Stack Cheat Sheet: Concepts, Operations, Patterns

1. What is a Stack?

- **LIFO** (Last-In-First-Out) data structure.
- Add/remove from the same end (the top).
- Typical uses: parsing, backtracking, function call stacks, undo/redo, and expression evaluation.

2. Python Stack Implementations

a. Using List (Built-in)

```
stack = []
stack.append(1)     # Push
stack.append(2)
top = stack.pop()   # Pop, returns 2. Raises IndexError if empty.
peek = stack[-1]    # Top of stack (but not removed)
empty = len(stack) == 0
```

b. Using collections.deque (safer/faster for large stacks)

```
from collections import deque
stack = deque()
stack.append('a')    # Push
stack.pop()          # Pop
```

3. Custom Stack Class

```
class Stack:
    def __init__(self):
        self.stack = []
    def push(self, val):
        self.stack.append(val)
    def pop(self):
        return self.stack.pop() if not self.is_empty() else None
    def is_empty(self):
        return len(self.stack) == 0
```

```
    def peek(self):
        return self.stack[-1] if not self.is_empty() else None
```

4. Classic Stack Coding Patterns

A. Balanced Parentheses

```
def is_balanced(expr):
    stack = []
    pairs = {')':'(', ']':'[', '}':'{'}
    for ch in expr:
        if ch in '([{':
            stack.append(ch)
        elif ch in ')]}':
            if not stack or stack.pop() != pairs[ch]:
                return False
    return not stack
```

B. Min Stack (support O(1) min)

```
class MinStack:
    def __init__(self):
        self.stack = []
        self.min_stack = []
    def push(self, x):
        self.stack.append(x)
        if not self.min_stack or x <= self.min_stack[-1]:
            self.min_stack.append(x)
    def pop(self):
        if self.stack.pop() == self.min_stack[-1]:
            self.min_stack.pop()
    def get_min(self):
        return self.min_stack[-1] if self.min_stack else None
```

C. Next Greater Element

```
def next_greater(arr):
    stack, res = [], [-1]*len(arr)
```

```
    for i, v in enumerate(arr):
        while stack and v > arr[stack[-1]]:
            res[stack.pop()] = v
        stack.append(i)
    return res
```

D. Evaluate Reverse Polish Notation (RPN)

```
def eval_rpn(tokens):
    stack = []
    for t in tokens:
        if t in "+-*/":
            b, a = stack.pop(), stack.pop()
            stack.append(eval(f"{a}{t}{b}"))
        else:
            stack.append(int(t))
    return stack[0]
```

Queue Cheat Sheet: Concepts, Operations, Patterns

1. What is a Queue?

- **FIFO** (First-In-First-Out) data structure.
- Add at the rear (enqueue), remove from the front (dequeue).
- Typical uses: task scheduling, BFS (Breadth-First Search), producer/consumer, print queues.

2. Python Queue Implementations

a. Using collections.deque (Recommended)

```
from collections import deque
queue = deque()
queue.append(1)      # Enqueue
queue.append(2)
front = queue.popleft()  # Dequeue
empty = len(queue) == 0
```

b. Queue with Built-in List (not recommended, slow dequeue)

```
queue = []
queue.append(1)
queue.append(2)
first = queue.pop(0)  # O(n) time
```

c. queue.Queue (multi-thread safe, for advanced cases only)

```
from queue import Queue
q = Queue()
q.put(1)
q.get()
```

3. Custom Queue Class

```
class Queue:
    def __init__(self):
        self.q = deque()
    def enqueue(self, val):
        self.q.append(val)
    def dequeue(self):
        return self.q.popleft() if not self.is_empty() else None
    def is_empty(self):
        return len(self.q) == 0
    def peek(self):
        return self.q[0] if not self.is_empty() else None
```

4. Classic Queue Coding Patterns

A. Breadth-First Search (BFS) in Tree/Graph

```
def bfs(root):
    result = []
    queue = deque([root])
    while queue:
        node = queue.popleft()
        result.append(node.val)
```

```
        queue.extend(node.children)   # or (for trees): if node.left:
queue.append(node.left)
    return result
```

B. Sliding Window Maximum

```
from collections import deque
def max_sliding_window(nums, k):
    dq, res = deque(), []
    for i, num in enumerate(nums):
        while dq and nums[dq[-1]] < num:
            dq.pop()
        dq.append(i)
        if dq[0] == i - k:
            dq.popleft()
        if i >= k - 1:
            res.append(nums[dq[0]])
    return res
```

C. Implement Stack with Queues

```
from collections import deque
class MyStack:
    def __init__(self):
        self.q = deque()
    def push(self, x):
        self.q.append(x)
        for _ in range(len(self.q)-1):
            self.q.append(self.q.popleft())
    def pop(self):
        return self.q.popleft()
    def top(self):
        return self.q[0]
    def empty(self):
        return len(self.q) == 0
```

D. Implement Queue with Stacks

```
class MyQueue:
    def __init__(self):
        self.stack_in, self.stack_out = [], []
    def enqueue(self, x):
        self.stack_in.append(x)
    def dequeue(self):
        if not self.stack_out:
            while self.stack_in:
                self.stack_out.append(self.stack_in.pop())
        return self.stack_out.pop()
```

Tips & Best Practices

- Use `collections.deque` for both stacks and queues in Python—fast, safe, supports all needed O(1) operations.
- For DSA interviews, always implement stack/queue from scratch if required.
- Stacks: watch for underflow (empty pop).
- Queues: always check for empty before dequeue.
- Practice: Leetcode, GeeksforGeeks “Stack” and “Queue” tags have hundreds of real interview patterns.