CoGrammar

Welcome to this session:

Coding Interview Workshop - Stacks and Queues

The session will start shortly...

Questions? Drop them in the chat. We'll have dedicated moderators answering questions.

Safeguarding & Welfare

We are committed to all our students and staff feeling safe and happy; we want to make sure there is always someone you can turn to if you are worried about anything.

If you are feeling upset or unsafe, are worried about a friend, student or family member, or you feel like something isn't right, speak to our safeguarding team:

Ian Wyles
Designated Safeguarding Lead

Simone Botes

Nurhaan Snyman

Rafiq Manan

Ronald Munodawafa

Tevin Pitts

Scan to report a safeguarding concern

or email the Designated Safeguarding Lead:
Ian Wyles
safeguarding@hyperiondev.com

CoGrammar HyperionDev

Skills Bootcamp Coding Interview Workshop

- The use of disrespectful language is prohibited in the questions, this is a supportive, learning environment for all - please engage accordingly. **(Fundamental British Values: Mutual Respect and Tolerance)**
- No question is daft or silly - **ask them!**
- There are **Q&A sessions** midway and at the end of the session, should you wish to ask any follow-up questions. Moderators are going to be answering questions as the session progresses as well.
- If you have any questions outside of this lecture, or that are not answered during this lecture, please do submit these for upcoming Academic Sessions. You can submit these questions here: **Questions**

Skills Bootcamp Coding Interview Workshop

- For all **non-academic questions**, please submit a query:
  www.hyperiondev.com/support
- **Report a safeguarding incident: *www.hyperiondev.com/safeguardreporting***
- We would love your feedback on lectures: **Feedback on Lectures**
- If you are hearing impaired, please kindly use your computer's function through Google chrome to enable captions.

Learning Outcomes

- ❖ **Describe and implement stack and queue** (normal and priority) data structures in Python, understanding their use cases.
- ❖ **Implement the basic operations** associated with each data structure and compute and discuss the time and space complexities associated with each.
- ❖ **Explain the distinction between the stack and heap memory models** and their implications for memory management in Python.
- ❖ **Demonstrate how to use Stacks and Queues to solve common coding problems**, including detecting cycles.

What is the time complexity of the 'push' and 'pop' operations in a typical stack implementation?

A. O(1) for both push and pop

B. O(n) for push and O(1) for pop

C. O(1) for push and O(n) for pop

D. O(n) for both push and pop

CoGrammar

Which of the following data structures is typically used to implement a simple queue?

A. Array

B. Linked List

C. Hash Table

D. Both Arrays and Linked Lists

CoGrammar

What is a distinguishing feature of a priority queue compared to a standard queue?

A. Elements are removed based on their priority rather than their order in the queue.

B. Elements are always sorted in ascending order.

C. Elements can only be numeric values.

D. Priority queues have faster access times than standard queues.

CoGrammar

Lecture Overview

- → Task Management System
- → Introduction
- → Stacks
- → Queues
- → Priority Queues
- → Memory Management

CoGrammar

CoGrammar

Task Management

Consider a software development team working on multiple projects at the same time. Each developer is assigned tasks from each project and these tasks need to be stored somewhere.

- ➢ For one group of projects, tasks need to assigned in the order that they were submitted.
- ➢ Another group of projects require that the newest tasks be assigned first.
- ➢ The other group of projects have prioritised tasks and tasks with the highest priority need to be assigned first.

Task Management

- How can we use different data structures to organise the different lists of tasks in the most efficient way?
- How do we efficiently organise data that will be removed from the structure in a particular order?
- How do we guarantee that data will be removed and added to the structure in the correct manner?
- How can we evaluate the efficiency of these data structures?

CoGrammar

Example: Organising Tasks

- **Project A:** This project has been released for a while and users may suggest different features to be added to the project.
  - *These suggestions can be added to a "queue". Whichever suggestion was made first would be added to the project first.*
- **Project B:** This project is in the process of being developed and suggestions of the most useful features are made by the developers throughout the development of the project.
  - *Features can be added to a "stack". Whichever feature was most recently suggested would be added to the project first.*
- **Project C**: This project has just been released and a lot of users are experiencing bugs of varying severity.
  - *Issues can be added to a "priority queue" organised by the severity of the bug. Issues that are the most severe would have the highest priority.*

Stacks and Queues

Types of linear data structures that allow for storage and retrieval of data based on a specific method of ordering.

- ❖ Data structures allow us to **organise storage** so that data can be accessed **faster and more efficiently**.
- ❖ Stacks and queues are simple, easy to implement and widely applicable.
- ❖ Each has defined methods of **ordering** and **operations** for adding and removing elements to and from the structure.
- ❖ Can be implemented using an Array, Linked List or the `deque` or `queue` modules in Python.

CoGrammar

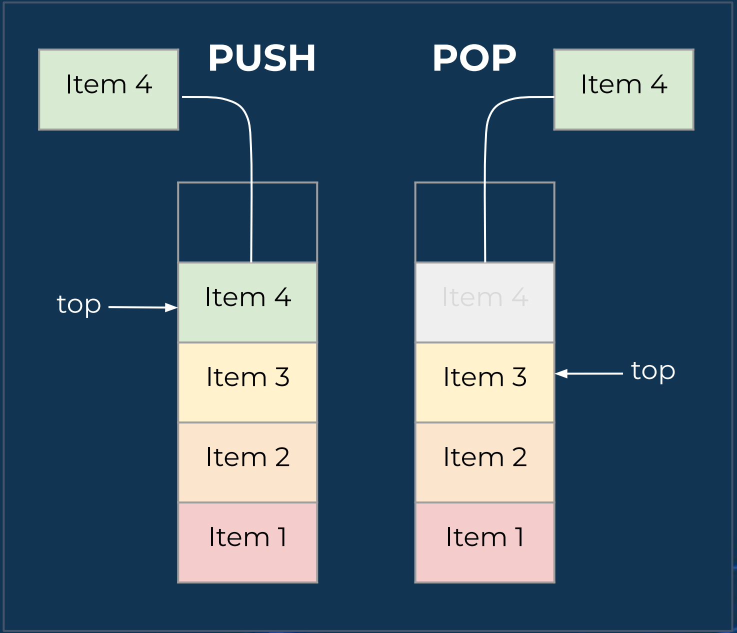
Stacks

Method of Ordering

- **LIFO:** Last element added to the stack is the first to be removed
- Elements are added on **top** of one another in a “stack”
- A pointer points to the element at the **top** of the stack

Operations

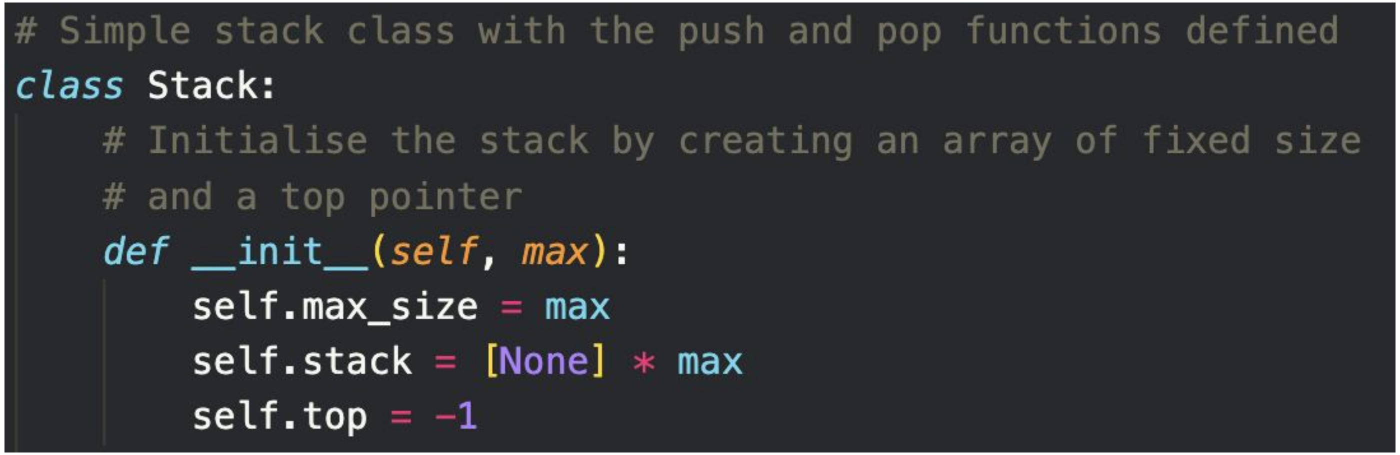
- **Push:** Adds an element to the top of the stack
- **Pop:** Removes the element from the top of the stack



CoGrammar

CoGrammar

Stacks: Array Implementation

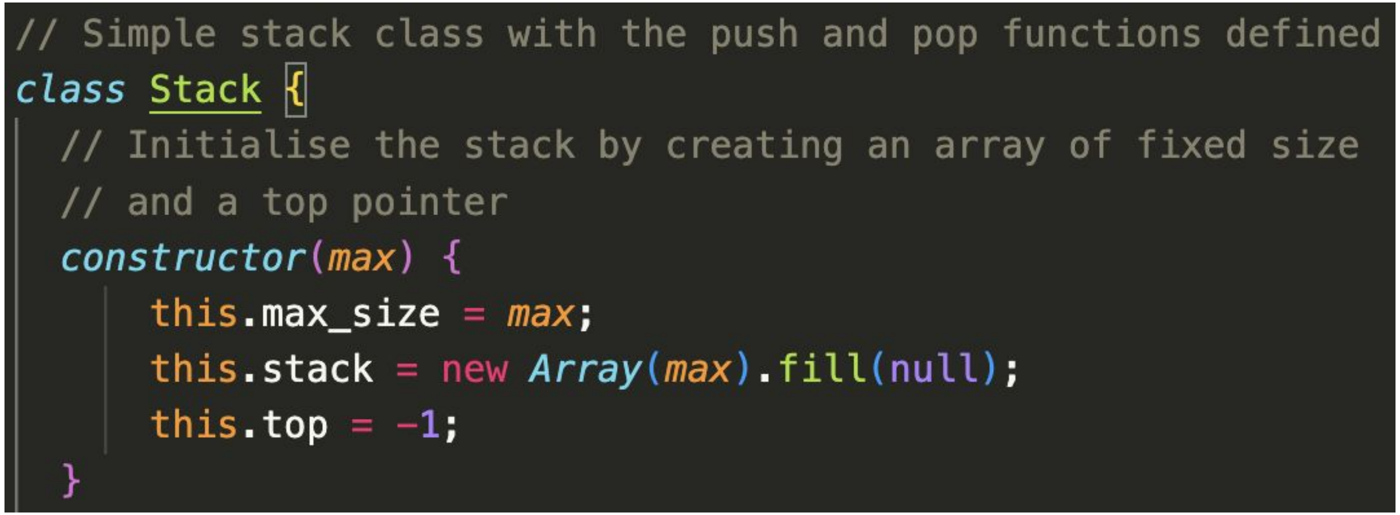


```python
# Simple stack class with the push and pop functions defined
class Stack:
    # Initialise the stack by creating an array of fixed size
    # and a top pointer
    def __init__(self, max):
        self.max_size = max
        self.stack = [None] * max
        self.top = -1
```

Note: This implementation is the most efficient. Be careful when attempting to implement a stack using the built-in insert() and pop() methods since it's slower to pop or insert an element at any other position besides the end of the list [O(n)].

CoGrammar

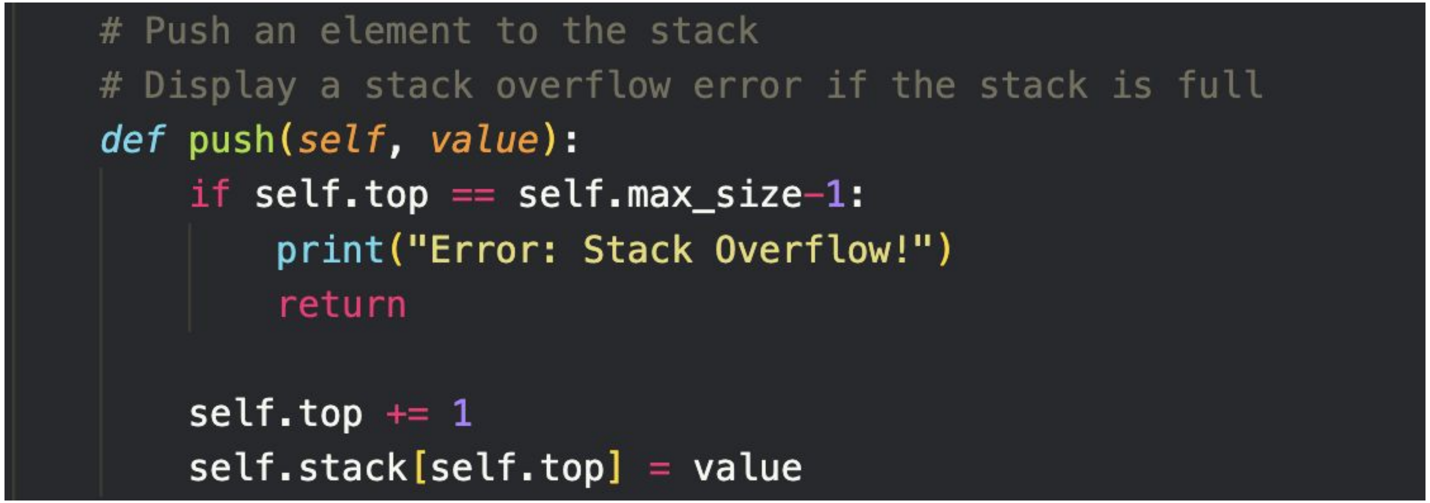
Stacks: Array Implementation



```
// Simple stack class with the push and pop functions defined
class Stack {
  // Initialise the stack by creating an array of fixed size
  // and a top pointer
  constructor(max) {
    this.max_size = max;
    this.stack = new Array(max).fill(null);
    this.top = -1;
  }
```

Note: This implementation is the most efficient. Be careful when attempting to implement a stack using the built-in insert() and pop() methods since it's slower to pop or insert an element at any other position besides the end of the list [O(n)].

Stacks: Array Implementation

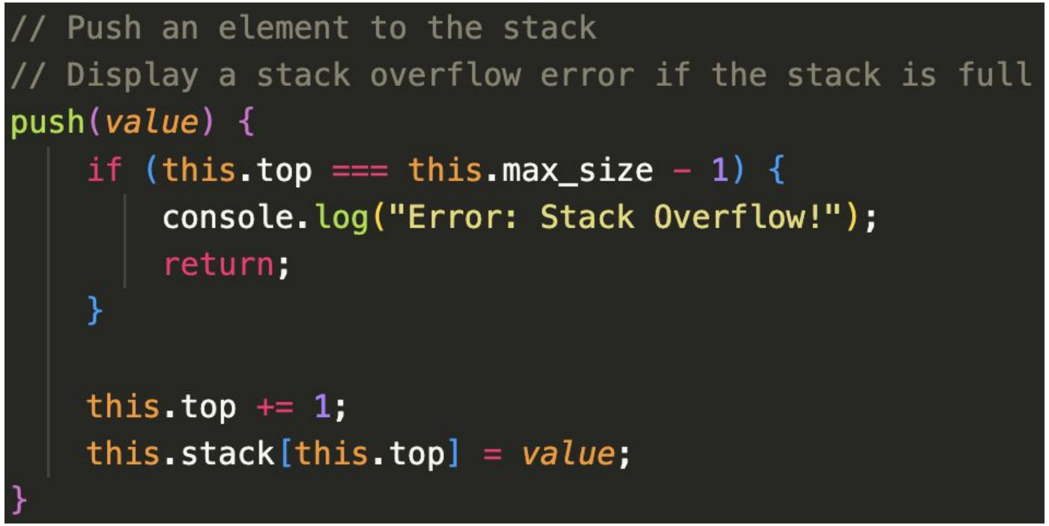


```python
# Push an element to the stack
# Display a stack overflow error if the stack is full
def push(self, value):
    if self.top == self.max_size-1:
        print("Error: Stack Overflow!")
        return

    self.top += 1
    self.stack[self.top] = value
```

Note: This implementation is the most efficient. Be careful when attempting to implement a stack using the built-in insert() and pop() methods since its slower to pop or insert an element at any other position besides the end of the list [O(n)].

CoGrammar

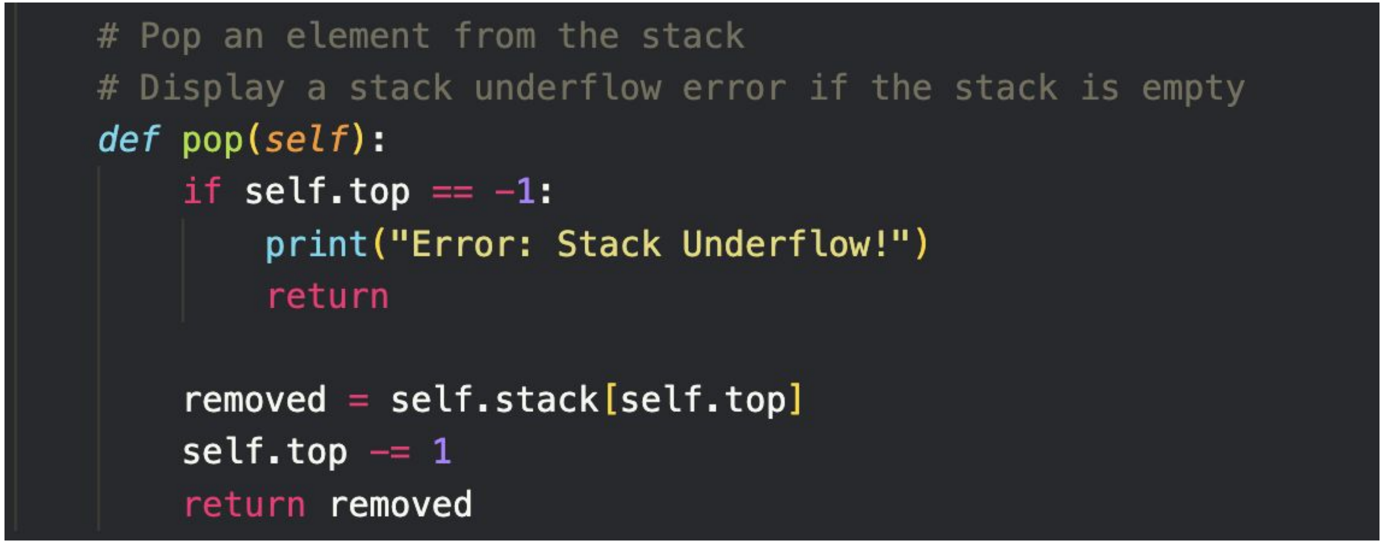
Stacks: Array Implementation



```
// Push an element to the stack
// Display a stack overflow error if the stack is full
push(value) {
    if (this.top === this.max_size - 1) {
        console.log("Error: Stack Overflow!");
        return;
    }

    this.top += 1;
    this.stack[this.top] = value;
}
```

Note: This implementation is the most efficient. Be careful when attempting to implement a stack using the built-in insert() and pop() methods since its slower to pop or insert an element at any other position besides the end of the list [O(n)].

Stacks: Array Implementation

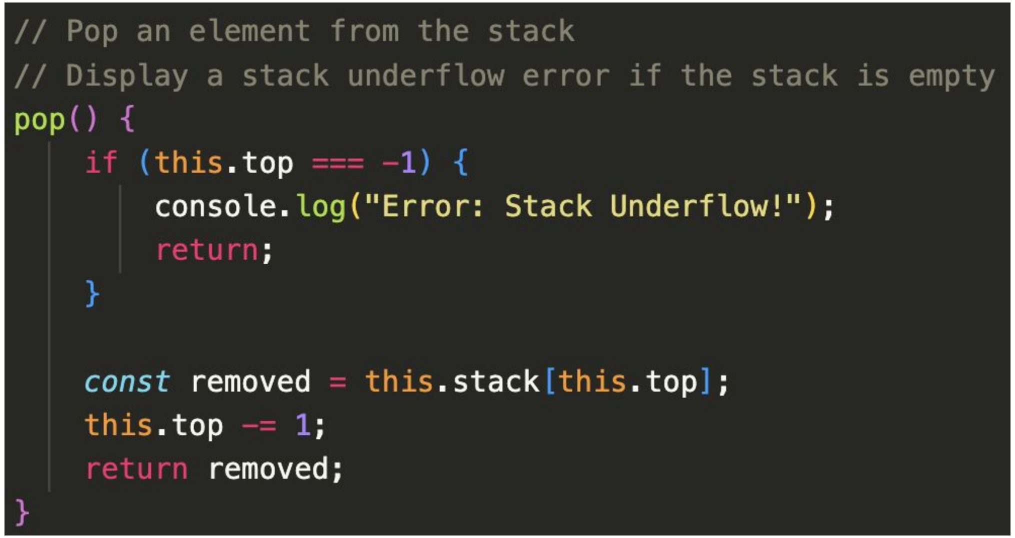


```python
# Pop an element from the stack
# Display a stack underflow error if the stack is empty
def pop(self):
    if self.top == -1:
        print("Error: Stack Underflow!")
        return

    removed = self.stack[self.top]
    self.top -= 1
    return removed
```

Note: This implementation is the most efficient. Be careful when attempting to implement a stack using the built-in insert() and pop() methods since its slower to pop or insert an element at any other position besides the end of the list [O(n)].

CoGrammar

Stacks: Array Implementation



```
// Pop an element from the stack
// Display a stack underflow error if the stack is empty
pop() {
    if (this.top === -1) {
        console.log("Error: Stack Underflow!");
        return;
    }

    const removed = this.stack[this.top];
    this.top -= 1;
    return removed;
}
```

Note: This implementation is the most efficient. Be careful when attempting to implement a stack using the built-in insert() and pop() methods since its slower to pop or insert an element at any other position besides the end of the list [O(n)].

Stacks

Complexity Analysis

- **Push**
  - **Space: O(1)**
    No extra space is used
  - **Time: O(1)**
    A single memory allocation done in constant time

- **Pop**
  - **Space: O(1)**
    No extra space is used
  - **Time: O(1)**
    Pointer is decremented by 1

Common Uses

- Function and Recursive Calls
- Undo and Redo Mechanisms
- “Most recently used” features
- Backtracking algorithms
- Expression evaluations and syntax parsing

CoGrammar

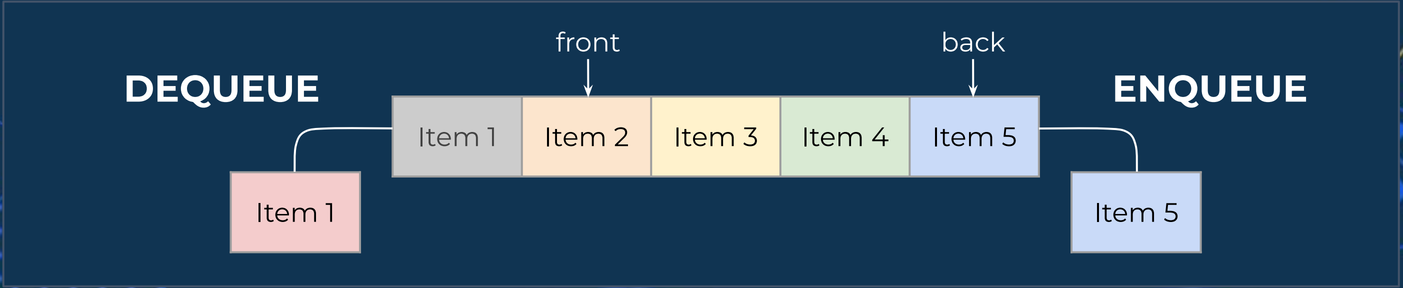
Queues

Method of Ordering

- **FIFO:** First element added to the stack is the first to be removed
- Elements added to **rear** of a “queue” and removed from **front**
- Two pointers point to the **front** and the **rear**

Operations

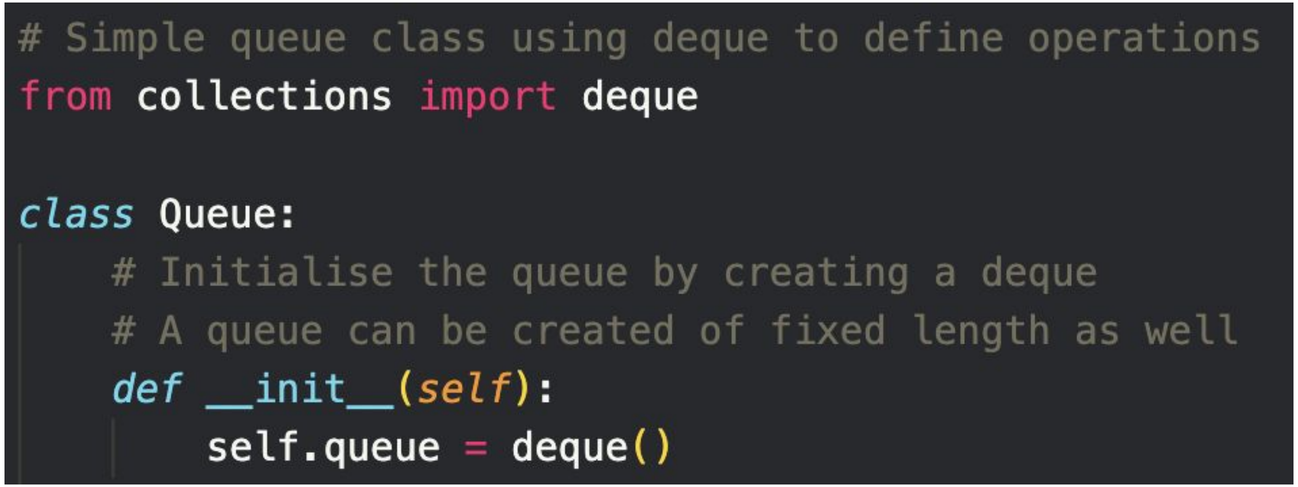
- **Enqueue:** Adds an element to the end of the queue
- **Dequeue:** Removes the element from the front of the queue



CoGrammar

Queues: deque Implementation

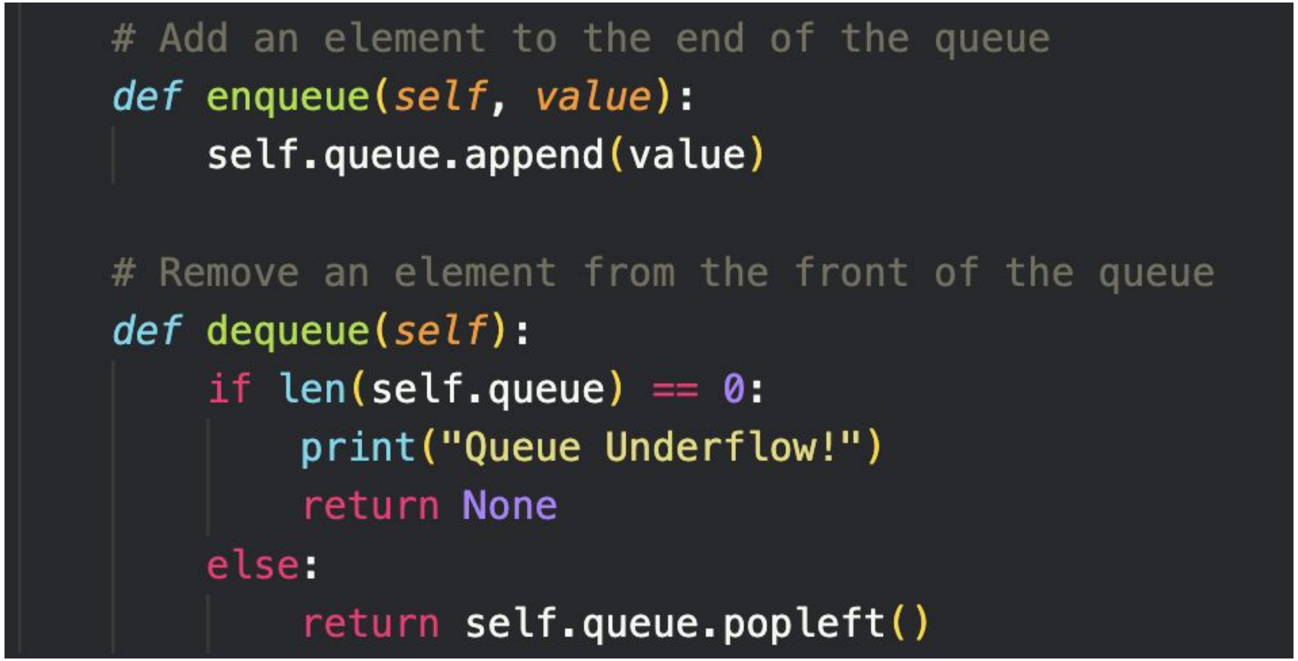
- `deque` makes use of a doubly-linked list



```python
# Simple queue class using deque to define operations
from collections import deque

class Queue:
    # Initialise the queue by creating a deque
    # A queue can be created of fixed length as well
    def __init__(self):
        self.queue = deque()
```

Note: Lists can be used to implement a queue, but this will either come at a cost of time, since using the pop(n) function has an O(n) time complexity, or at a cost of space, since using pointers would mean that the list would grow infinitely but also have empty elements in the front.

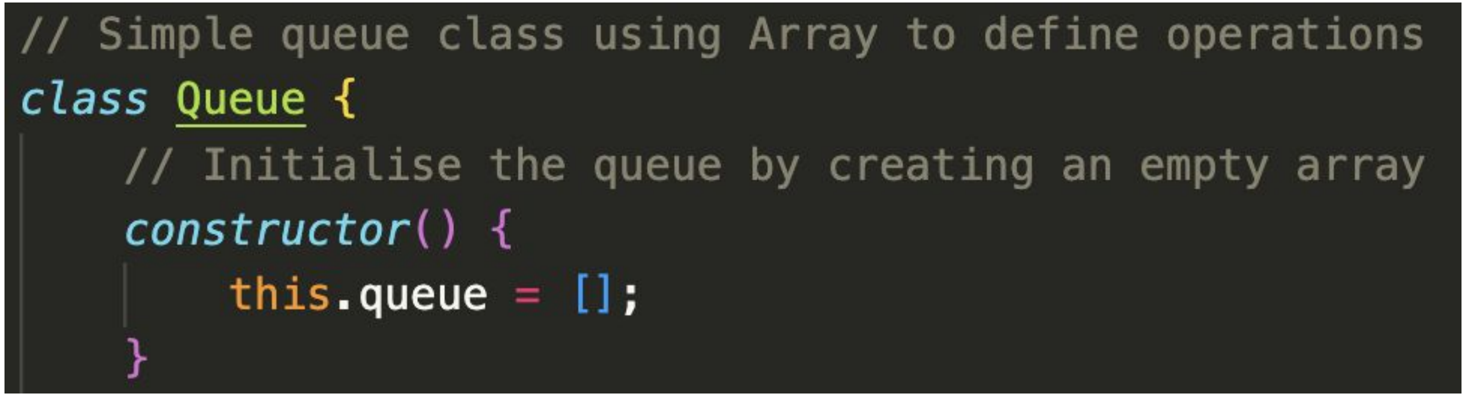
Queues: deque Implementation



```python
    # Add an element to the end of the queue
    def enqueue(self, value):
        self.queue.append(value)

    # Remove an element from the front of the queue
    def dequeue(self):
        if len(self.queue) == 0:
            print("Queue Underflow!")
            return None
        else:
            return self.queue.popleft()
```

CoGrammar

Queues: deque Implementation

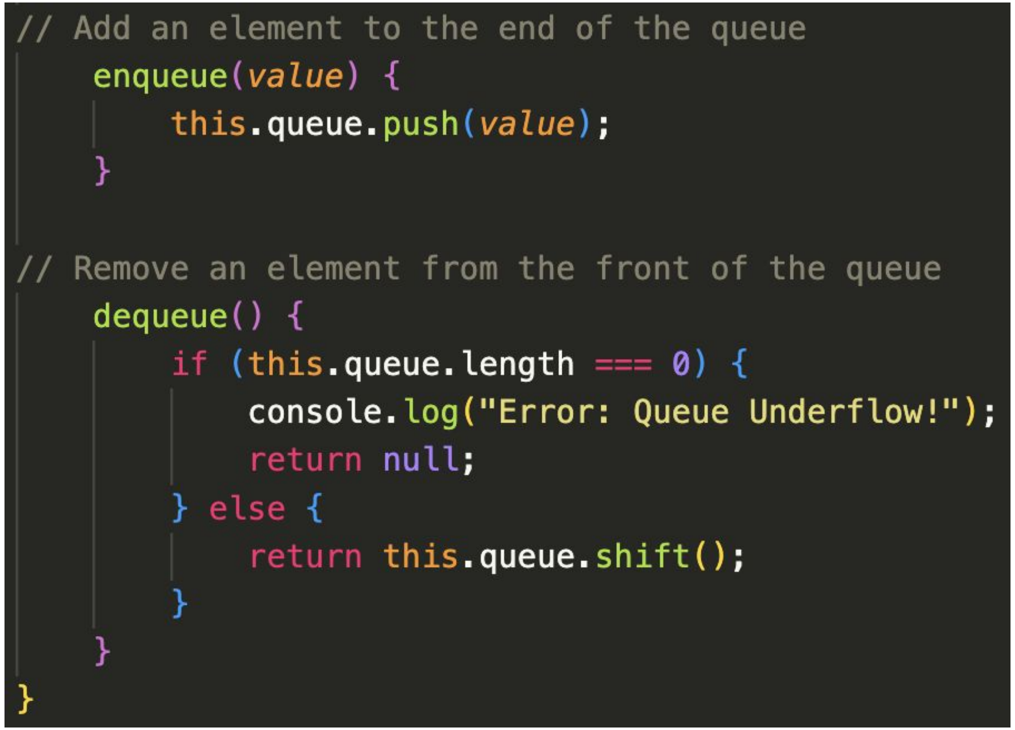


```
// Simple queue class using Array to define operations
class Queue {
    // Initialise the queue by creating an empty array
    constructor() {
        this.queue = [];
    }
```

Note: Lists can be used to implement a queue, but this will either come at a cost of time, since using the pop(n) function has an O(n) time complexity, or at a cost of space, since using pointers would mean that the list would grow infinitely but also have empty elements in the front.

CoGrammar

Queues: deque Implementation



```javascript
// Add an element to the end of the queue
    enqueue(value) {
        this.queue.push(value);
    }

// Remove an element from the front of the queue
    dequeue() {
        if (this.queue.length === 0) {
            console.log("Error: Queue Underflow!");
            return null;
        } else {
            return this.queue.shift();
        }
    }
}
```

Queues

Complexity Analysis

- **Enqueue**
  - **Space: O(1)**
    No extra space used
  - **Time: O(1)**
    Single memory allocation done in constant time

- **Dequeue**
  - **Space: O(1)**
    No extra space used
  - **Time: O(1)**
    Front pointer incremented by 1 and node deallocated

Common Uses

- Task Scheduling
- Resource Allocation
- Network Protocols a
- Printing Queues
- Web Servers
- Breadth-First Search

CoGrammar

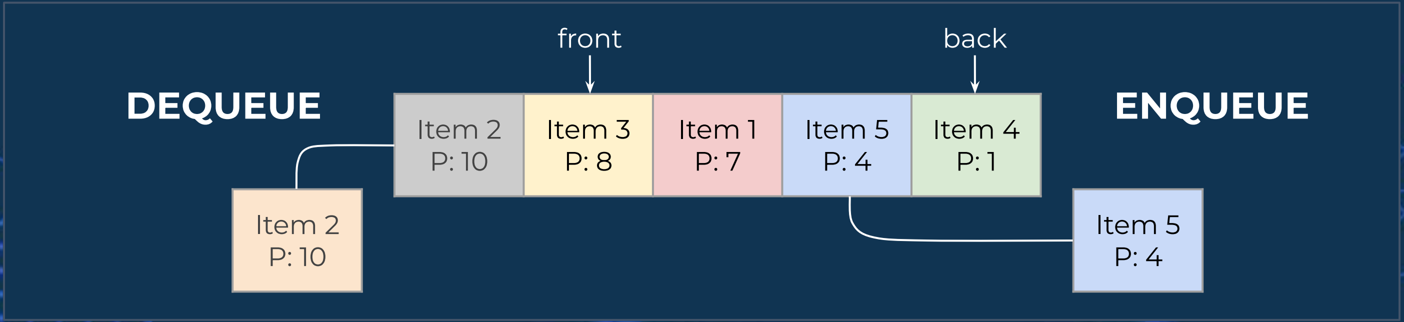
Priority Queues

Method of Ordering

- Arranged according to the assigned **priority** of elements
- Order direction doesn't matter, as long as **the highest priority elements are removed first**

Operations

- **Enqueue:** Adds an element to the queue based on its priority
- **Dequeue:** Removes the highest priority element from the queue



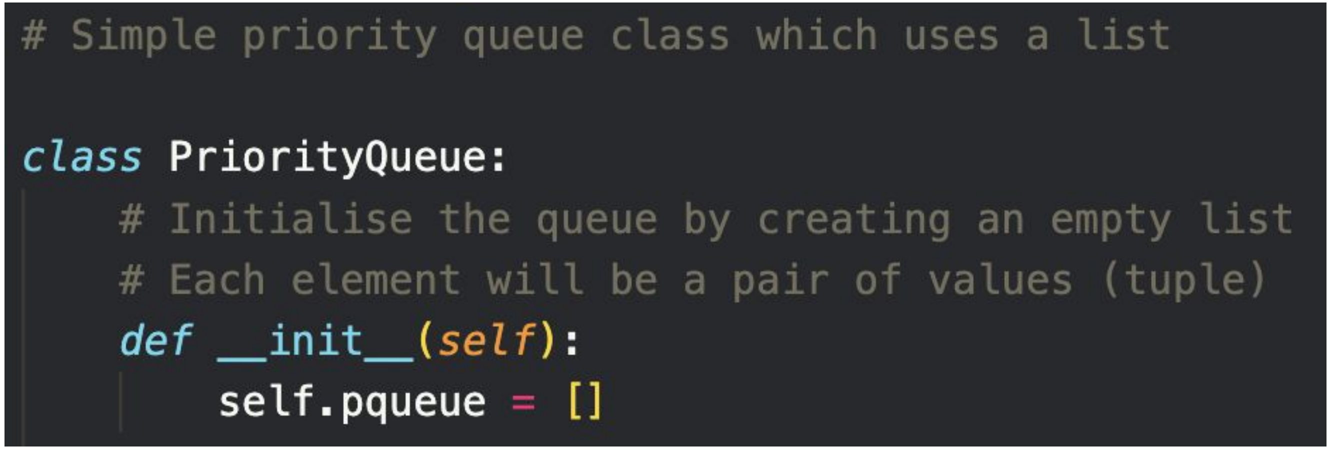
CoGrammar

Let’s Breathe!

Let’s take a small break before moving on to the next topic.

CoGrammar

Priority Queues: List Implementation

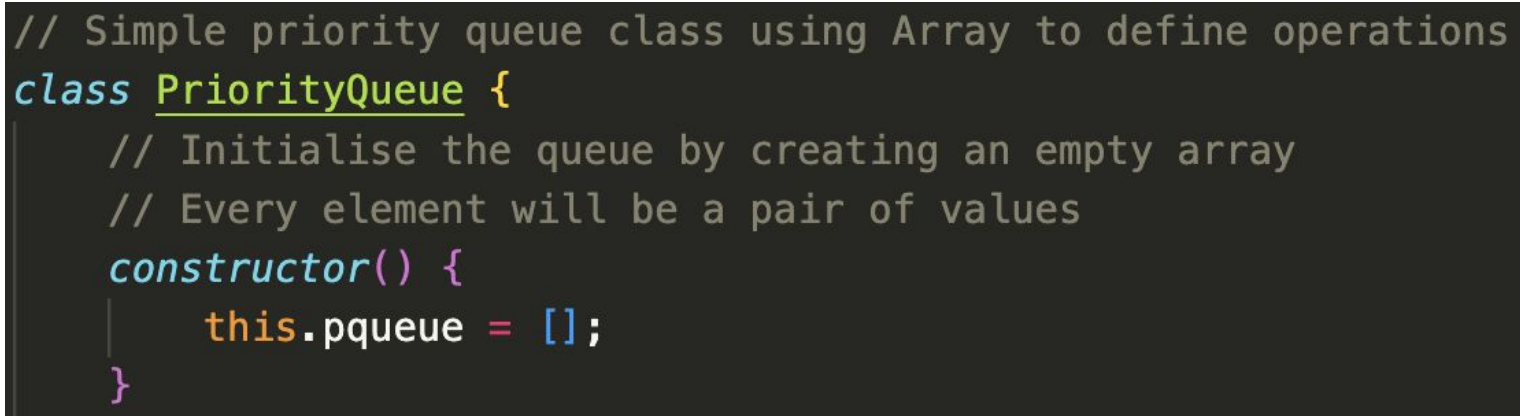


```python
# Simple priority queue class which uses a list

class PriorityQueue:
    # Initialise the queue by creating an empty list
    # Each element will be a pair of values (tuple)
    def __init__(self):
        self.pqueue = []
```

Note: The sort function is used at all times so elements are added to a sorted list which improves time complexities. The list is sorted in ascending order so that the element at the end of the list has the highest priority thus the pop() will have a O(1) complexity.

CoGrammar

Priority Queues: List Implementation

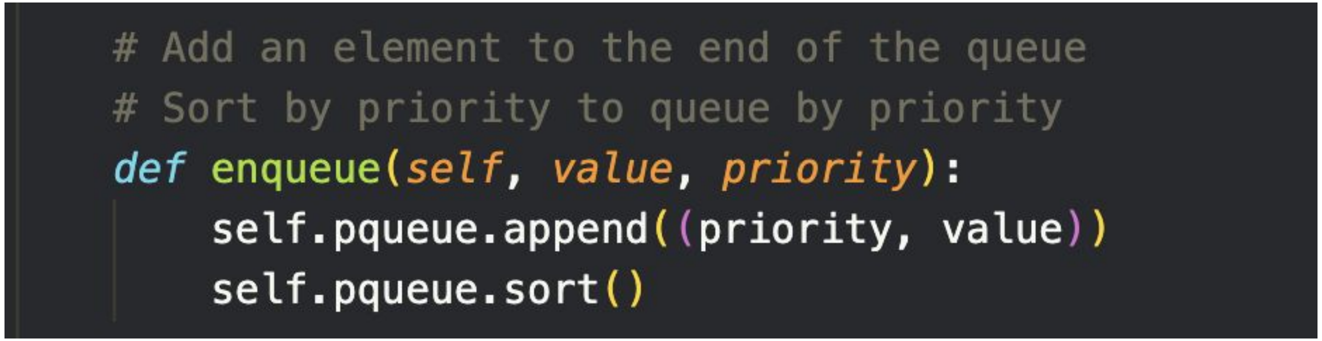


```
// Simple priority queue class using Array to define operations
class PriorityQueue {
    // Initialise the queue by creating an empty array
    // Every element will be a pair of values
    constructor() {
        this.pqueue = [];
    }
```

Note: The sort function is used at all times so elements are added to a sorted list which improves time complexities. The list is sorted in ascending order so that the element at the end of the list has the highest priority thus the pop() will have a O(1) complexity.

CoGrammar

Priority Queues: List Implementation

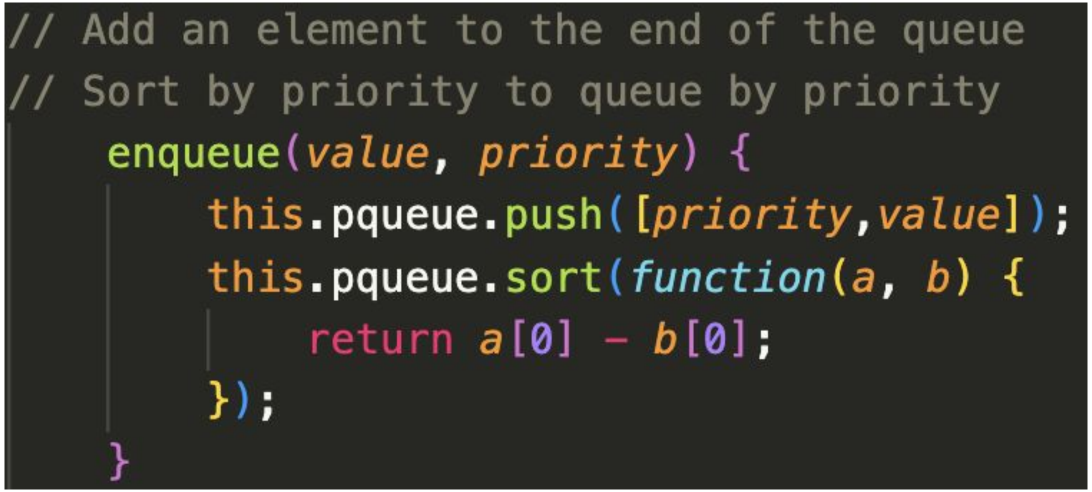


```python
# Add an element to the end of the queue
# Sort by priority to queue by priority
def enqueue(self, value, priority):
    self.pqueue.append((priority, value))
    self.pqueue.sort()
```

Note: The sort function is used at all times so elements are added to a sorted list which improves time complexities. The list is sorted in ascending order so that the element at the end of the list has the highest priority thus the pop() will have a O(1) complexity.

CoGrammar

Priority Queues: List Implementation

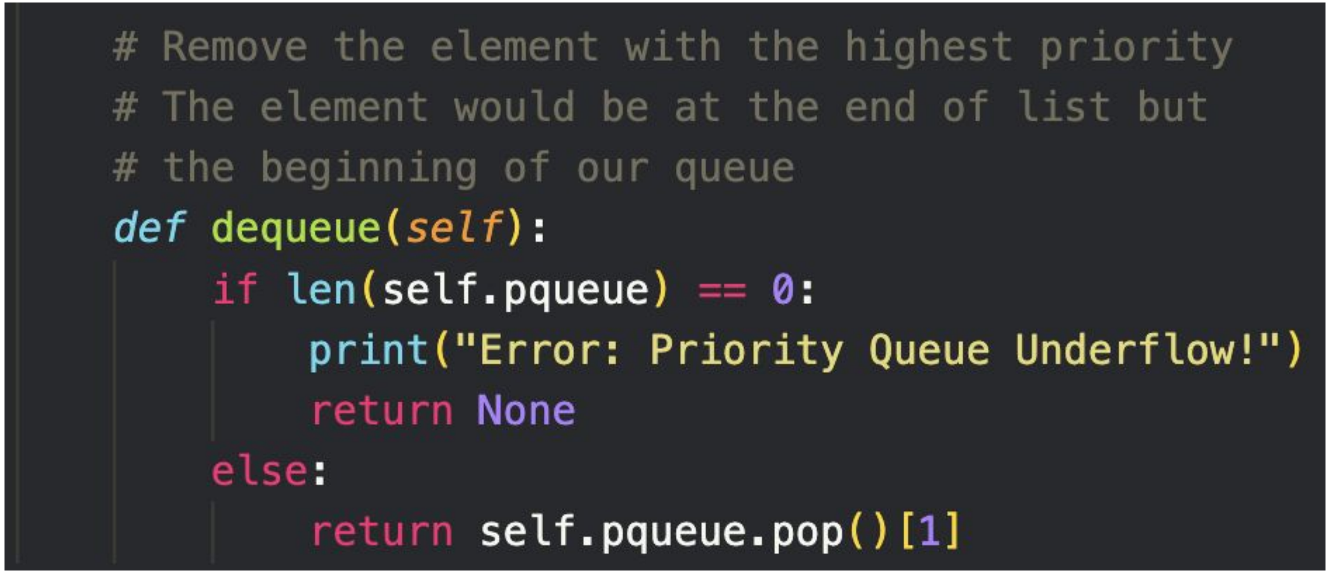


```
// Add an element to the end of the queue
// Sort by priority to queue by priority
    enqueue(value, priority) {
        this.pqueue.push([priority,value]);
        this.pqueue.sort(function(a, b) {
            return a[0] - b[0];
        });
    }
```

Note: The sort function is used at all times so elements are added to a sorted list which improves time complexities. The list is sorted in ascending order so that the element at the end of the list has the highest priority thus the pop() will have a O(1) complexity.

CoGrammar

Priority Queues: List Implementation

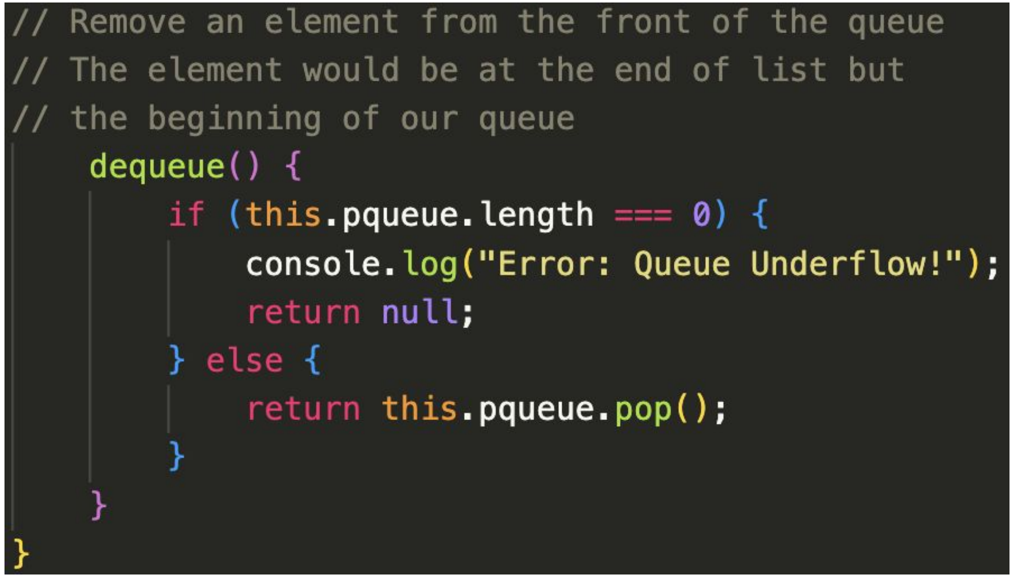


```python
# Remove the element with the highest priority
# The element would be at the end of list but
# the beginning of our queue
def dequeue(self):
    if len(self.pqueue) == 0:
        print("Error: Priority Queue Underflow!")
        return None
    else:
        return self.pqueue.pop()[1]
```

Note: The sort function is used at all times so elements are added to a sorted list which improves time complexities. The list is sorted in ascending order so that the element at the end of the list has the highest priority thus the pop() will have a O(1) complexity.

CoGrammar

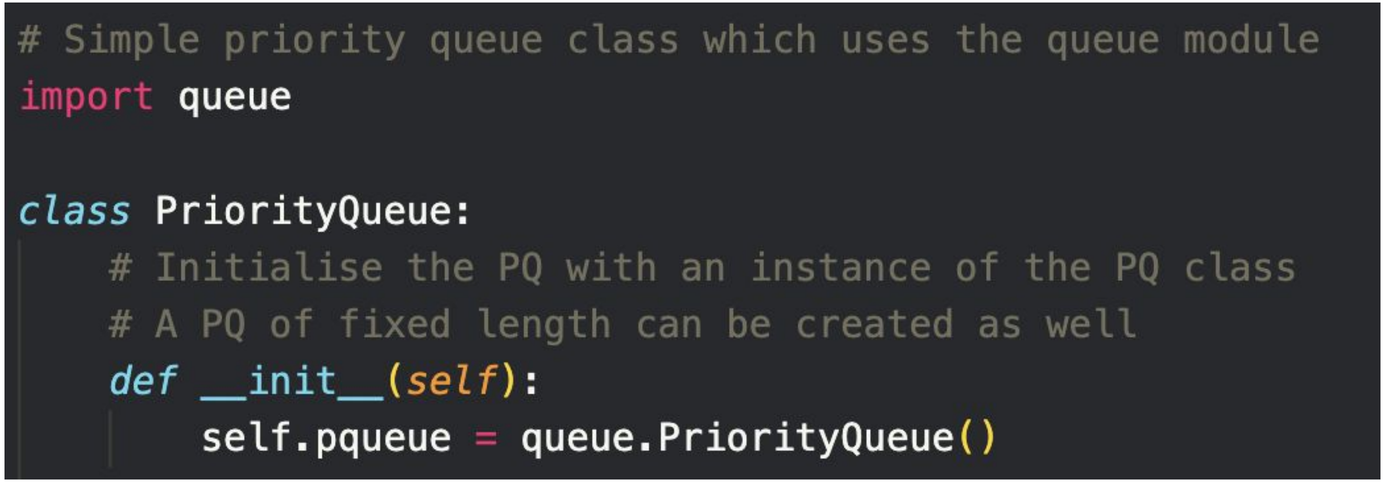
Priority Queues: List Implementation



```
// Remove an element from the front of the queue
// The element would be at the end of list but
// the beginning of our queue
    dequeue() {
        if (this.pqueue.length === 0) {
            console.log("Error: Queue Underflow!");
            return null;
        } else {
            return this.pqueue.pop();
        }
    }
}
```

Note: The sort function is used at all times so elements are added to a sorted list which improves time complexities. The list is sorted in ascending order so that the element at the end of the list has the highest priority thus the pop() will have a O(1) complexity.

Priority Queues: `queue` Implementation

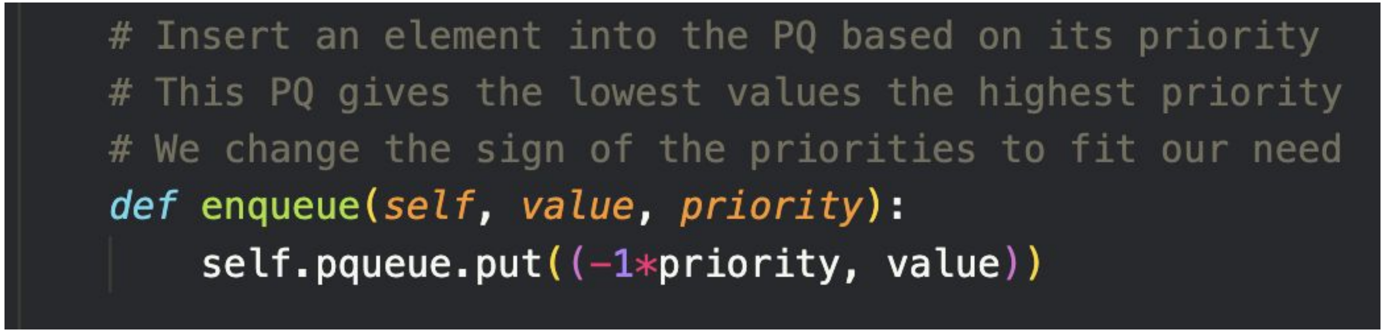


```
# Simple priority queue class which uses the queue module
import queue

class PriorityQueue:
    # Initialise the PQ with an instance of the PQ class
    # A PQ of fixed length can be created as well
    def __init__(self):
        self.pqueue = queue.PriorityQueue()
```

Note: This implementation makes use of a structure called a min-heap to improve efficiency. A min-heap maintains that all parent nodes are less than or equal to all child nodes.

CoGrammar

Priority Queues: `queue` Implementation

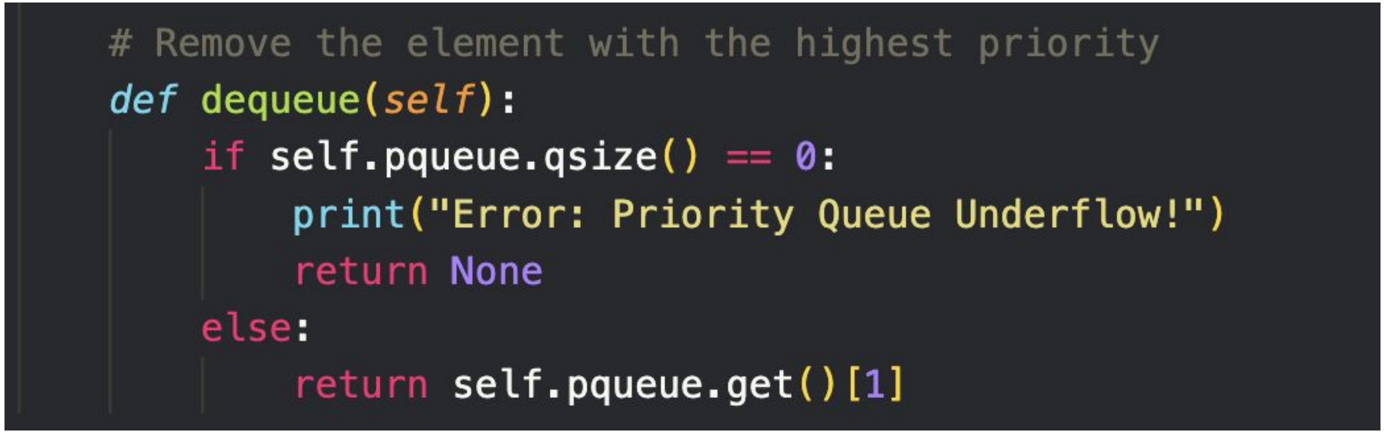


```python
# Insert an element into the PQ based on its priority
# This PQ gives the lowest values the highest priority
# We change the sign of the priorities to fit our need
def enqueue(self, value, priority):
    self.pqueue.put((-1*priority, value))
```

Note: This implementation makes use of a structure called a min-heap to improve efficiency. A min-heap maintains that all parent nodes are less than or equal to all child nodes.

CoGrammar

Priority Queues: `queue` Implementation



```python
# Remove the element with the highest priority
def dequeue(self):
    if self.pqueue.qsize() == 0:
        print("Error: Priority Queue Underflow!")
        return None
    else:
        return self.pqueue.get()[1]
```

Note: This implementation makes use of a structure called a mini-heap to improve efficiency. A min-heap maintains that all parent nodes are less than or equal to all child nodes.

Priority Queues

<u>**Complexity Analysis**</u>

- **Enqueue**
  - **Space:**
    - **List - O(1)**

      No extra space is used
    - **PriorityQueue - O(1)**

      No extra space is used
  - **Time:**
    - **List - O(n)**

      Each element's priority must be checked and compared
    - **PriorityQueue - O(log n)**

      Adding node to heap

Dequeue

- **Space:**
  - **List - O(1)**

    No extra space is used
  - **PriorityQueue - O(1)**

    No extra space is used
- **Time:**
  - **List - O(1)**

    Pointer is updated or last element is removed
  - **PriorityQueue - O(log n)**

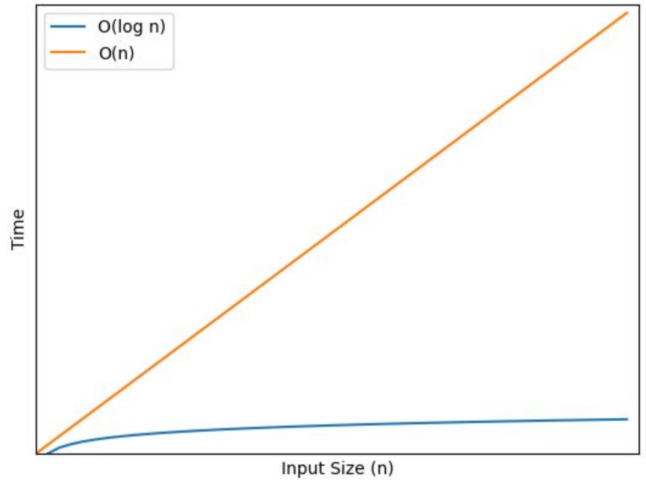
    Removing node from heap

CoGrammar

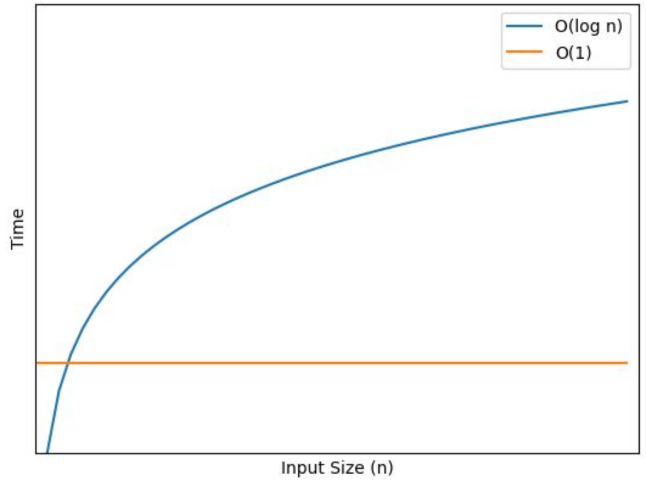
Priority Queues

Complexity Analysis Visualisation

Enqueue Time Complexity



Dequeue Time Complexity

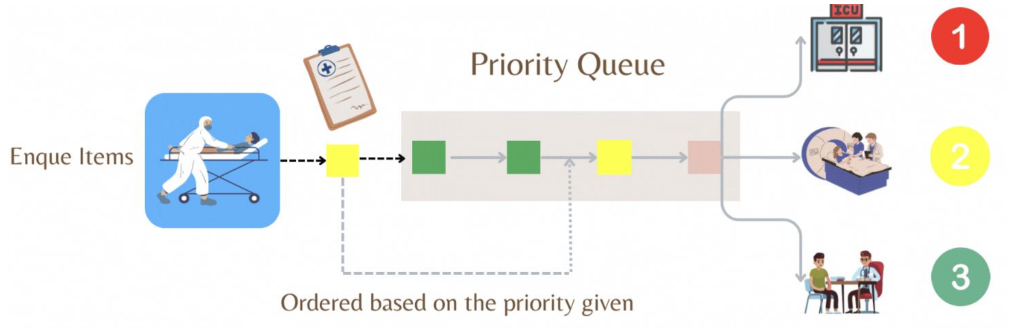


CoGrammar

Priority Queues

Common Uses

- Dijkstra's Shortest Path Algorithm
- Data Compression
- Artificial Intelligence
- Load Balancing in OSs
- Optimisation Problems
- Robotics
- Medical Systems
- Event-driven simulations



Source: Priority Queues in Healthcare (Medium)

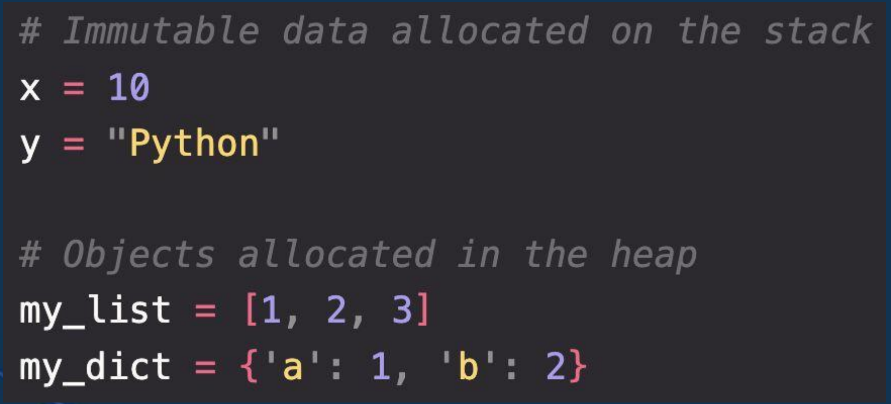
Memory Management

The function responsible for managing the computer's primary memory. It tracks the status of each memory location, either allocated or free, and decides how memory is allocated and deallocated among competing processes.

- ❖ In Python and JavaScript, memory is managed in two key areas: the **stack** and the **heap**.
- ❖ The **stack** is used for **static memory allocation**, which includes local variables and function calls.
- ❖ The heap is used for dynamic memory allocation, which is necessary for objects that need to persist outside the scope of a single function call.

CoGrammar

Memory Allocation



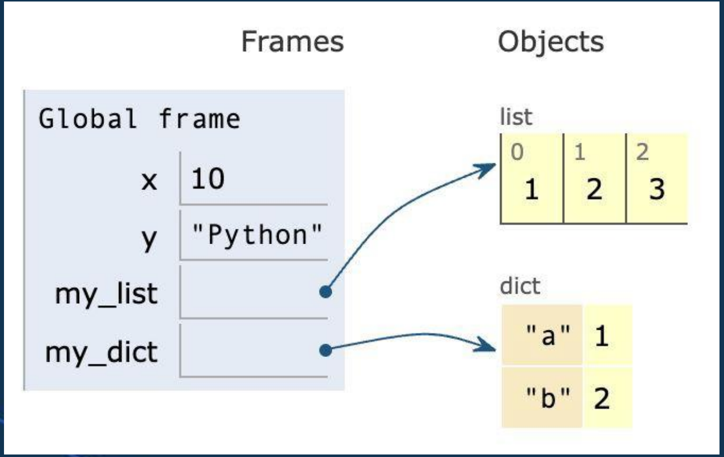
```python
# Immutable data allocated on the stack
x = 10
y = "Python"

# Objects allocated in the heap
my_list = [1, 2, 3]
my_dict = {'a': 1, 'b': 2}
```

Memory for variables, integers or string is typically allocated on the stack. This space is automatically managed and is efficient for variables that have a short lifespan.

For more complex data structures, lists, dictionaries, and class instances, Python allocates memory on the heap. These structures are usually larger and live longer than simple, stack-allocated variables.

CoGrammar

Local Variables vs Objects



The "Global frame" in the visualization represents the stack, holding simple, quickly accessed variables like x and y, whose lifecycle is tied to their scope of declaration.

The "Objects" section signifies the heap, where complex objects such as my_list and my_dict are stored, referenced by variables in the stack and managed dynamically for persistent and flexible memory allocation.

CoGrammar

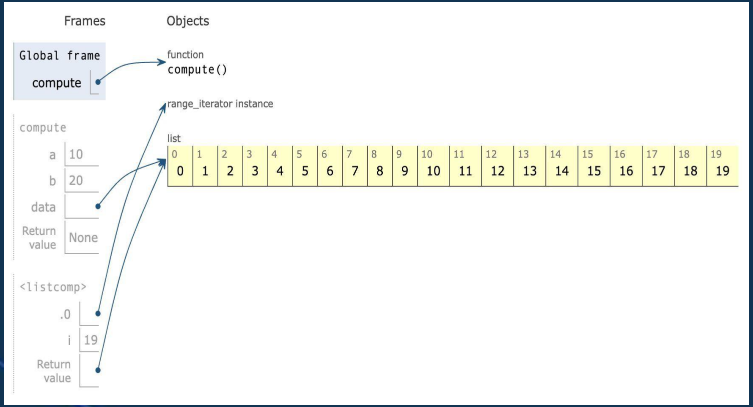
Memory Allocation

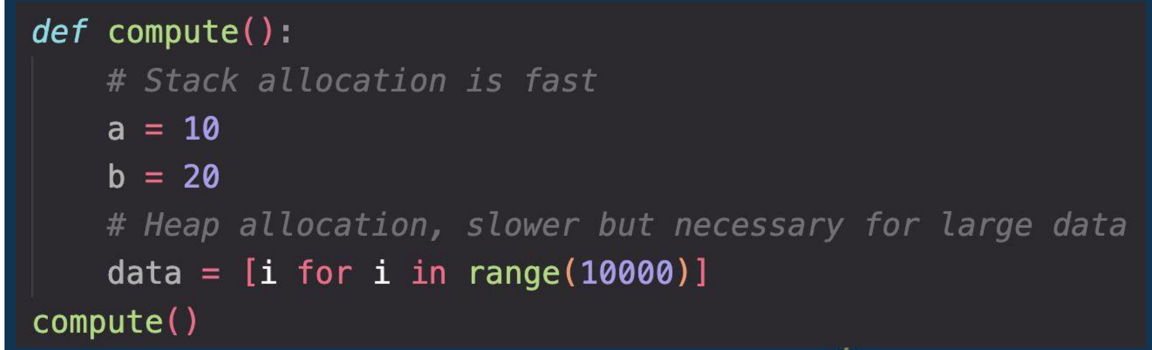
- Stack memory allocation is a fast operation. The stack works with a LIFO (Last-In-First-Out) principle, which allows for quick push and pop operations. This makes it ideal for temporary data that has a well-defined lifespan.

- Heap memory, while more flexible, requires dynamic memory allocation, which is a slower process. It involves finding a free block of memory large enough for the object, which can lead to fragmentation over time.

CoGrammar

Memory Allocation





```python
def compute():
    # Stack allocation is fast
    a = 10
    b = 20
    # Heap allocation, slower but necessary for large data
    data = [i for i in range(10000)]
compute()
```

Notice how much longer it takes to store the data object when we walk through the storage process compared to the variables, and how much simpler the stack storage looks compared to the heap storage.

Which data structure would be most appropriate for implementing a backtracking algorithm?

A. Stack

B. Queue

C. Priority Queue

D. Linked List

CoGrammar

In a task scheduling system, why might a priority queue be preferred over a simple queue?

A. For faster access to elements.

B. To process tasks based on their priority.

C. For its Last In, First Out (LIFO) nature.

D. To minimise memory usage.

CoGrammar

Homework

Practise the skills we've developed by completing the following problems:

❖ The next slide contains two questions to test your theoretical understanding of stacks and queues in a real world scenario.

❖ We'll be going through two LeetCode examples in the lecture over the weekend, attempt them yourself and come ready with questions:
  ➢ Example 1
  ➢ Example 2

❖ Practice speaking through your solutions and explaining how you approached each problem.

Further Learning

❖ StackAbuse - Stacks and Queues Basics

❖ GeeksforGeeks - Data Structures includes in depth information and implementation of Stacks, Queues and Priority Queues

❖ GeeksforGeeks - Stacks Complexity Analysis

❖ GeeksforGeeks - Queues Complexity Analysis

Homework Example

Consider a simple scheduler designed to manage the execution of tasks of varying priority.

We will consider two different implementations, one using a stack and another which uses a priority queue.

1. Discuss the choice between a stack or a priority queue for this implementation based on each data structure's performance and flexibility.

1. Based on the answer in 1, determine under what conditions it would be better to use each structure.

1. Implement a simple scheduler using either a stack or a priority queue.

CoGrammar

Homework Example

Consider a simple scheduler designed to manage the execution of tasks of varying priority.

We will consider two different implementations, one using a stack and another which uses a priority queue.

1. Discuss the choice between a stack or a priority queue for this implementation based on each data structure's performance and flexibility.

   Performance: Better speed when using a stack [O(1) vs O(log n)] but better suited order of execution with a priority queue.

   Flexibility: Priority queues have better flexibility since priority can influence order of execution.

1. Based on the answer in 1, determine the best suited conditions for each data structure.

   Stack: Performance is important and task prioritisation is not critical

   Priority Queue: Task prioritisation critically impacts order of execution.

CoGrammar

Homework Example

Consider a simple scheduler designed to manage the execution of tasks of varying priority.

We will consider two different implementations, one using a stack and another which uses a priority queue.

3. Implement a simple scheduler using either a stack or a priority queue.

   Examples of both implementations can be found with the source code for this lecture.

CoGrammar

CoGrammar

Q & A SECTION

Please use this time to ask any questions relating to the topic, should you have any.

Thank you for attending

CoGrammar

SKILLS FOR LIFE
SKILLS BOOTCAMPS

Department for Education