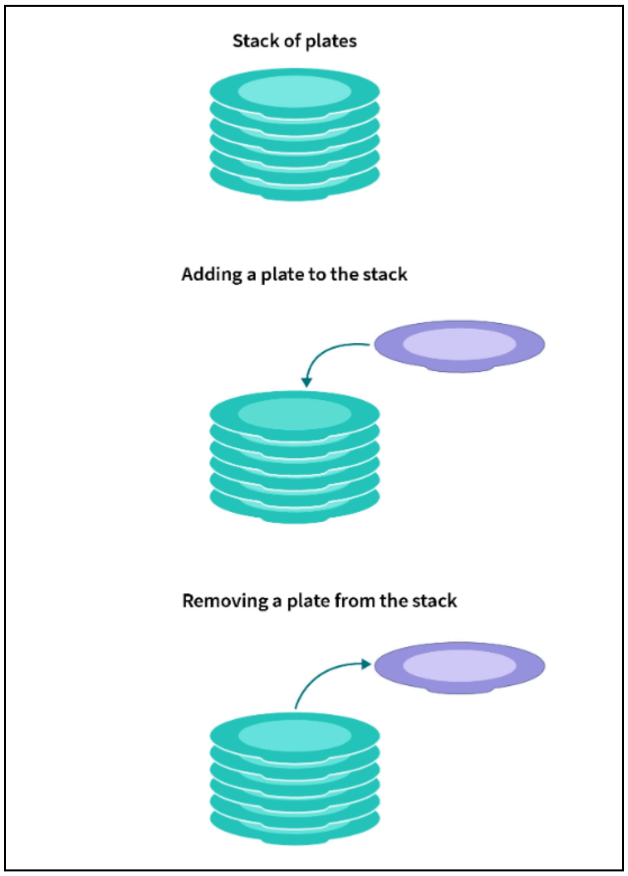
Stack

Like Array and Linked List, a stack is a simple data structure for storing data. In a stack, the order in which the data arrives is important. A pile of plates in a cafeteria is a good example of a stack. The plates are added to the stack as they are cleaned and placed on top. When a plate, is required it is taken from the top of the stack. The first plate placed on the stack is the last one to be used.

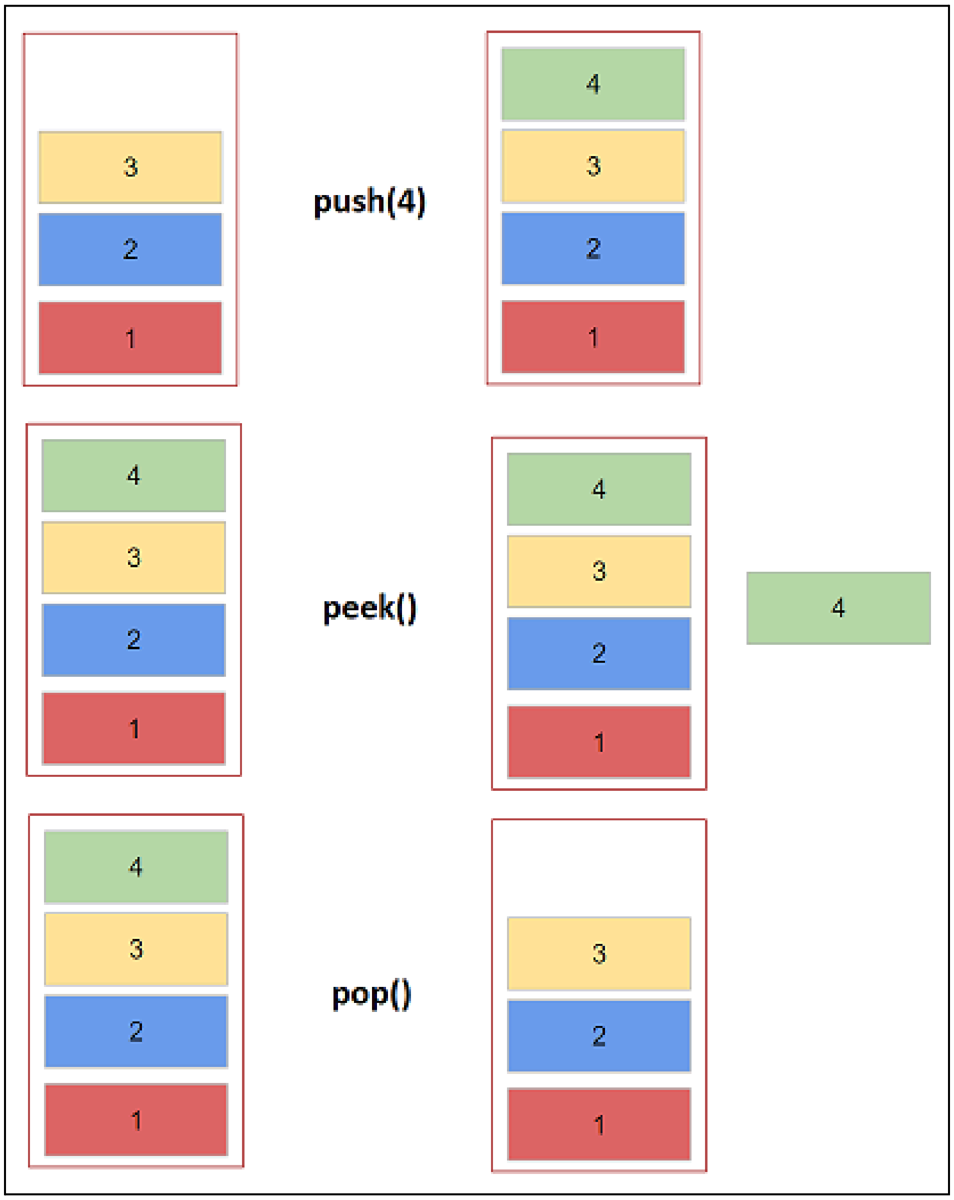


Definition: A stack is an ordered list in which insertion and deletion are done at one end, called top. The last element inserted is the first one to be deleted. Hence, it is called the Last in First out (LIFO)

Hence, it is impossible to add or remove elements from the middle of the stack. The only element that can be taken out is the most recently added one.

Stack Operations:

- push(obj): adds obj to the top of the stack ("overflow" error if the stack has fixed capacity, and is full)
- pop: removes and returns the item from the top of the stack ("underflow" error if the stack is empty)
- peek: returns the item that is on the top of the stack, but does not remove it



Stack Implementation:

- Array-Based Stack:

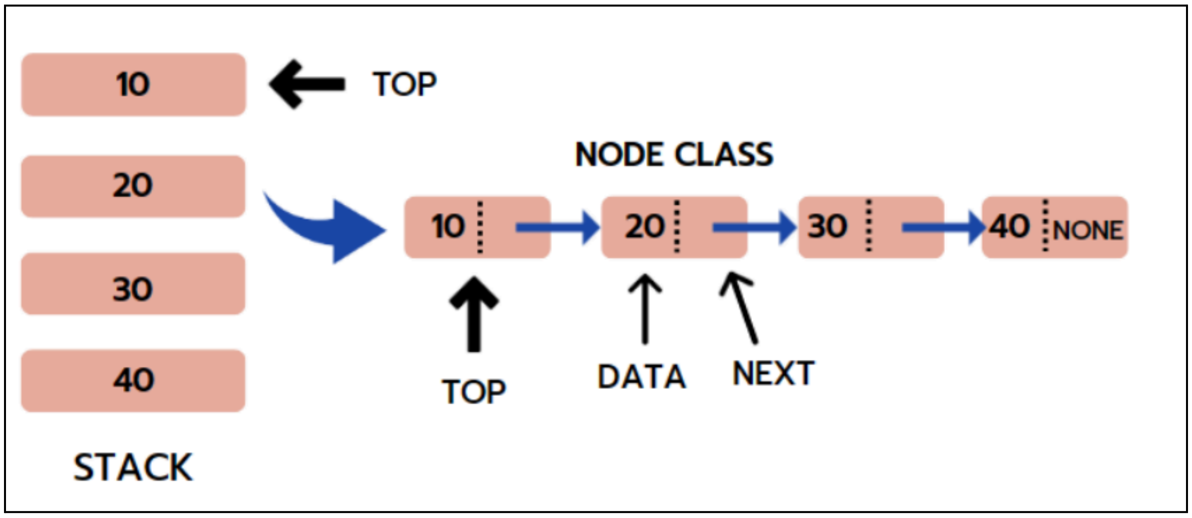
```
class Stack:
    def __init__(self, size):
        self.stack = np.zeros(size, dtype=int)
        self.top = -1
```

```
def push(self, item):
    if self.top == len(self.stack) - 1:
        print("Stack Overflow")
    else:
        self.top += 1
        self.stack[self.top] = item
```

```
def pop(self):
    if self.top == -1:
        print("Stack Underflow")
    else:
        item = self.stack[self.top]
        self.stack[self.top] = 0
        self.top -= 1
        return item
```

```
def peek(self):
    if self.top == -1:
        print("Stack is empty")
    else:
        return self.stack[self.top]
```

- Linked List Based Stack:



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

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

```
def push(self, elem):
    if self.top == None:
        self.top = Node(elem, None)
    else:
        n = Node(elem, None)
        n.next = self.top
        self.top = n
```

```
def pop(self):
    if self.top == None:
        print ("Stack Underflow")
    else:
        popped = self.top
        self.top = self.top.next
        popped.next = None
        return popped.elem
```

```
def peek(self):
    if self.top == None:
        print ("Stack is empty")
    else:
        return self.top.elem
```

Stack Applications:

- Reverse:

```
1. begin with an empty stack and an input stream.
2. while there are more characters to read:
3.     read the next input character;
4.     push it onto the stack;
5. while the stack is not empty:
7.     c = pop the stack;
8.     print c;
```

```
def reverse(string):
        n = len(string)
        stack = Stack()
        for i in range(0, n):
                stack.push(string[i])
        reversed_string = ""
        for i in range(0, n):
                reversed_string += stack.pop()
```

- Expression Evaluation:

Let's first see what is infix, prefix, and postfix

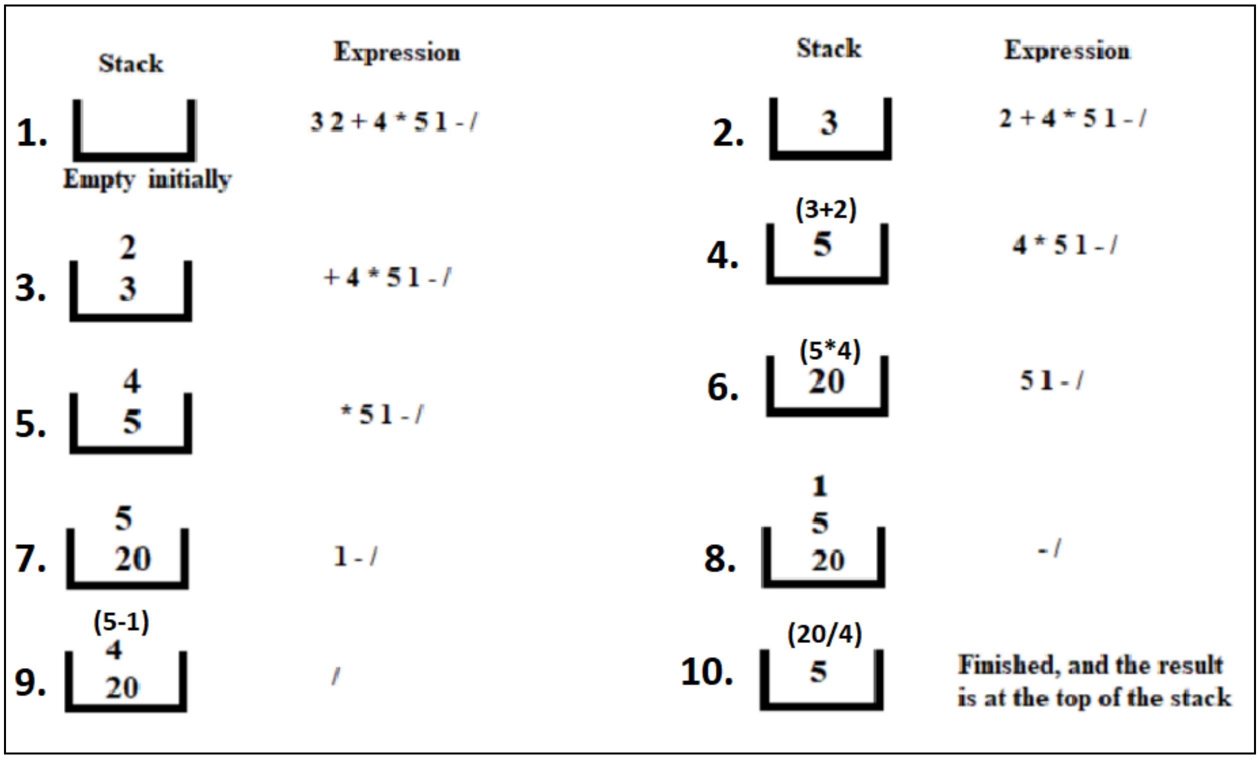
Infix: The operator is placed between the operands. This is the most common and familiar form to humans. E.g. A + B, (A + B) * C

Postfix: The operator is placed after the operands. E.g. A+B -> A B +, (A + B) * C ⇒ A B + C *

Prefix: The operator is placed before the operands. E.g. A+B -> + A B, (A + B) * C ⇒ * + A B C

In Expression Evaluation, when an arithmetic expression is presented in the postfix form, you can use a stack to evaluate it to get the final value. E.g.
((3 + 2) * 4) / (5 - 1) ⇒ 3 2 + 4 * 5 1 - / ⇒ 5

```
1. begin with an empty stack and an input stream (for the expression).
2. while there is more input to read:
3.      read the next input symbol;
4.      if it's an operand,
5.              then push it onto the stack;
6.      if it's an operator
7.              then pop two operands from the stack;
8.              operate on the operands;
9.              push the result;
10. pop the answer;
```



- Parentheses Matching

In many editors, it is common to want to know which left parenthesis (or brace or bracket, etc) will be matched when we type a right parenthesis. We want the most recent left parenthesis. Therefore, the editor can keep track of parentheses by pushing them onto a stack, and then popping them off as each right parenthesis is encountered.

We often have expressions involving "()[]{}" that require that the different types of parentheses are balanced. For example, the following are properly balanced:

(a (b + c) + d)
[(a b) (c d)]
([a {x y} b])

But the following are not:

(a (b + c) + d
[(a b] (c d))
([a {x y) b])

```
1.  begin with an empty stack and an input stream (for the expression).
2.  while there is more input to read:
3.      read the next input character;
4.      if it's an opening parenthesis/brace/bracket ("(" or "{" or "[")
5.              then push it onto the stack;
6.      if it's a closing parenthesis/brace/bracket (")" or "}" or "]")
7.              then pop the opening symbol from the stack;
8.              compare the closing with the opening symbol;
9.              if it matches
10.                     then continue with the next input character;
11.             if it does not match
12.                     then return false;
13. return true;
```