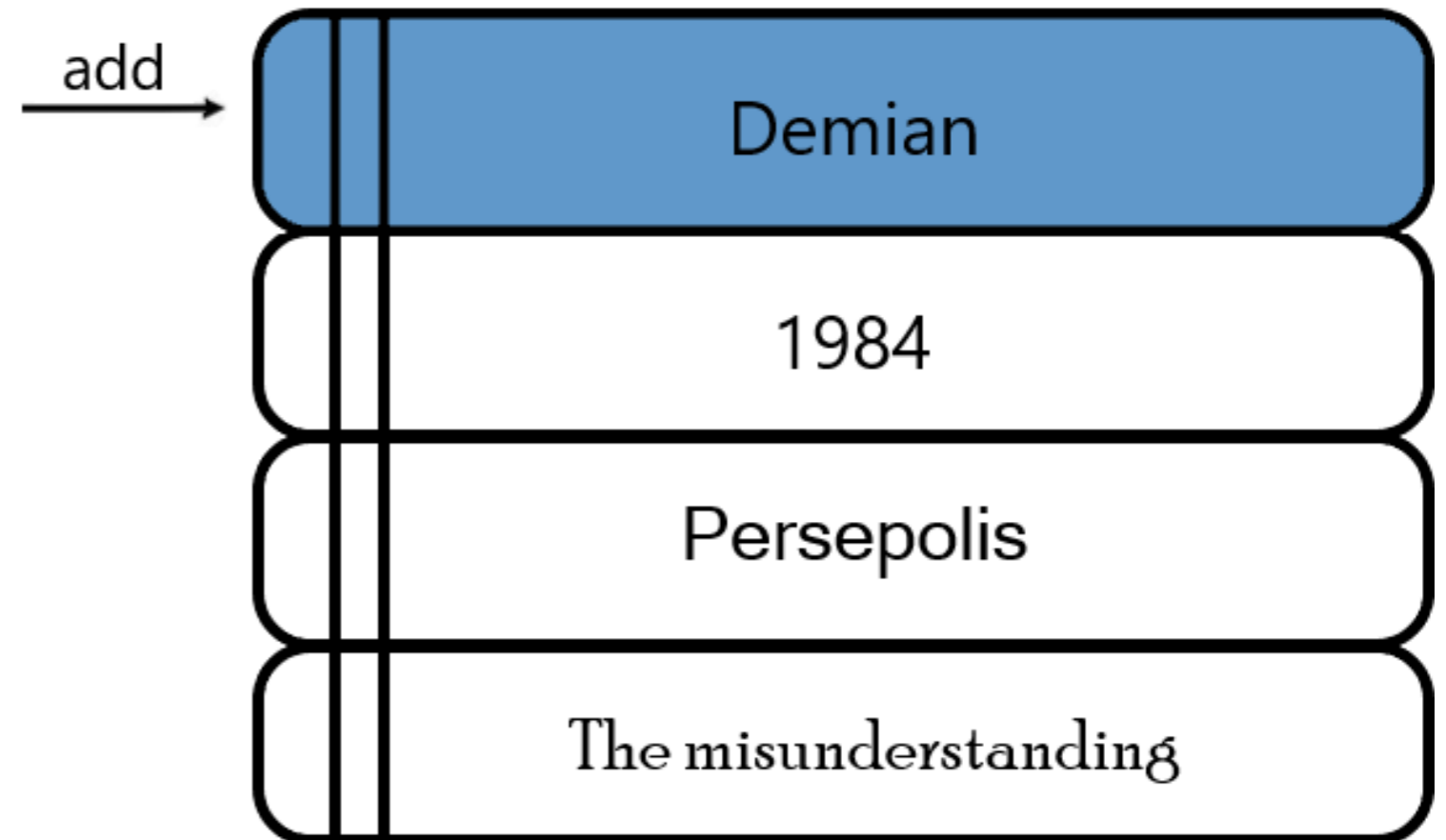
Stacks

- **LIFO:** Last-In First-Out
 - **Last inserted** item will be the **first** item to be removed



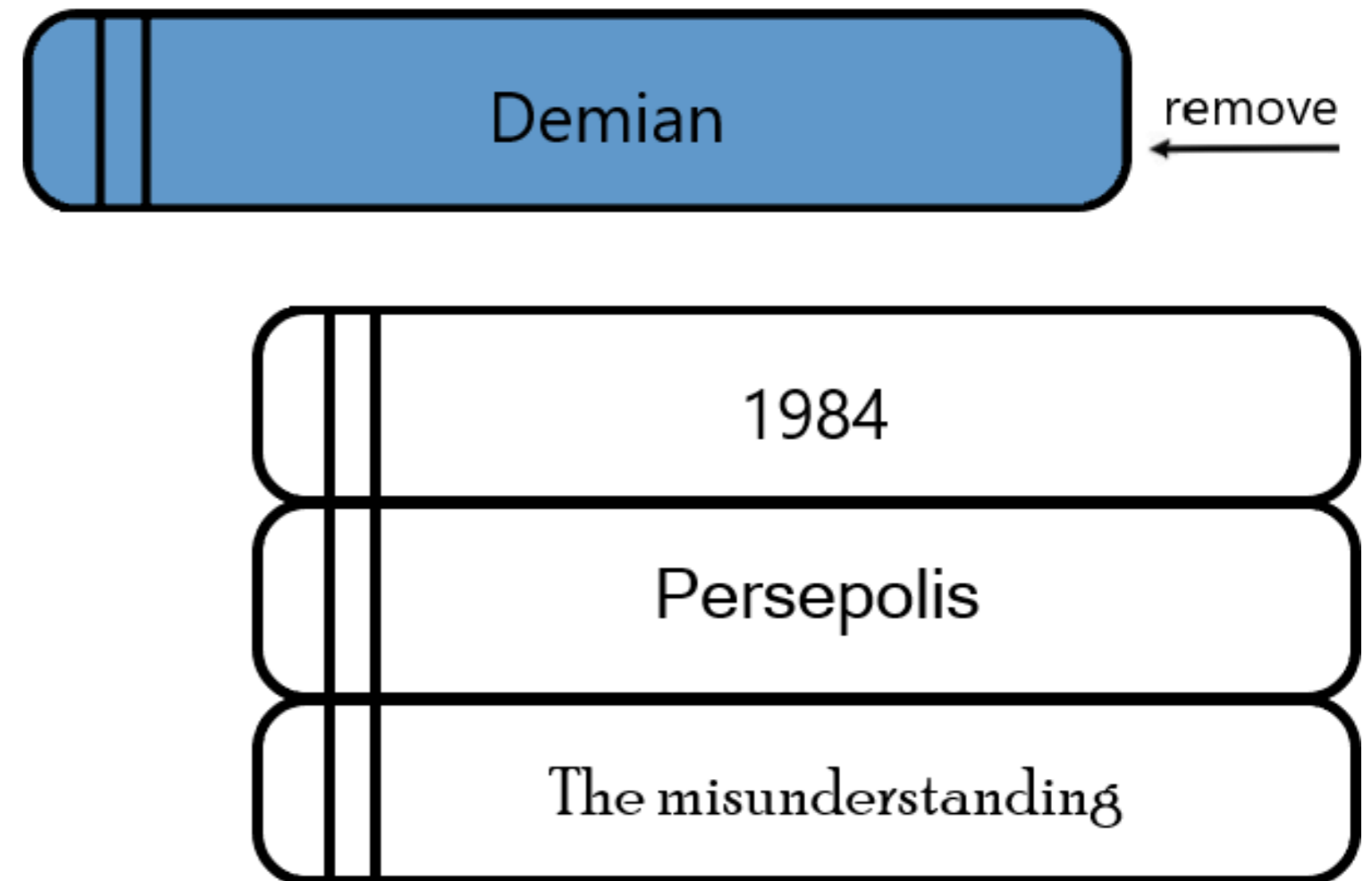
Stacks

- **LIFO:** Last-In First-Out
 - **Last inserted** item will be always the **first** item to be **removed**
- Can only **add** at the **top**
 - *Pushing onto the stack*



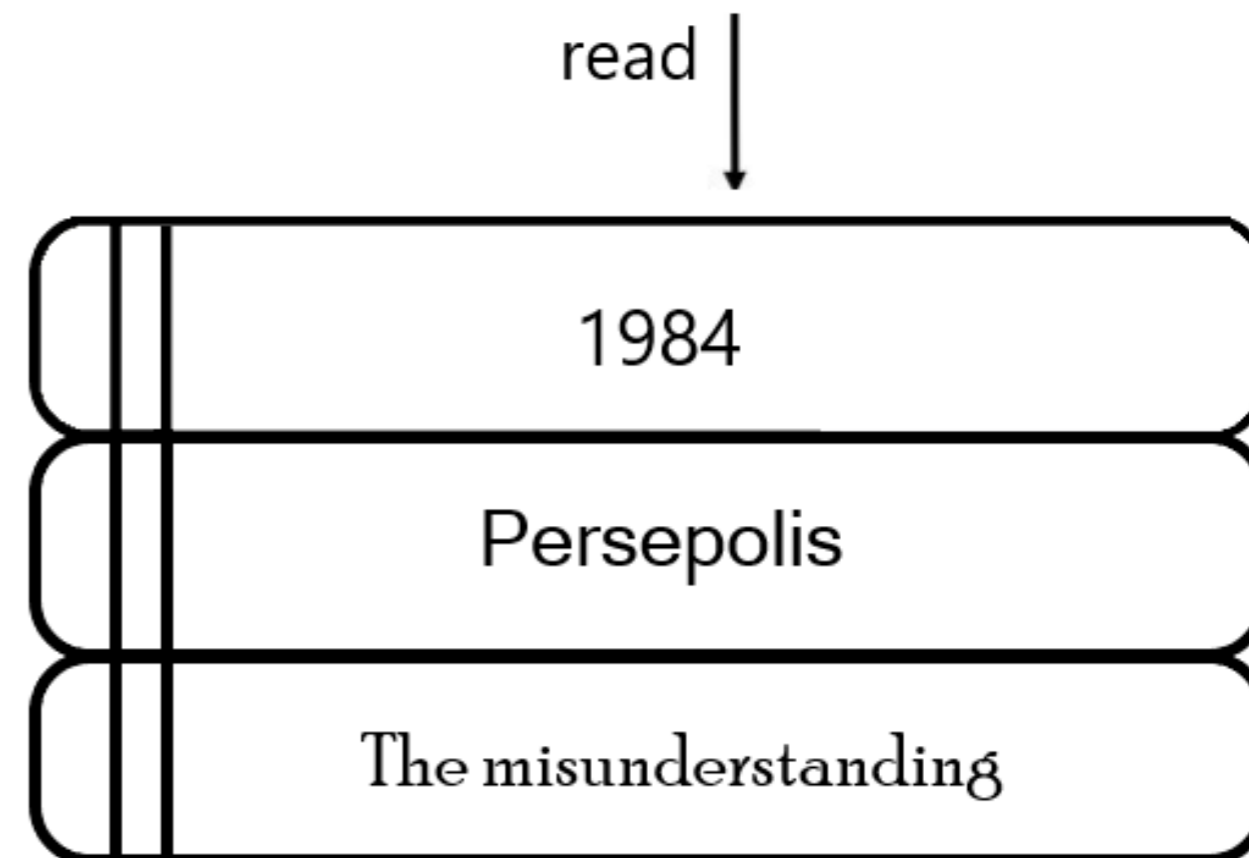
Stacks

- LIFO: Last-In First-Out
 - Last inserted item will be always the first item to be removed
- Can only add at the top
 - *Pushing onto the stack*
- Can only take from the top
 - *Popping from the stack*



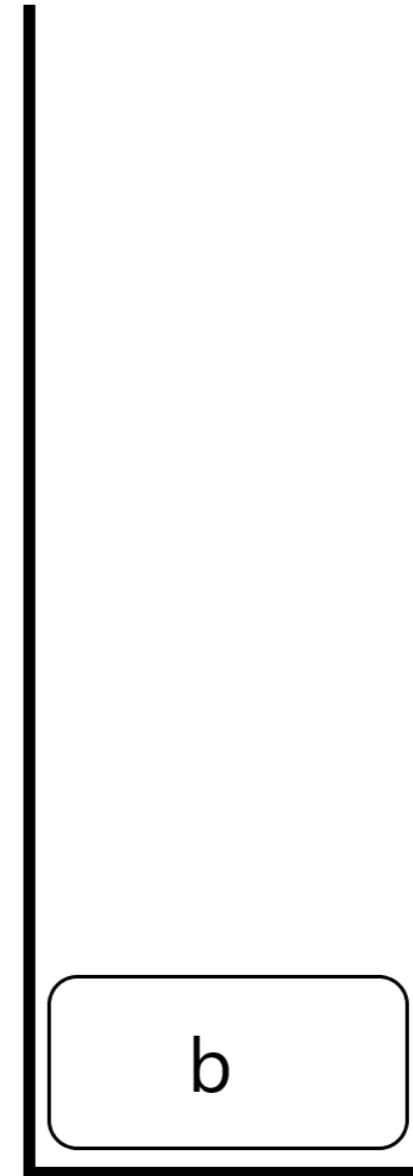
Stacks

- **LIFO:** Last-In First-Out
 - **Last inserted** item will be always the **first** item to be **removed**
- Can only **add** at the **top**
 - *Pushing onto the stack*
- Can only **remove** from the **top**
 - *Popping from the stack*
- Can only **read** the **last element**
 - *Peeking from the stack*



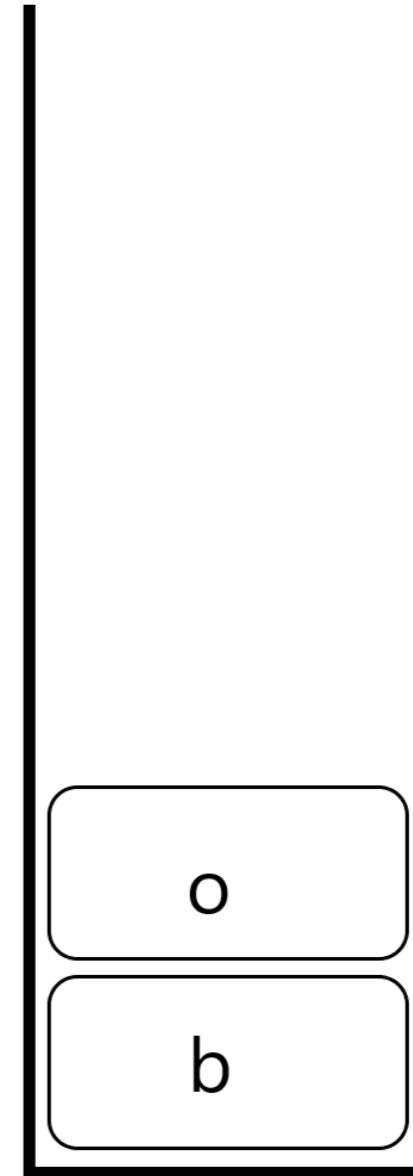
Stacks - real uses

- Undo functionality



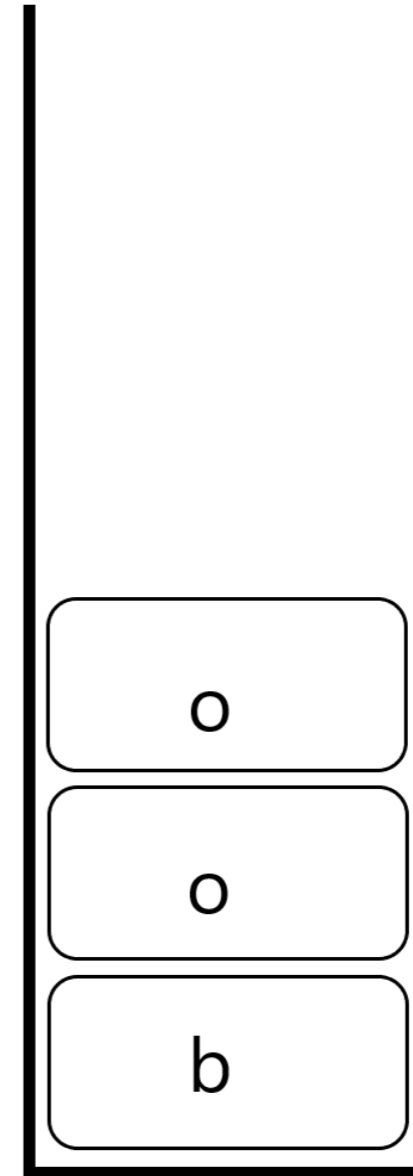
Stacks - real uses

- Undo functionality
 - **push** each keystroke



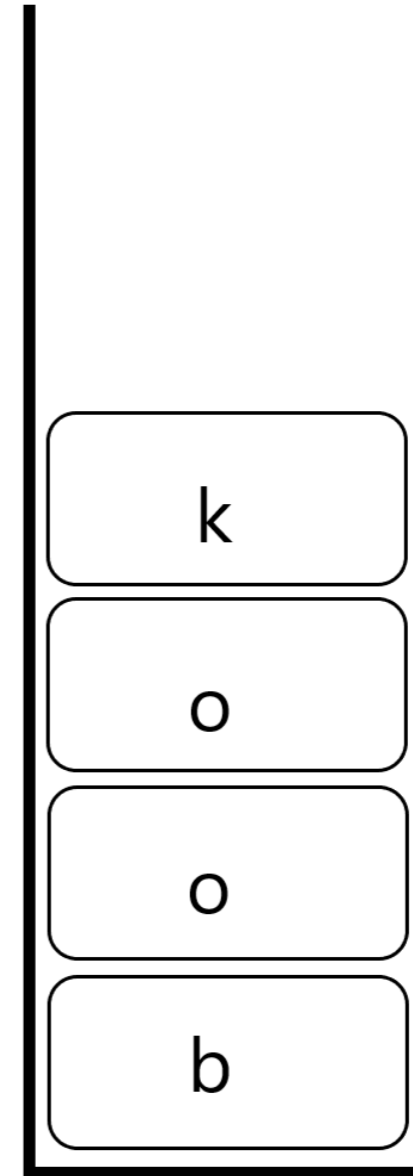
Stacks - real uses

- Undo functionality
 - **push** each keystroke



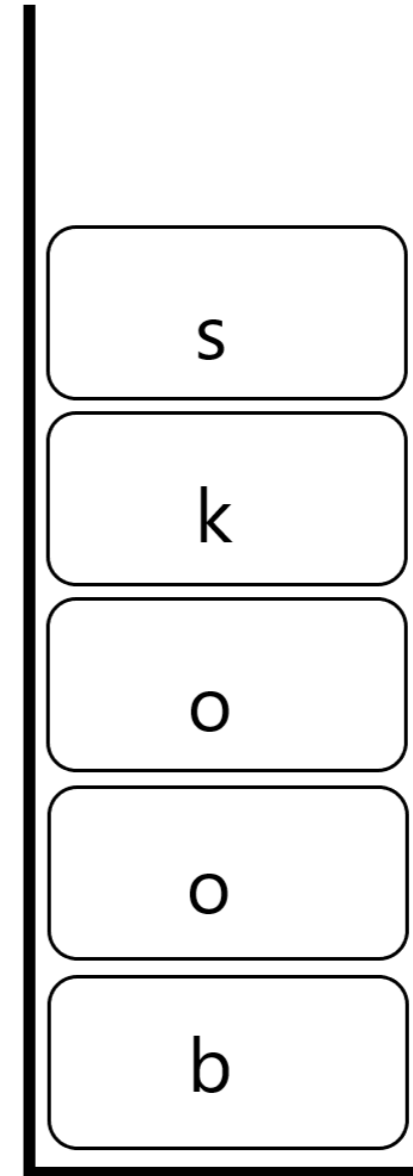
Stacks - real uses

- Undo functionality
 - **push** each keystroke



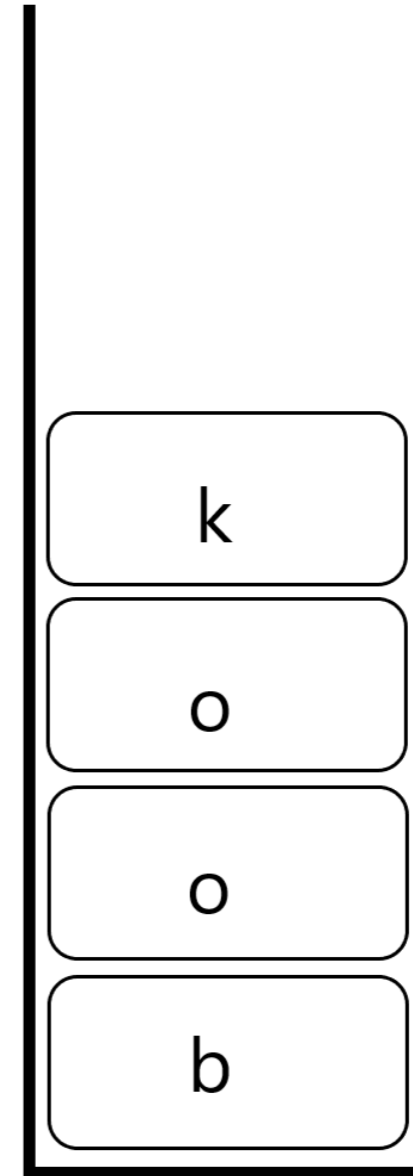
Stacks - real uses

- Undo functionality
 - **push** each keystroke



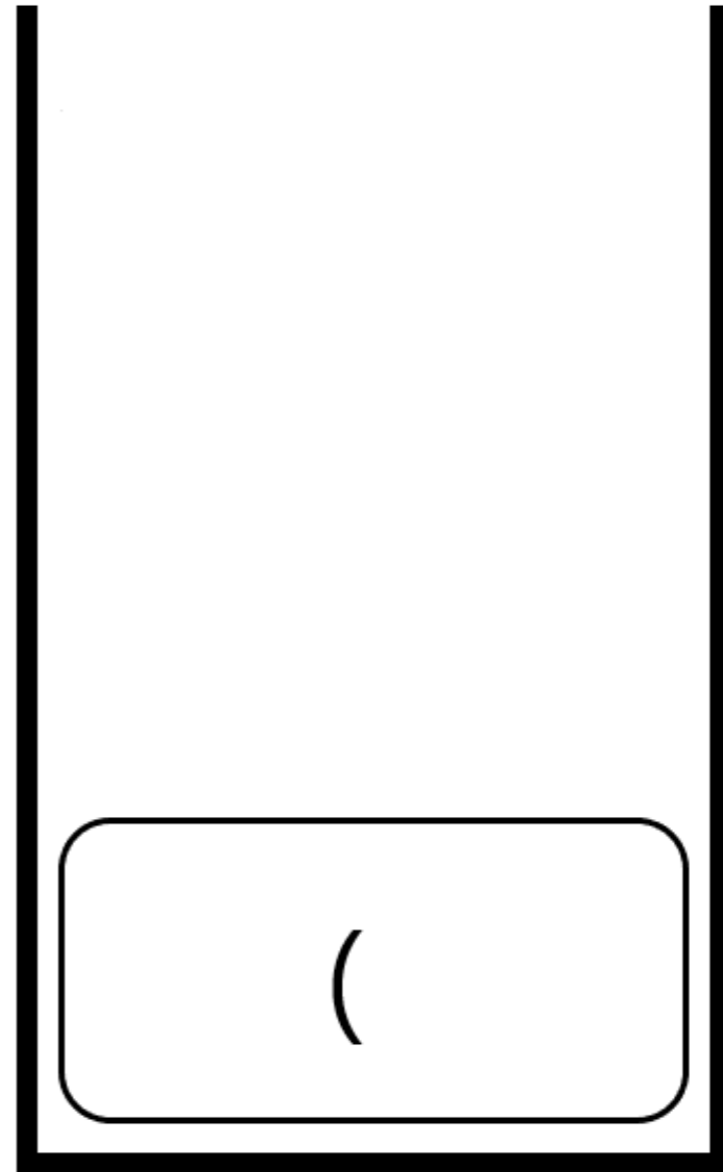
Stacks - real uses

- Undo functionality
 - **push** each keystroke
 - **pop** last inserted keystroke



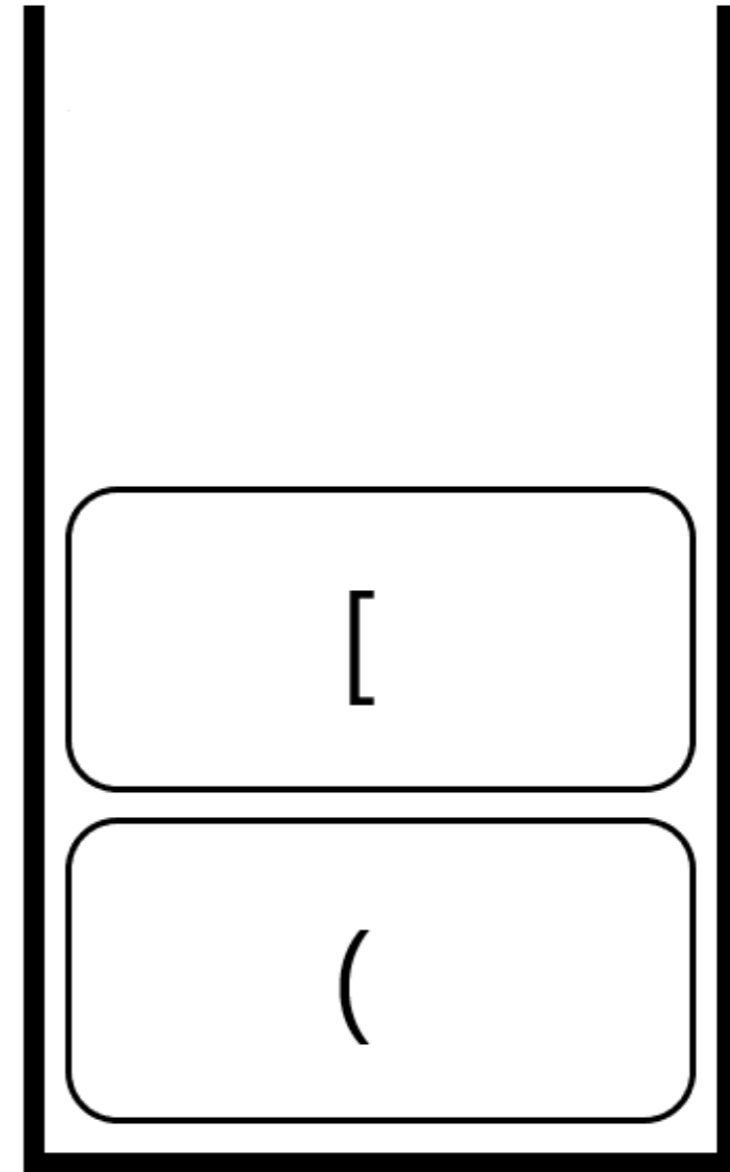
Stacks - real uses

- Symbol checker: ([{}])
 - **push** opening symbols



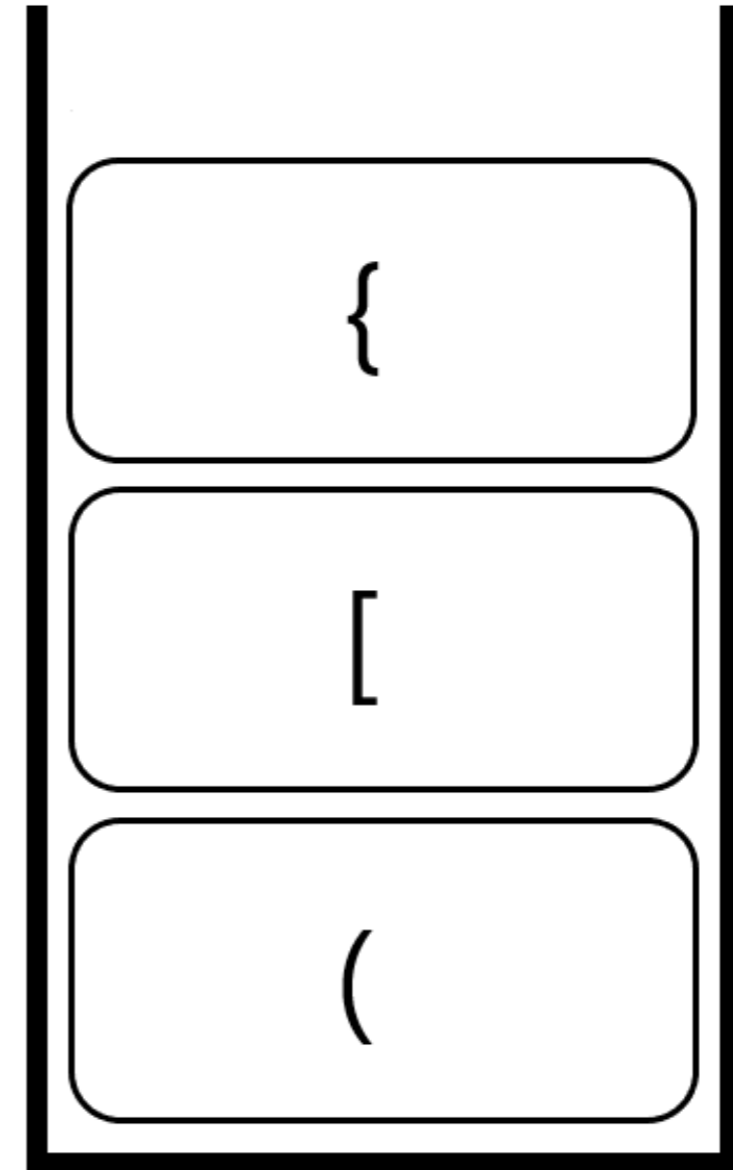
Stacks - real uses

- Symbol checker: ([{}])
 - push opening symbols



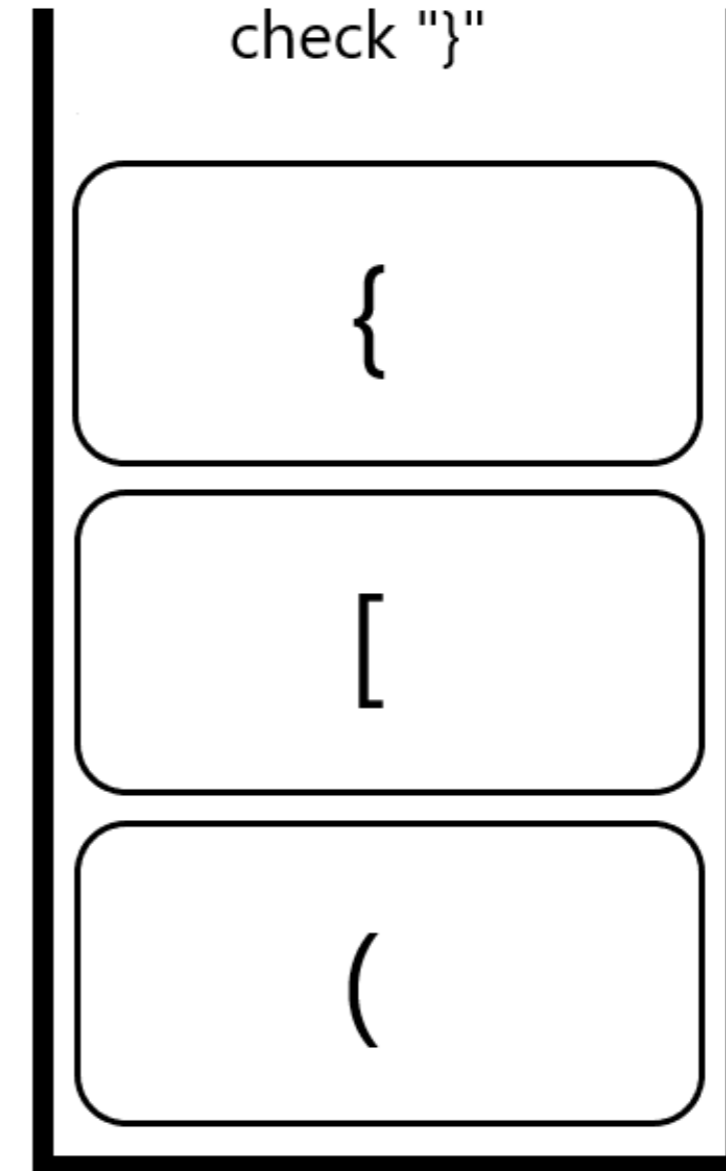
Stacks - real uses

- Symbol checker: ([{}])
 - push opening symbols



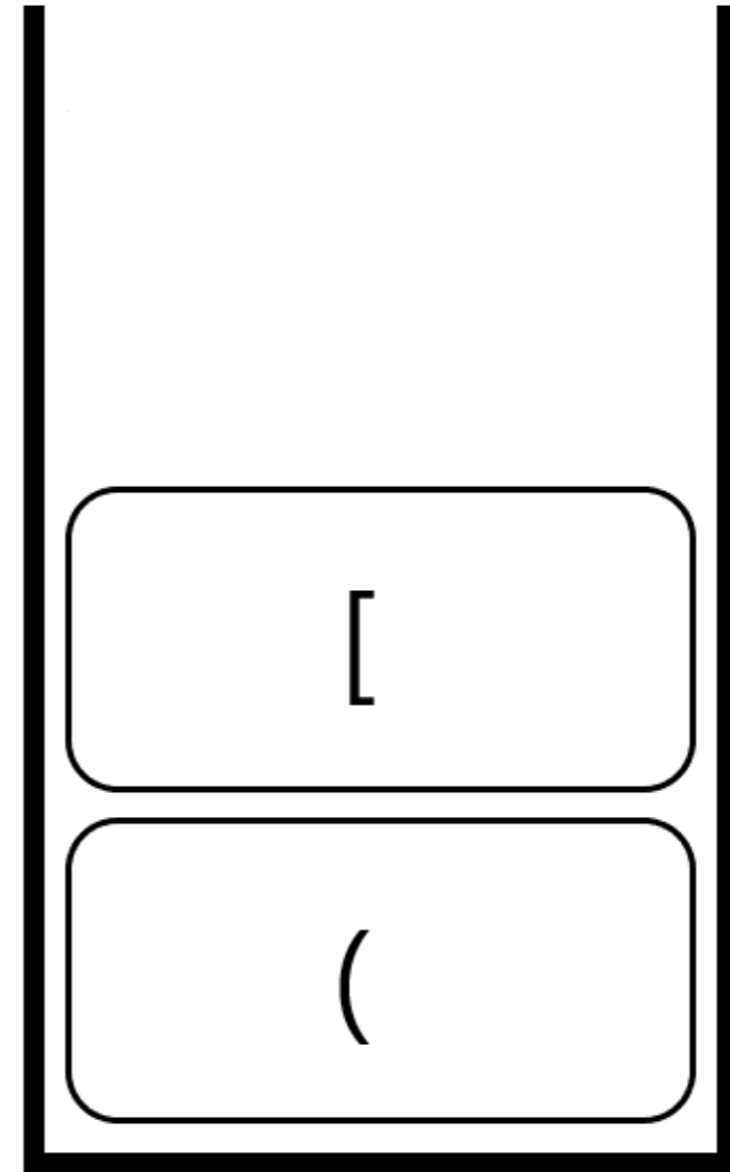
Stacks - real uses

- Symbol checker: ([{}])
 - **push** opening symbols
 - **check** closing symbol



Stacks - real uses

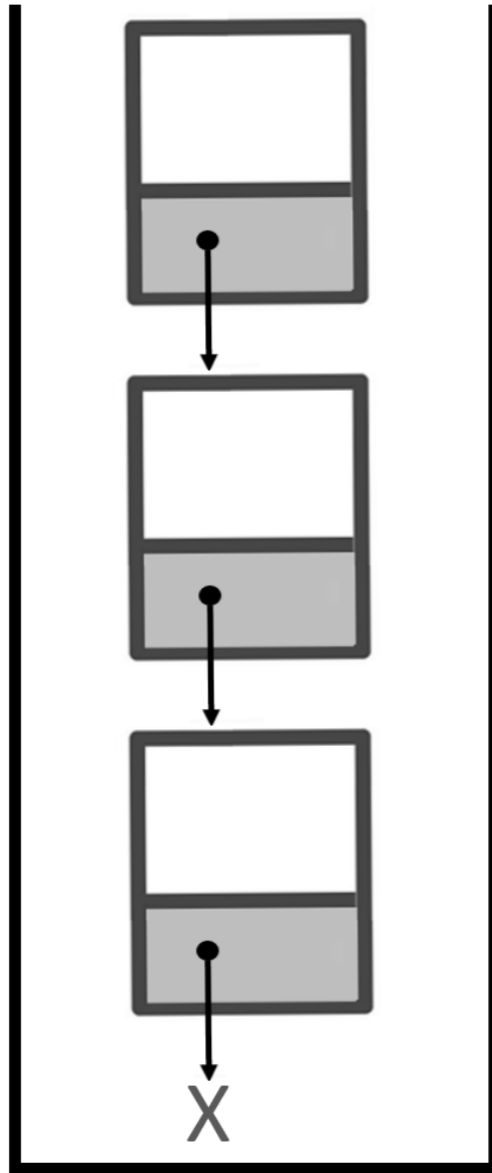
- Symbol checker: ([{ }])
 - **push** opening symbols
 - **check** closing symbol
 - **pop** matching opening symbol



Stacks - real uses

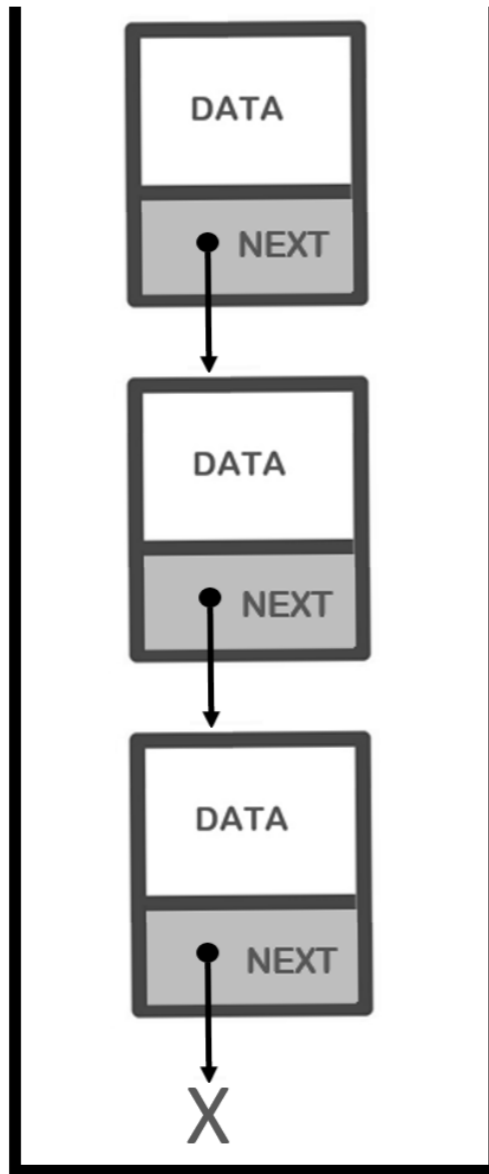
- Function calls
 - **push** block of memory
 - **pop** after the execution ends

Stacks - implementation using singly linked lists



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

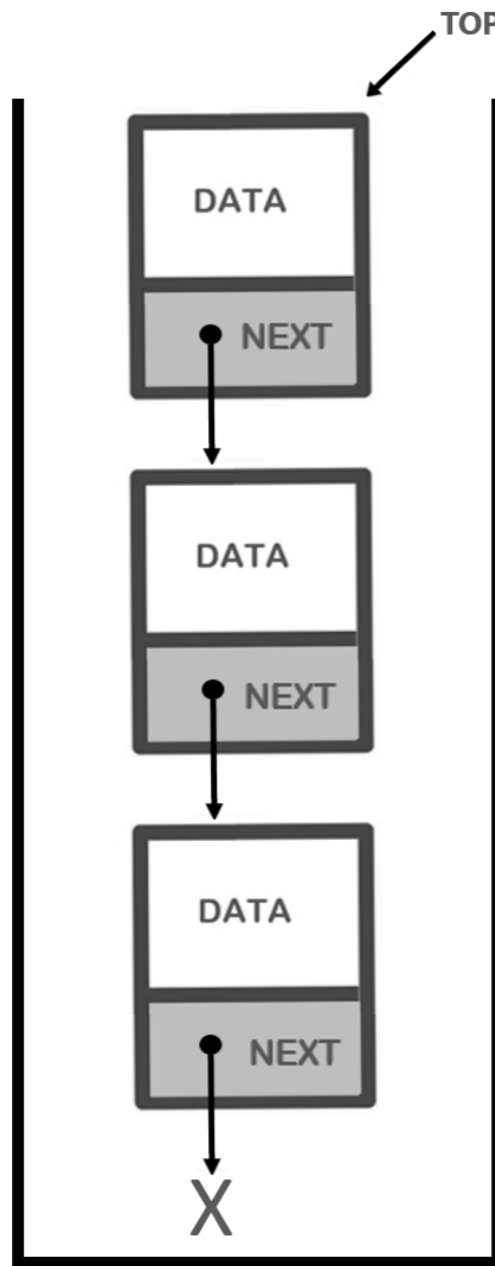
Stacks - implementation using singly linked lists



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

```
class Stack:
    def __init__(self):
        self.top = None
```

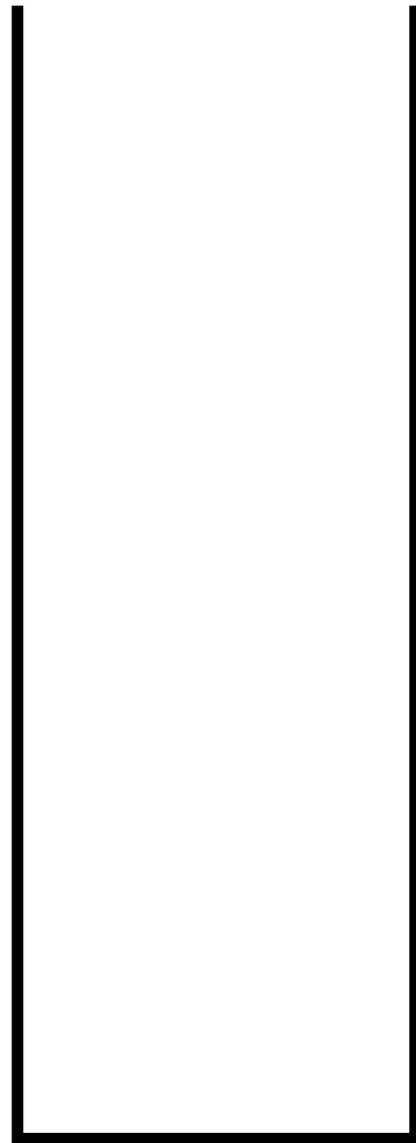
Stacks - implementation using singly linked lists



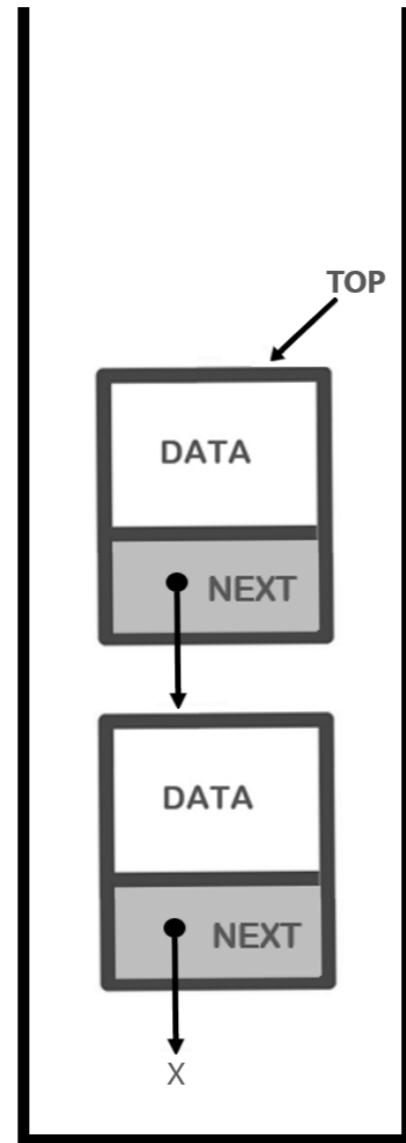
```
class Node:  
    def __init__(self, data):  
        self.data = data  
        self.next = None
```

```
class Stack:  
    def __init__(self):  
        self.top = None
```

Stacks - push



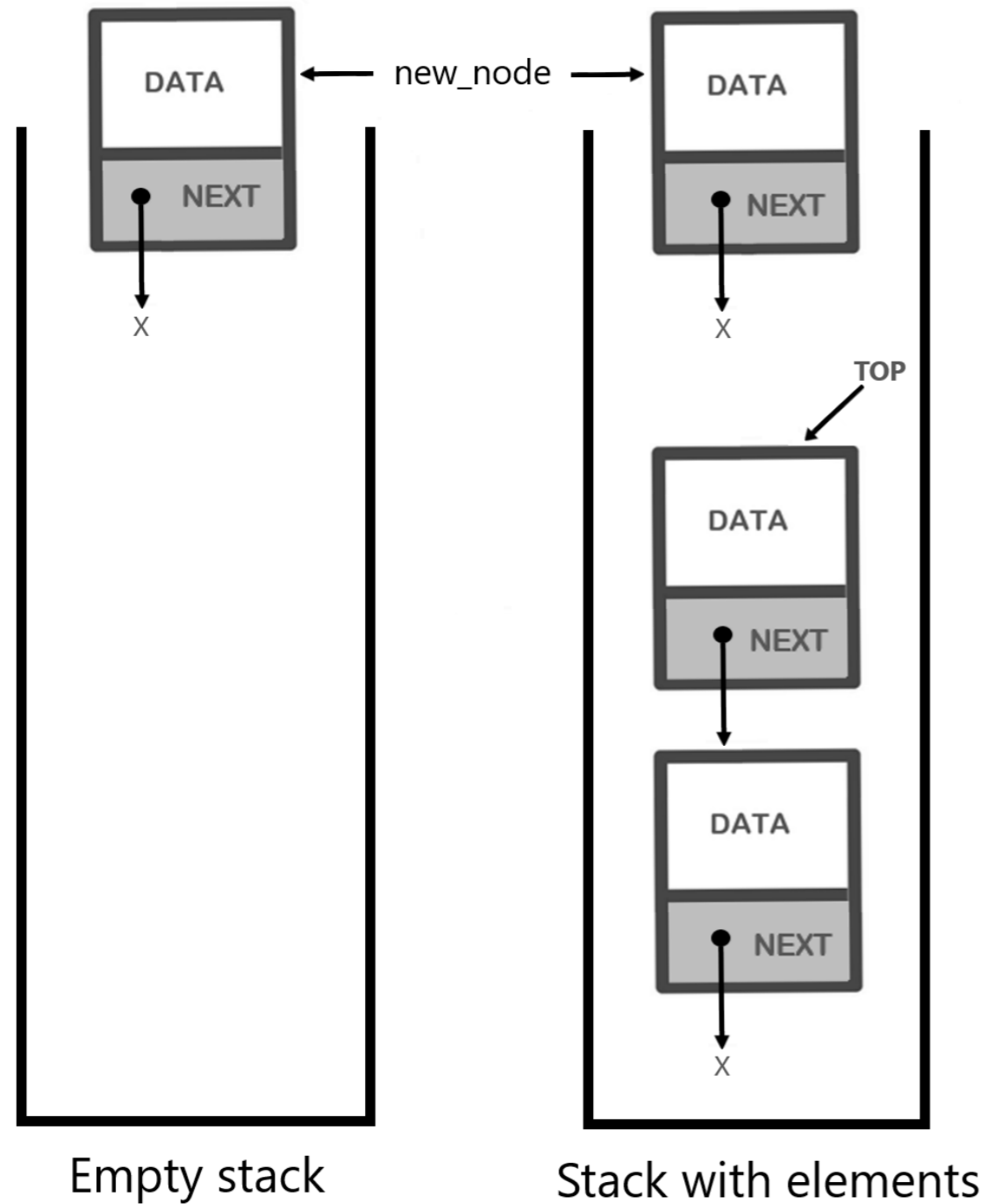
Empty stack



Stack with elements

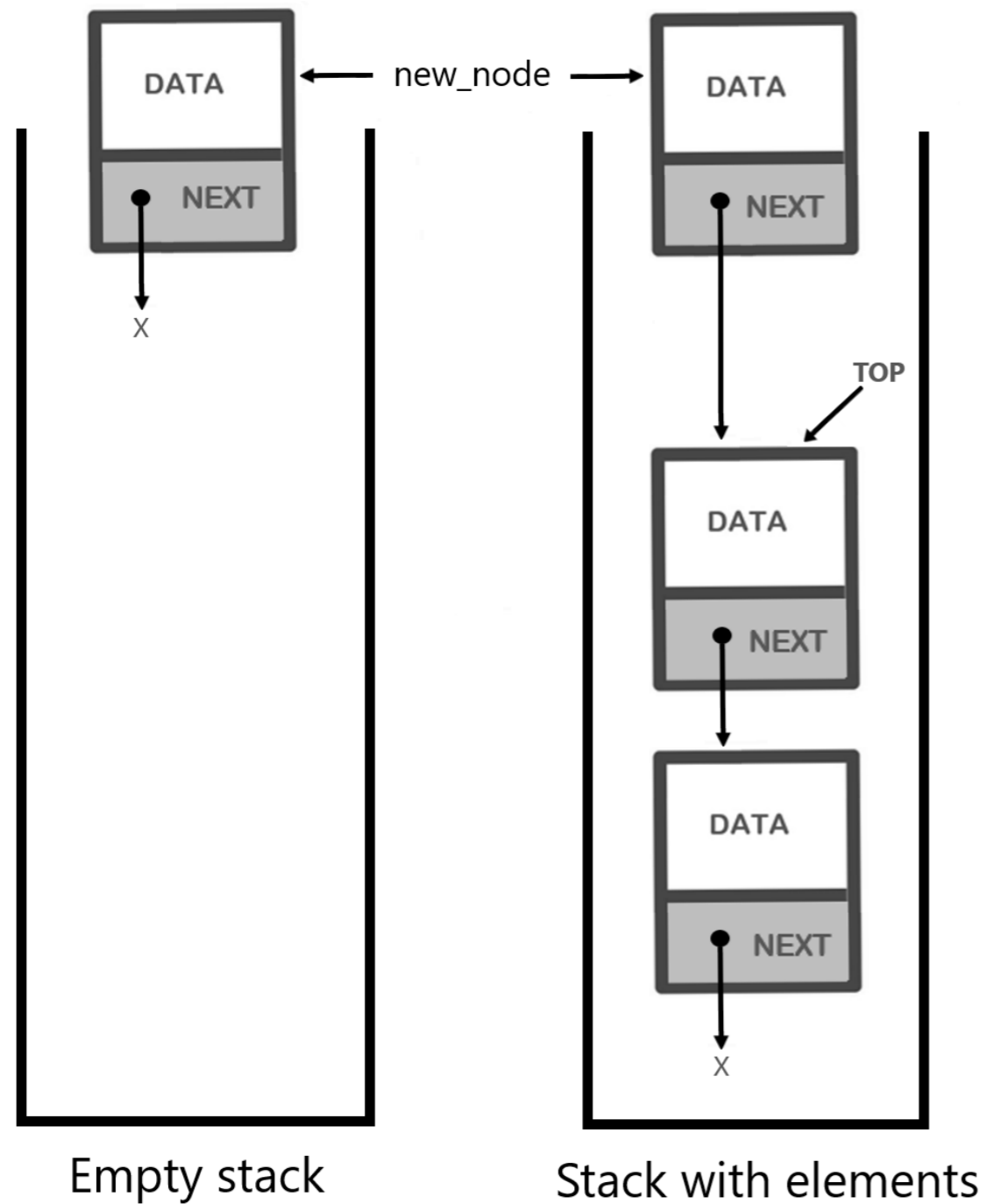
```
def push(self, data):
```

Stacks - push



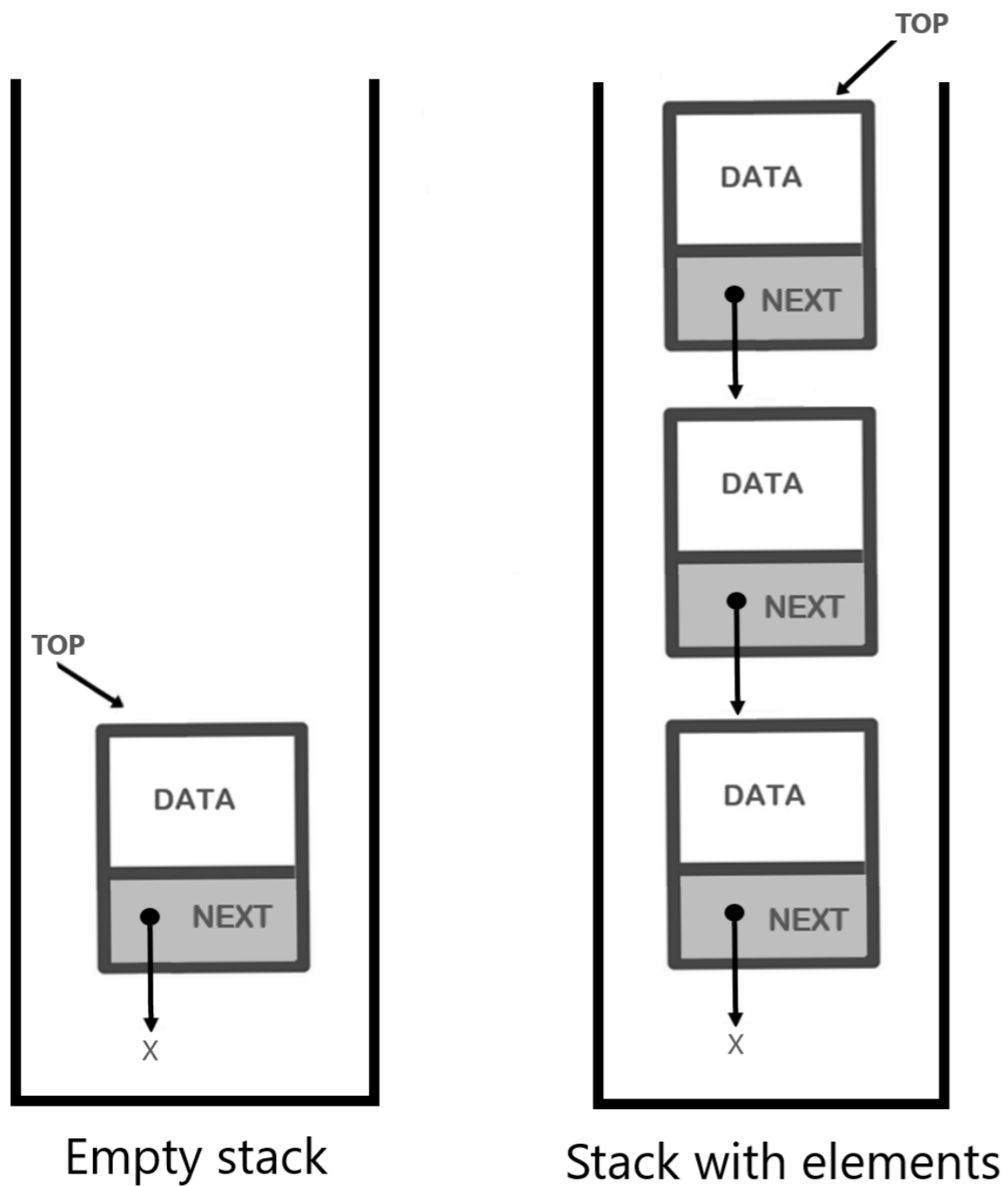
```
def push(self, data):  
    new_node = Node(data)  
    if self.top:
```

Stacks - push



```
def push(self, data):  
    new_node = Node(data)  
    if self.top:  
        new_node.next = self.top
```

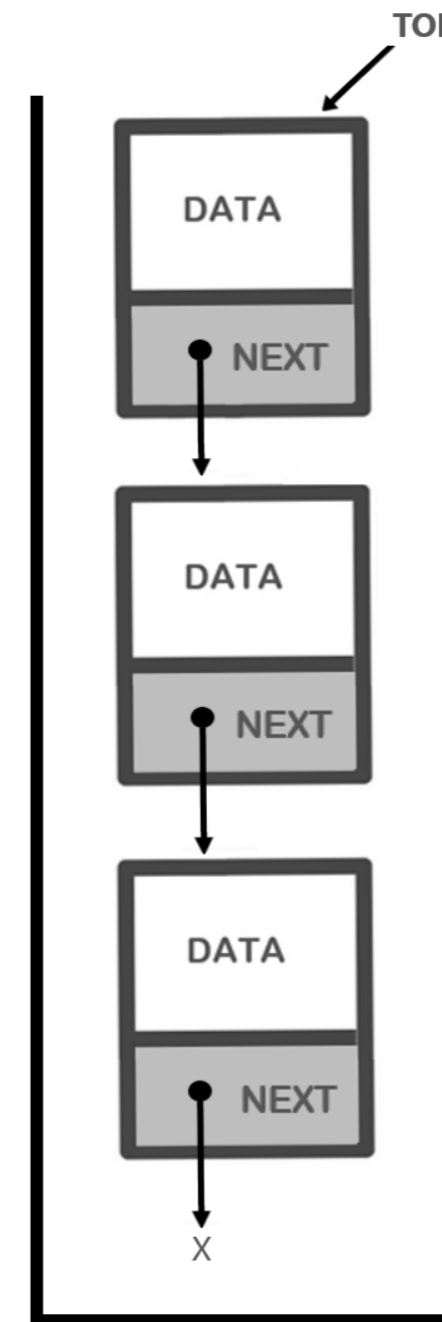
Stacks - push



```
def push(self, data):  
    new_node = Node(data)  
    if self.top:  
        new_node.next = self.top  
    self.top = new_node
```

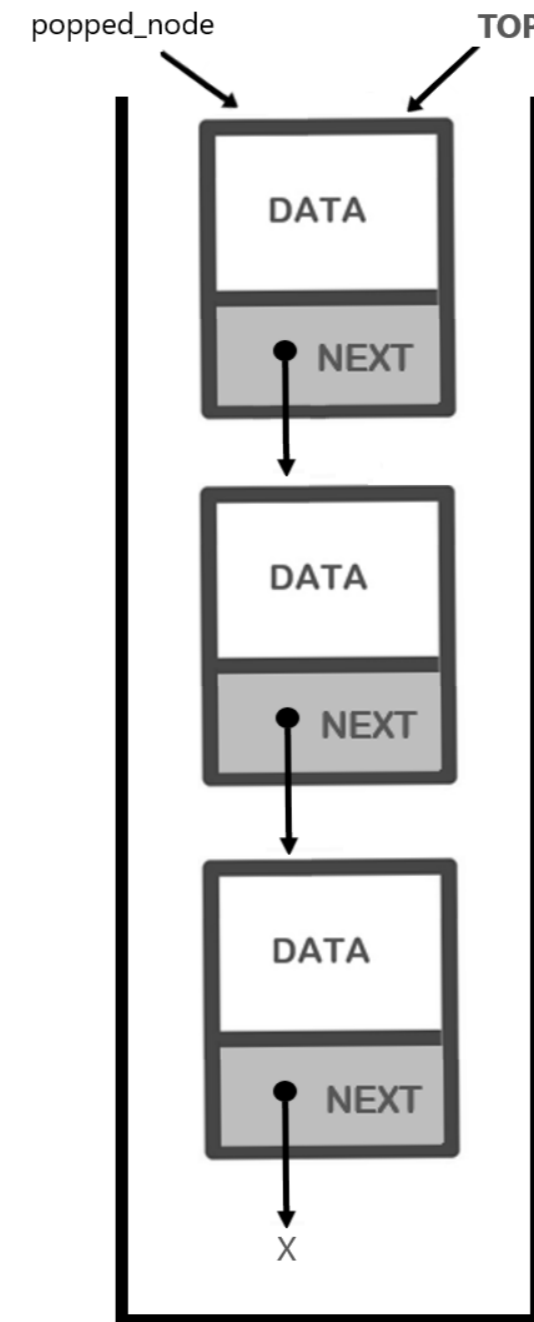
Stacks - pop

```
def pop(self):  
    if self.top is None:  
        return None  
    else:
```



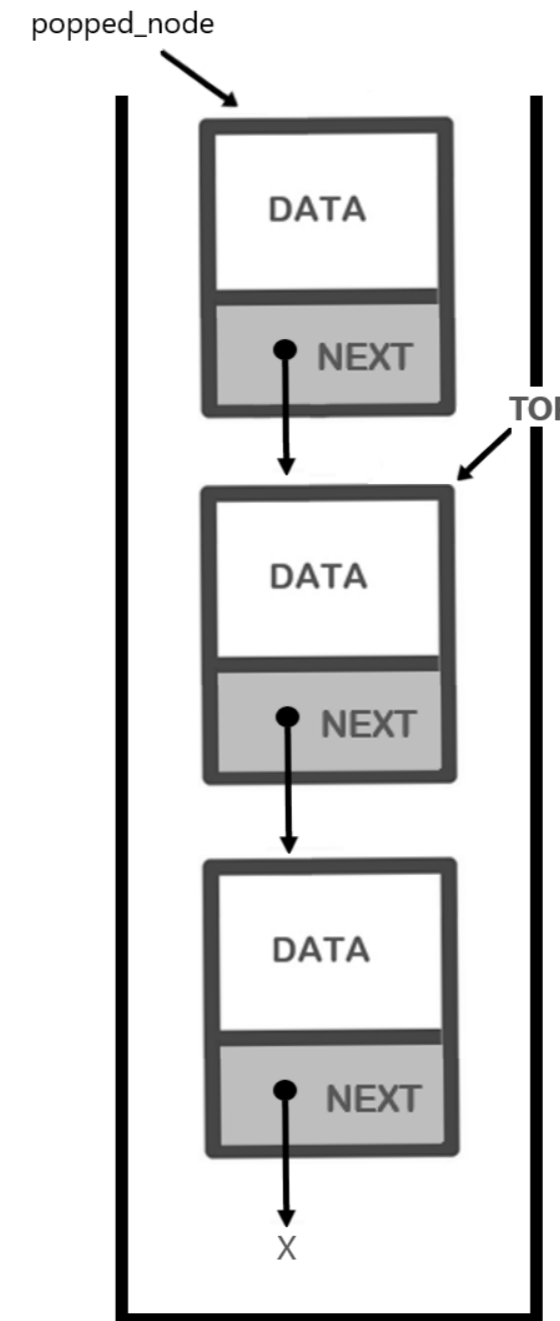
Stacks - pop

```
def pop(self):  
    if self.top is None:  
        return None  
    else:  
        popped_node = self.top
```



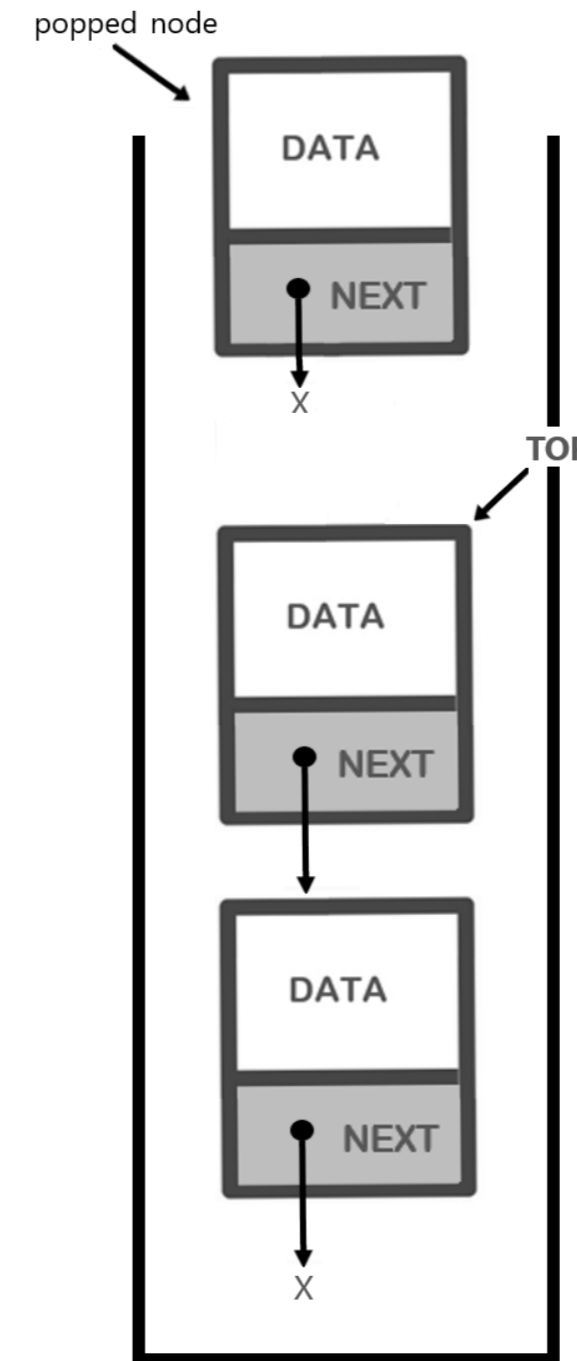
Stacks - pop

```
def pop(self):  
    if self.top is None:  
        return None  
    else:  
        popped_node = self.top  
        self.top = self.top.next
```



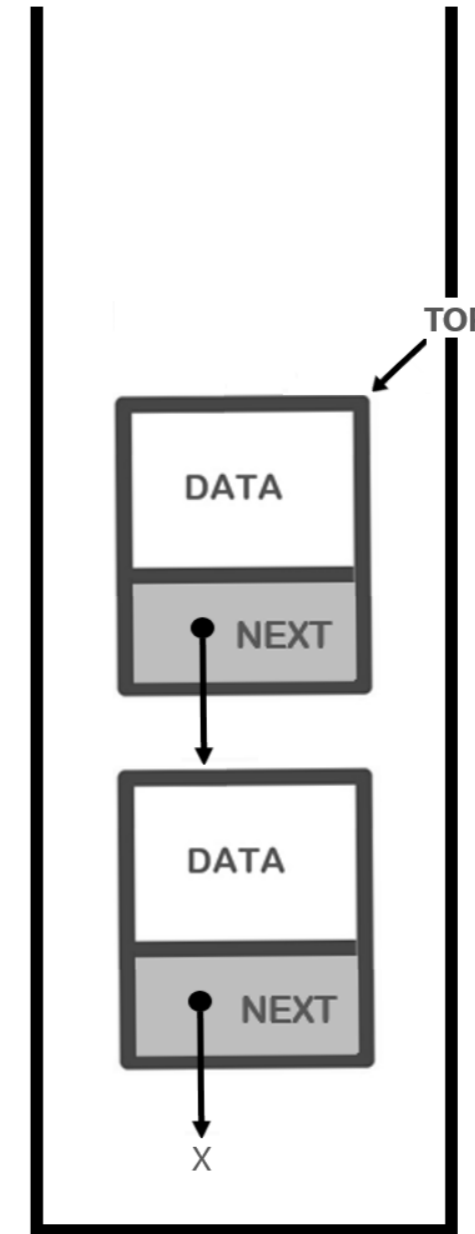
Stacks - pop

```
def pop(self):  
    if self.top is None:  
        return None  
    else:  
        popped_node = self.top  
        self.top = self.top.next  
        popped_node.next = None
```



Stacks - pop

```
def pop(self):  
    if self.top is None:  
        return None  
    else:  
        popped_node = self.top  
        self.top = self.top.next  
        popped_node.next = None  
        return popped_node.data
```



Stacks - peek

```
def peek(self):  
    if self.top:  
        return self.top.data  
    else:  
        return None
```

LifoQueue in Python

- LifoQueue:
 - Python's `queue` module
 - behaves like a stack

```
import queue
```

```
my_book_stack = queue.LifoQueue(maxsize=0)
my_book_stack.put("The misunderstanding")
my_book_stack.put("Persepolis")
my_book_stack.put("1984")

print("The size is: ", my_book_stack.qsize())
```

```
The size is: 3
```

```
print(my_book_stack.get())
print(my_book_stack.get())
print(my_book_stack.get())
```

```
1984
Persepolis
The misunderstanding
```

```
print("Empty stack: ", my_book_stack.empty())
```

```
Empty stack: True
```

Let's practice!

DATA STRUCTURES AND ALGORITHMS IN PYTHON