Welcome to the CoGrammar Stacks and Queues, Function Objects and the Memory Model

The session will start shortly...

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

CoGrammar

Coding Interview Workshop Housekeeping

- 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**

Coding Interview Workshop Housekeeping cont.

- 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]()

Skills Bootcamp
8-Week Progression Overview

Fulfil 4 Criteria to Graduation

✅ Criterion 1: Initial Requirements

Timeframe: First 2 Weeks
Guided Learning Hours (GLH): Minimum of 15 hours
Task Completion: First four tasks

Due Date: 24 March 2024

✅ Criterion 2: Mid-Course Progress

60 Guided Learning Hours

Data Science - **13 tasks**
Software Engineering - **13 tasks**
Web Development - **13 tasks**

Due Date: 28 April 2024

CoGrammar

Skills Bootcamp Progression Overview

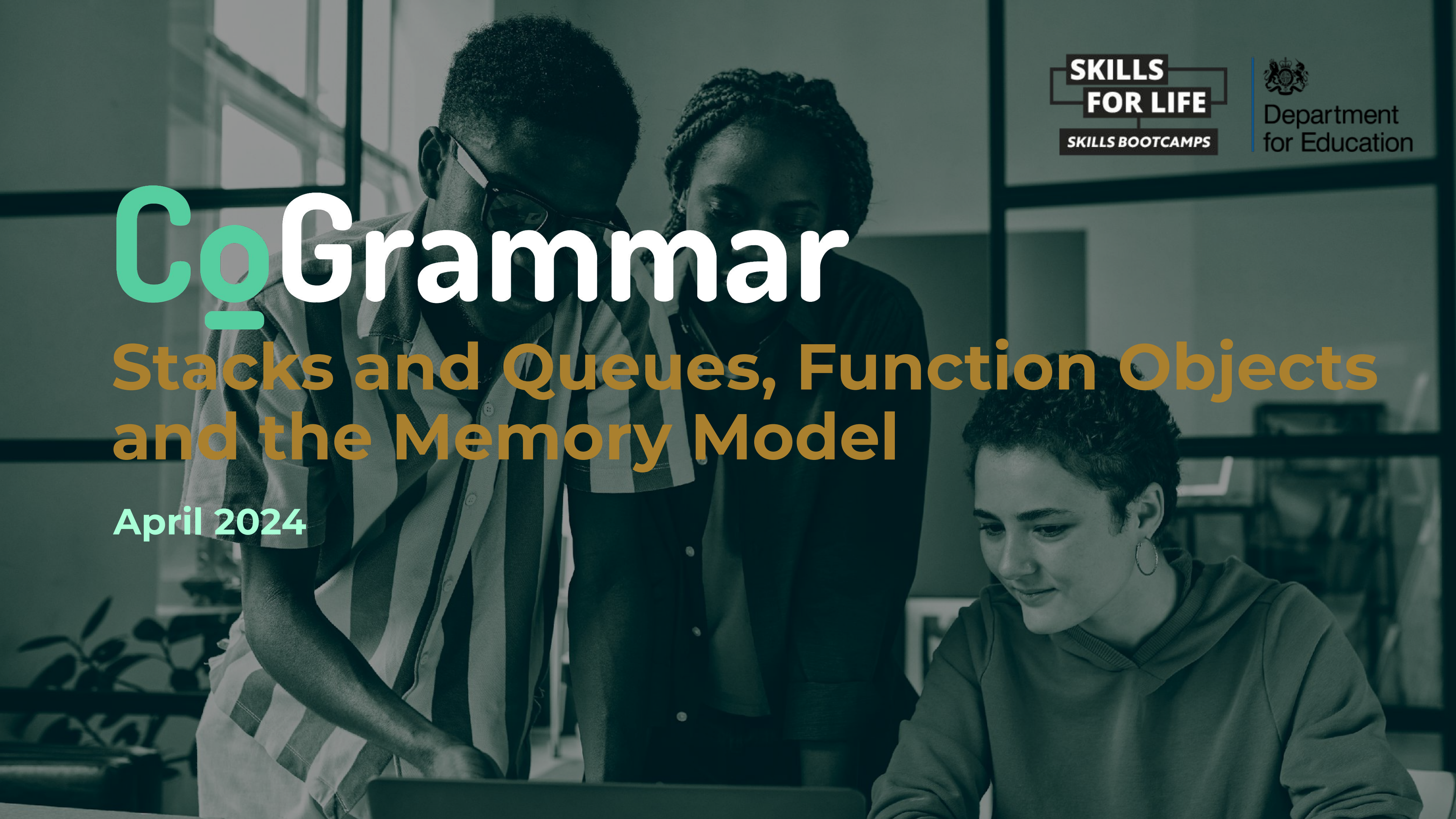
✅ Criterion 3: Course Progress

Completion: All mandatory tasks, including Build Your Brand and resubmissions by study period end
Interview Invitation: Within 4 weeks post-course
Guided Learning Hours: Minimum of 112 hours by support end date (10.5 hours average, each week)

✅ Criterion 4: Demonstrating Employability

Final Job or Apprenticeship Outcome: Document within 12 weeks post-graduation
Relevance: Progression to employment or related opportunity

CoGrammar



SKILLS FOR LIFE
SKILLS BOOTCAMPS

Department for Education

CoGrammar

Stacks and Queues, Function Objects and the Memory Model

April 2024

Learning objectives

- ❖ Describe and implement **stack** and **queue data structures** in Python and JavaScript, understanding their use cases.

- ❖ Demonstrate the use of **function objects** in Python and JavaScript, including **handling scope** and **closures**.

CoGrammar

Learning objectives

- Explain the distinction between the **stack** and **heap memory models** and their implications for **memory management** in Python and JavaScript.

- Differentiate between **pass by reference** and **pass by value** in both languages, illustrating with examples how each affects function arguments and behavior.

CoGrammar

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

1. O(1) for both push and pop
2. O(n) for push and O(1) for pop
3. O(1) for push and O(n) for pop
4. O(n) for both push and pop

CoGrammar

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

1. **O(1) for both push and pop**
2. O(n) for push and O(1) for pop
3. O(1) for push and O(n) for pop
4. O(n) for both push and pop

CoGrammar

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

1. Array
2. Linked List
3. Hash Table
4. Both Arrays and Linked Lists

CoGrammar

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

1. Array
2. Linked List
3. Hash Table
4. **Both Arrays and Linked Lists**

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

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

2. Elements are always sorted in ascending order.

3. Elements can only be numeric values.

4. Priority queues have faster access times than standard queues.
5.

CoGrammar

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

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

2. Elements are always sorted in ascending order.

3. Elements can only be numeric values.

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

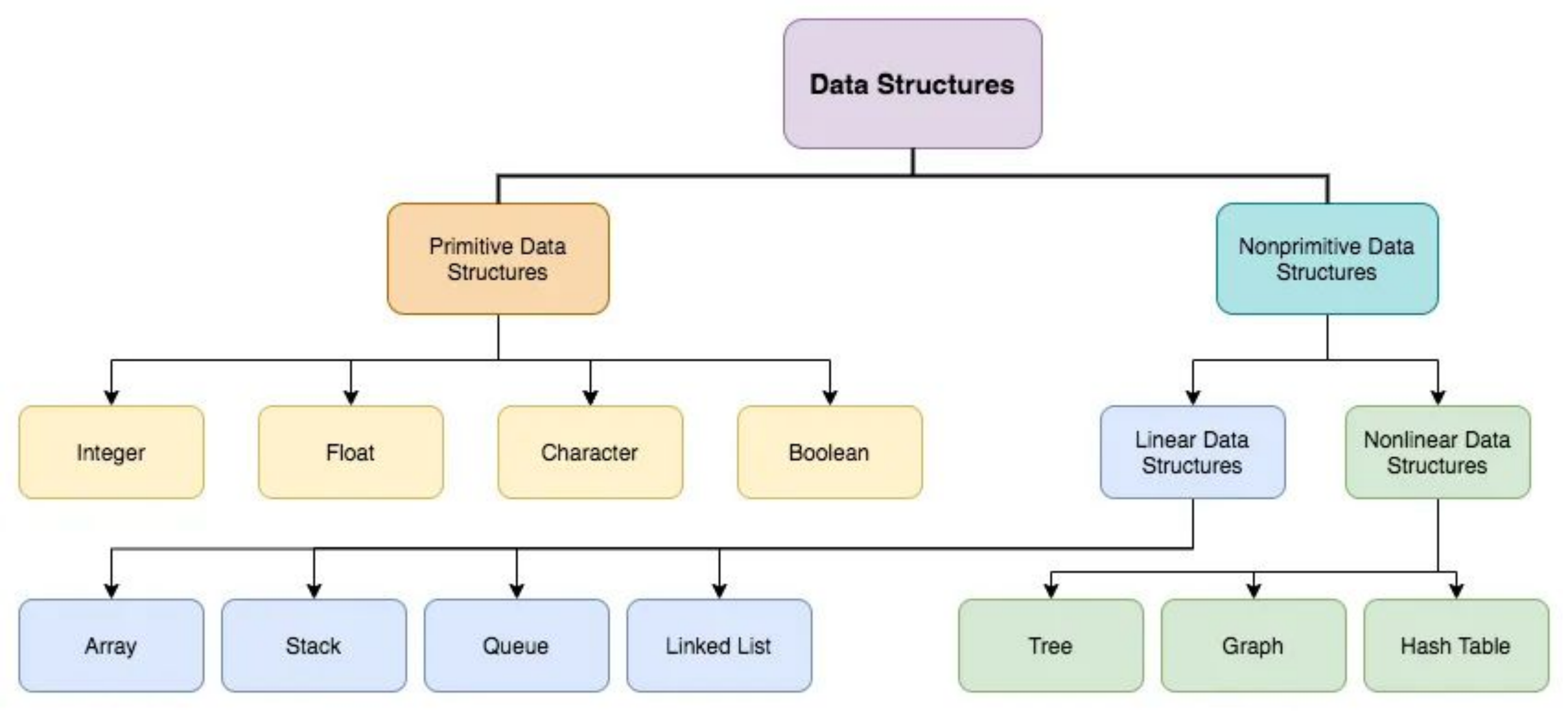
5.

CoGrammar

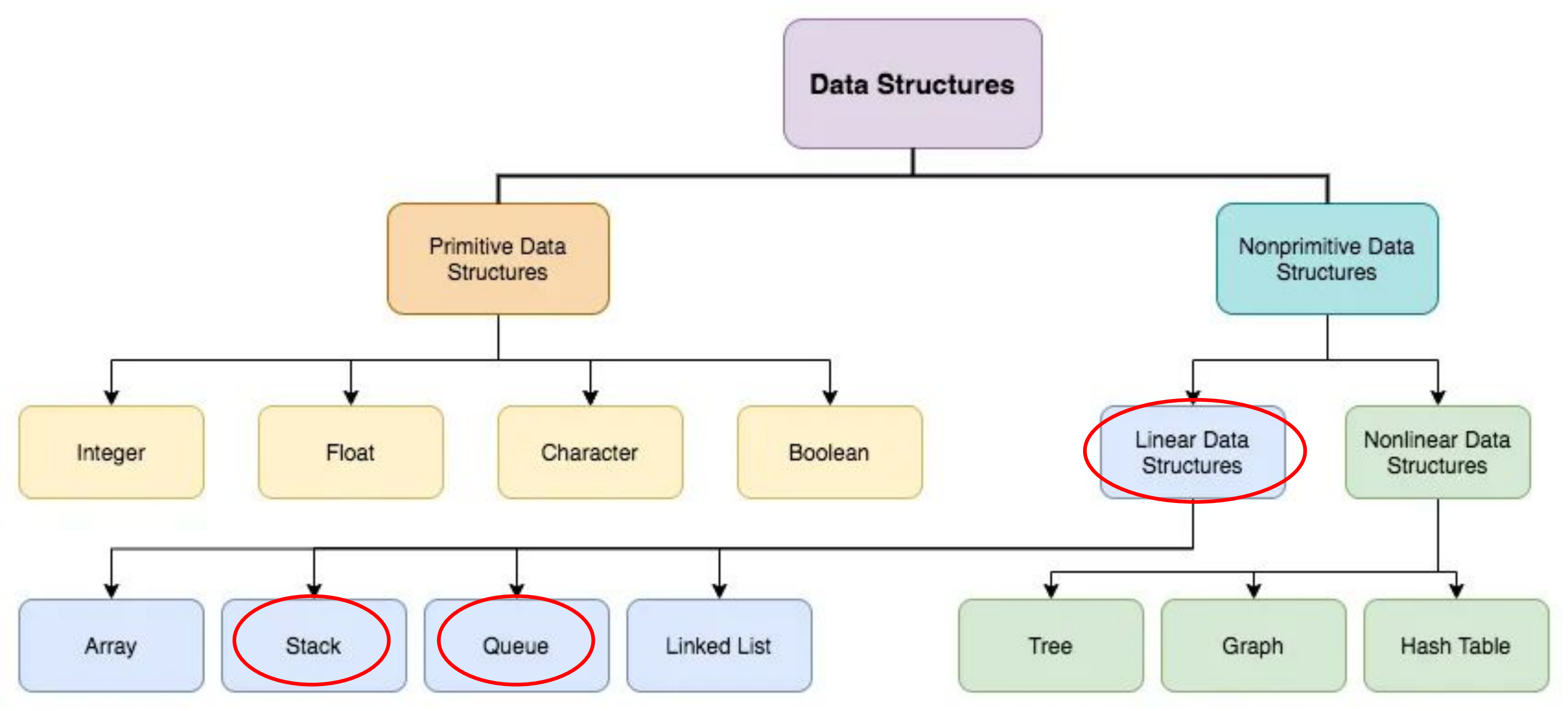
Linear Data Structures

CoGrammar

Data Structures



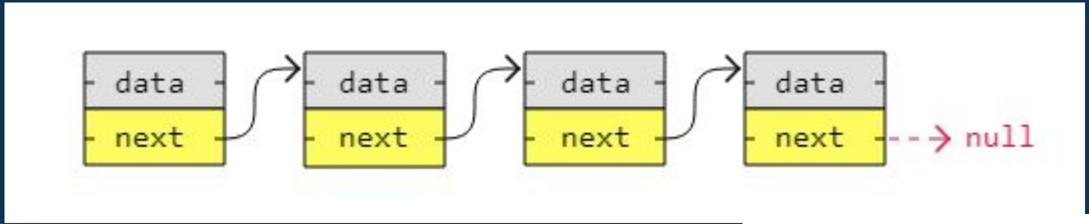
Data Structures



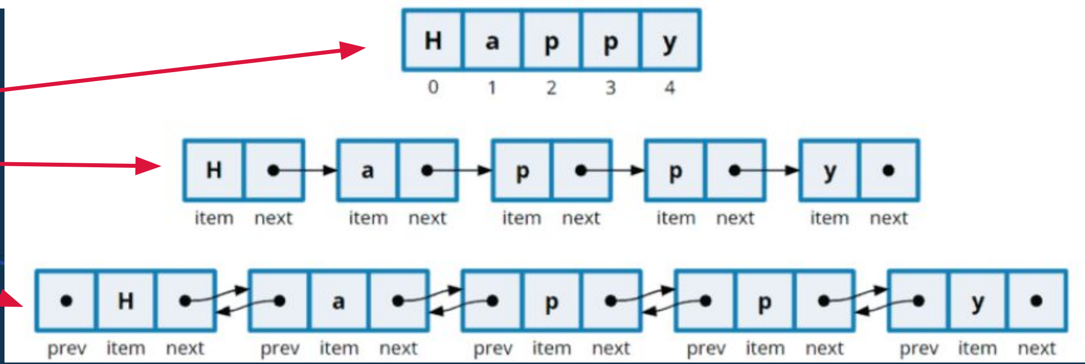
Peek into Linked List

A **linked list** is a **linear data structure** consisting of a **sequence of nodes**, where each node contains **data** and a **reference (link)** to the next node in the sequence. Ideal for **dynamic data storage** where the size of the data set is not fixed.

More details: Week 8 session



Array
Singly linked list
Doubly linked list



CoGrammar

Stacks and Queues

Introduction and Implementation

CoGrammar

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

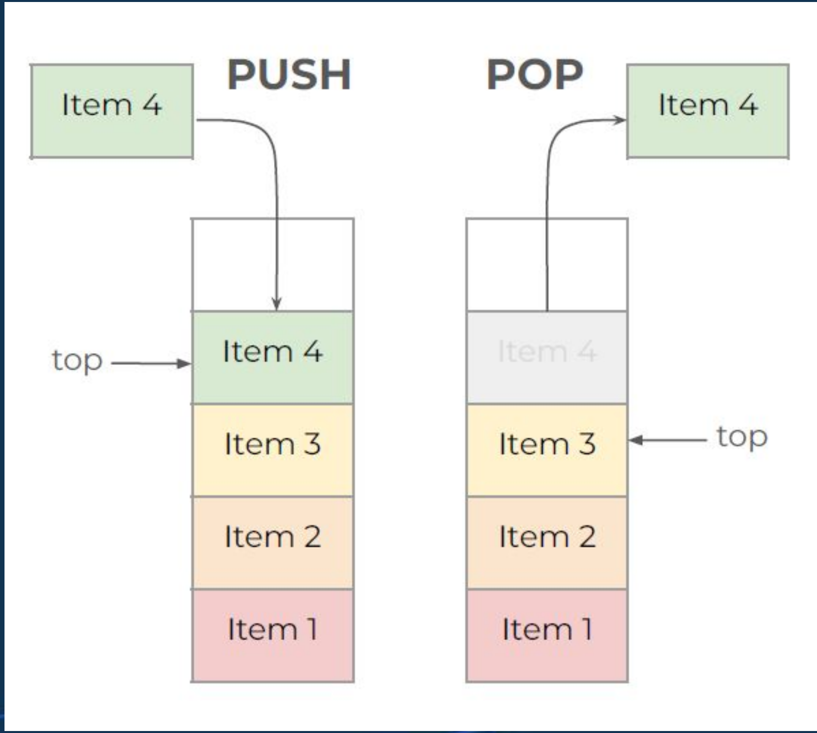
Stacks

❖ Method of ordering

- ➢ **Last In First Out (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**



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
```

```javascript
// 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)].

CoGrammar

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
```

```javascript
// 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 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

```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
```

```javascript
// 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 it's slower to pop or insert an element at any other position besides the end of the list [O(n)].

CoGrammar

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

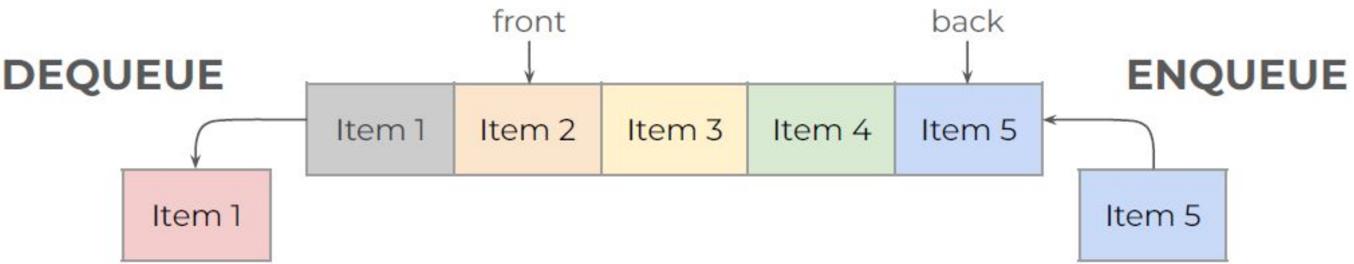
Common Uses

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

CoGrammar

Queues

- Method of ordering
  - **First In First Out (FIFO):** First element added to the stack is the first to be removed
  - Elements are **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



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()
```

```javascript
// 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: dequeue 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("Error: Queue Underflow!")
        return None
    else:
        return self.queue.popleft()
```

```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();
    }
  }
```

CoGrammar

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
- Printing Queues
- Web Servers
- Breadth-First Search

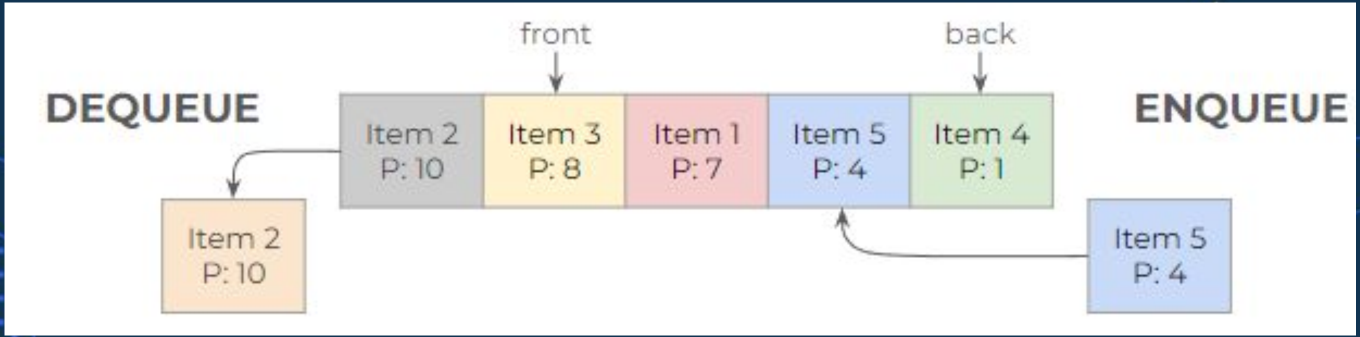
Priority Queues

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

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.

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

```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.

Priority Queues: `queue` Implementation

```python
# 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: 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

```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

Complexity Analysis

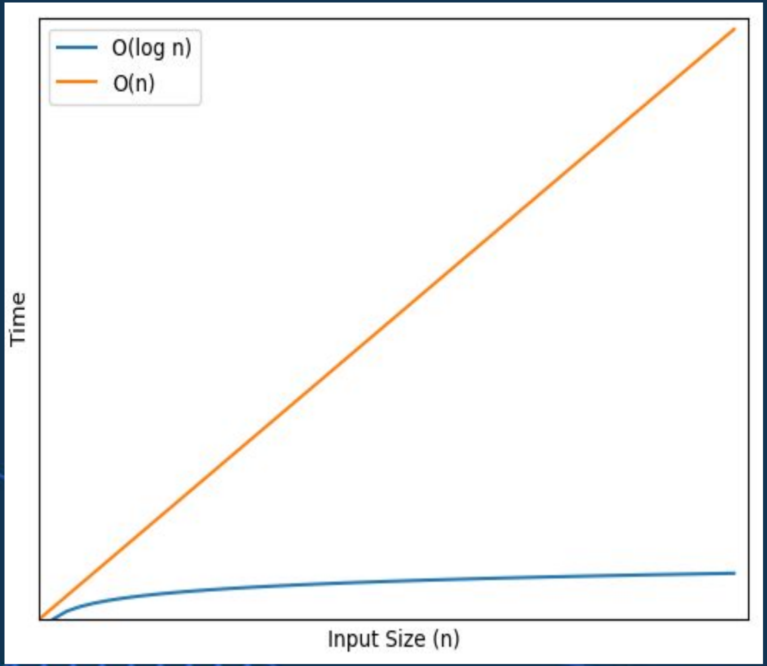
- **Enqueue**
  - **Space:**
    - **List - O(1)**
      No extra space is used
    - **PriorityQueue - O(1)**
      No extra space is used
  - **Time:**
    - **List - O(n)**
      Each element's priority must be checked and compared
    - **PriorityQueue - O(log n)**
      Adding node to heap

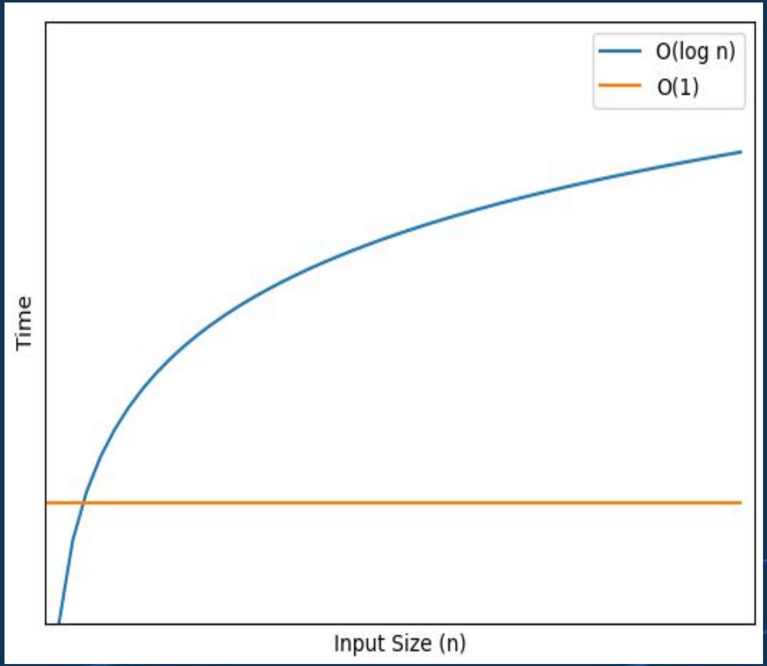
- **Dequeue**
  - **Space:**
    - **List - O(1)**
      No extra space is used
    - **PriorityQueue - O(1)**
      No extra space is used
  - **Time:**
    - **List - O(1)**
      Pointer is updated or last element is removed
    - **PriorityQueue - O(log n)**
      Removing node from heap

CoGrammar

Priority Queues

Complexity Analysis Visualisation



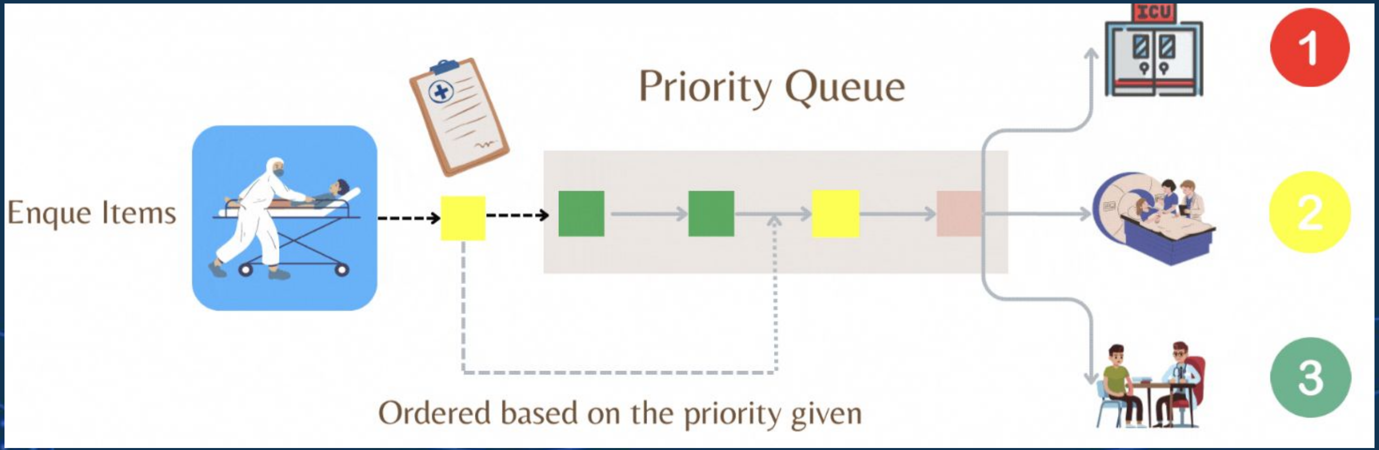


CoGrammar

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)

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

A. Stack

B. Queue

C. Priority Queue

D. Linked List

CoGrammar

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

A. **Stack**

B. Queue

C. Priority Queue

D. Linked List

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

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

Function Objects

CoGrammar

What will be the output of the following Python code?

```
x = 50
def func():
    global x
    print('x is', x)
    x = 2
    print('Changed global x to', x)
func()
print('Value of x is', x)
```

A. x is 50, Changed global x to 2, Value of x is 50

B. x is 50, Changed global x to 2, Value of x is 2

C. x is 50, Changed global x to 50, Value of x is 50

D. None of the above

CoGrammar

What will be the output of the following Python code?

```python
x = 50
def func():
    global x
    print('x is', x)
    x = 2
    print('Changed global x to', x)
func()
print('Value of x is', x)
```

A. x is 50, Changed global x to 2, Value of x is 50

B. x is 50, Changed global x to 2, Value of x is 2

C. x is 50, Changed global x to 50, Value of x is 50

D. None of the above

CoGrammar

Function Objects

- A programming language is said to support **first-class functions** when **functions** in that language are **treated like any other variable.**

- **Python** and **JavaScript** support the concept of first-class functions.

- Properties of **first-class functions**
  - A function is an **instance** of the **object** type.
  - You can **store** the **function** in a **variable**.
  - You can **pass** the **function** as a **parameter to another function**.
  - You can **return** the **function from a function**.
  - You can **store** them in **data structures** such as hash tables, lists, …

Function objects

Passing function to a variable

```python
#Python program: functions can be
#treated as objects
#Passing function to a variable
def greet(text):
    return text.upper()

print(greet('Hello'))
intro = greet
print (intro('Hello World!'))
#Output: HELLO
#HELLO WORLD!
```

```javascript
// Passing function to a variable
function greet(text){
    return text.toUpperCase();
}

console.log(greet('Hello'));
let intro = greet;
console.log(intro('Hello World!'));
//Output: HELLO
//HELLO WORLD!
```

Function objects

Passing function as arguments to other functions

```python
# Pasing functions as arguments to other functions
def shout(text):
    return text.upper()

def whisper(text):
    return text.lower()

def greet(func, name):
    # storing the function in a variable
    greeting = func('Function passed as argument in ' + name)
    print(greeting)

greet(shout, "Python")
greet(whisper, "Python")
#Output: FUNCTION PASSED AS ARGUMENT IN PYTHON
#function passed as argument in python
```

```javascript
// Passing functions as arguments to other functions
function shout(text) {
    return text.toUpperCase();
}

function whisper(text) {
    return text.toLowerCase();
}

function greeting(helloMessage, name) {
    console.log(helloMessage("Function passed as argument in ") + name);
}

greeting(shout, "JavaScript");
greeting(whisper, "JavaScript");
//Output: FUNCTION PASSED AS ARGUMENT IN JavaScript
// function passed as argument in JavaScript
```

Function objects

Function returning another function

```python
#Functions returning another function
def add_sub(a, b):
    add = a + b
    sub = a - b
    def mult(c):
        print(add * sub * c)
    return mult

# form a object of first method
returned_function = add_sub(5, 2)
#a=5 & b=2 -> add=7 & sub=3

# check object
print(returned_function)
#Output: It will be a <function>

# call second method by first method
returned_function(10)
#Output: 210 (With c = 10, a*b*c = 210)
```

```javascript
// Functions returning another function
function add_sub(a, b) {
    let add = a + b;
    let sub = a - b;
    function mult(c) {
        console.log(add * sub * c);
    }
    return mult;
}

// form a object of first method
let returned_function = add_sub(5, 2);
// a=5 & b=2 -> add=7 & sub=3

// check object
console.log(returned_function);
// Output: It will be a function

// call second method by first method
returned_function(10);
// Output: 210 (With c = 10, a*b*c = 210)
```

CoGrammar

Function Objects

Scope

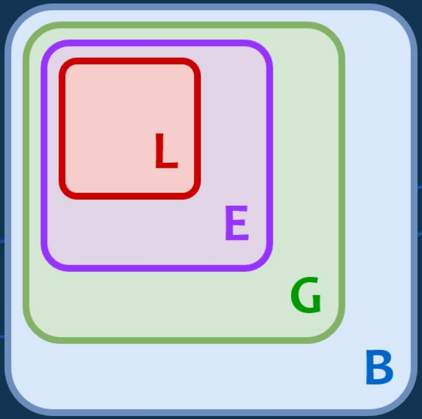
CoGrammar

Scope

The **scope** of a variable refers to the context in which that variable is visible/accessible to the Python interpreter.

A **namespace** is a mapping from names to objects.

- **Local**: If x is referred inside a function, then the interpreter first searches for it in the innermost scope that's local to that function.
- **Enclosing or Nonlocal**: If x is not in the local scope but appears in a nested function, interpreter searches in the enclosing function scope.
- **Global**: If neither of the above searches is fruitful, then the interpreter looks in the global scope next.
- **Built-in:** If it can't find x anywhere else, then the interpreter tries the built-in scope.



CoGrammar

Scope

- **Local scope:** A variable created inside a function belongs to the local scope of that function, and can only be used inside that function.

```python
#Local Scope
def myfunc():
  x = "Hello!"
  print(x)

myfunc()
#Output: Hello!
```

```javascript
// Local Scope
function myfunc() {
    let x = "Hello!";
    console.log(x);
  }

  myfunc();
  // Output: Hello!
```

- **Nested Function**: variable not available outside the function, but it is available for any nested

```python
#Nested function
def myfunc():
  x = "Hello"
  def myinnerfunc():
    print(x+" World!")
  myinnerfunc()

myfunc()
#Output: Hello World!
```

```javascript
// Nested function
function myfunc() {
    let x = "Hello";
    function myinnerfunc() {
      console.log(x + " World!");
    }
    myinnerfunc();
  }

  myfunc();
  // Output: Hello World!
```

CoGrammar

Scope rules

The **global** declaration allows a function to access and modify an object x in the global scope.

To modify x in the enclosing scope, keyword **nonlocal** is used.

Global and **local** widespread use is unwise.

```python
#Scope rules
def scope_test():
    def do_local():
        text = "local text"

    def do_nonlocal():
        nonlocal text
        text = "nonlocal text"

    def do_global():
        global text
        text = "global text"

    text = "sample text"
    do_local()
    print("After local assignment:", text)
    do_nonlocal()
    print("After nonlocal assignment:", text)
    do_global()
    print("After global assignment:", text)

scope_test()
print("In global scope:", text)
#Output: After local assignment: sample text
#After nonlocal assignment: nonlocal text
#After global assignment: nonlocal text
#In global scope: global text
```

```javascript
// Scope rules
function scopeTest() {
    function doLocal() {
        let text = "local text";
    }

    function doNonlocal() {
        text = "nonlocal text";
    }

    function doGlobal() {
        global.text = "global text";
    }

    let text = "sample text";
    doLocal();
    console.log("After local assignment:", text);
    doNonlocal();
    console.log("After nonlocal assignment:", text);
    doGlobal();
    console.log("After global assignment:", text);
}

scopeTest();
console.log("In global scope:", global.text);
// Output:
// After local assignment: sample text
// After nonlocal assignment: nonlocal text
// After global assignment: nonlocal text
// In global scope: global text
```

CoGrammar

Function Objects

Closure

CoGrammar

Closure

- **Outer Function (Enclosing Function):** The function contains another function (inner or nested function). It can take arguments and define variables that the inner function can access.

- **Inner Function (Nested Function):** The function defined within the outer function. It can access variables from the outer function even after the outer function has completed execution.

- **Variable Capture:** When an inner function references a variable from its enclosing (outer function) scope, Python "captures" or retains that variable's value, allowing it to be used later, even when the outer function has returned.

- **Closure** is a nested function that allows us to access variables of the outer function even after the outer function is closed.

Closure

- **Closures** can be used to avoid global values and provide data hiding, and can be an elegant solution for simple cases with one or few methods.

- For larger cases with multiple attributes and methods, a class implementation may be more appropriate.

```python
#Closure
def make_multiplier_of(n):
    def multiplier(x):
        return x * n
    return multiplier


# Multiplier of 2
times2 = make_multiplier_of(2)

# Multiplier of 5
times5 = make_multiplier_of(5)

#times2 and times5 are closure functions

print(times2(7))
#Output: 14

print(times5(9))
#Output: 45

print(times5(times2(3)))
# Output: 30
```

```javascript
// Closure
function make_multiplier_of(n) {
    return function multiplier(x) {
      return x * n;
    };
  }

  // Multiplier of 2
  const times2 = make_multiplier_of(2);

  // Multiplier of 5
  const times5 = make_multiplier_of(5);

  // times2 and times5 are closure functions

  console.log(times2(7));
  // Output: 14

  console.log(times5(9));
  // Output: 45

  console.log(times5(times2(3)));
  // Output: 30
```

Memory Model

Stack and Heap

CoGrammar

Which of these is typically stored in the stack?

A. Objects instantiated from classes
B. Global variables
C. Local variables within a function
D. Data loaded from a database

Which of these is typically stored in the stack?

A. Objects instantiated from classes

B. Global variables

C. Local variables within a function

D. Data loaded from a database

What type of memory allocation is used for objects in Python?

A. Stack Allocation

B. Heap Allocation

C. Static Allocation

D. Register Allocation

CoGrammar

What type of memory allocation is used for objects in Python?

A. Stack Allocation

B. Heap Allocation

C. Static Allocation

D. Register Allocation

CoGrammar

In Python, what can lead to a Stack Overflow error?

A. Accessing an undefined variable

B. Excessive use of global variables

C. Deep recursion without a base case

D. Allocating large arrays in the heap

In Python, what can lead to a Stack Overflow error?

A. Accessing an undefined variable
B. Excessive use of global variables
C. Deep recursion without a base case
D. Allocating large arrays in the heap

CoGrammar

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, 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.**

Example: Python Variables and Memory Allocation

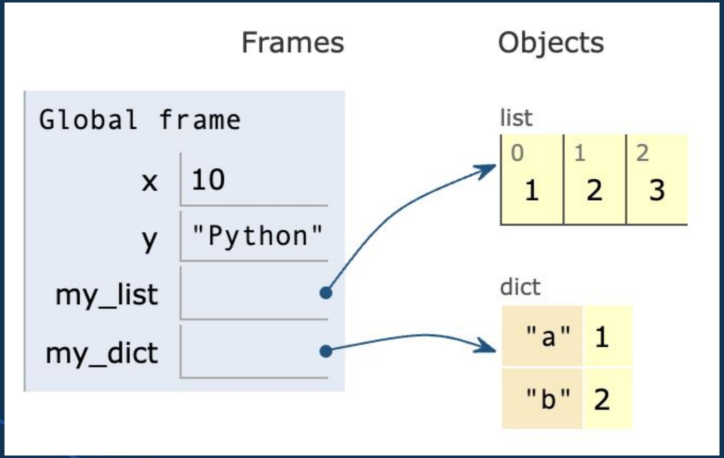
```
# Immutable data allocated on the stack
x = 10
y = "Python"

# 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.

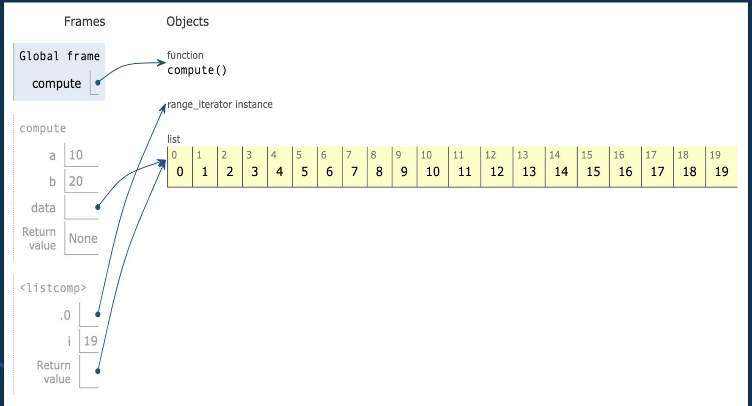
CoGrammar

Memory Allocation: The Trade-Off Between Speed and Flexibility

- 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.

Memory Allocation: The Trade-Off Between Speed and Flexibility




```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.

The Stack

The stack is a special area of a computer's memory which stores temporary variables created by a function.

- Python operates within a **finite memory space** and must allocate memory efficiently between the stack and heap.
- The stack is generally reserved for **smaller, short-lived data, while the heap is used for larger, longer-lived objects**.
- Understanding how Python manages memory helps developers **write more efficient code**.

CoGrammar

The Stack

Once again we see how faded the stack becomes once it is called, showcasing that the **function's execution is complete**, and its local variables, including message, are **out of scope and their memory is cleared from the stack.**

```python
def greet(name):
    # 'message' is a local variable, stored in the stack
    message = "Hello, " + name
    print(message)
greet('Alice')
```

CoGrammar

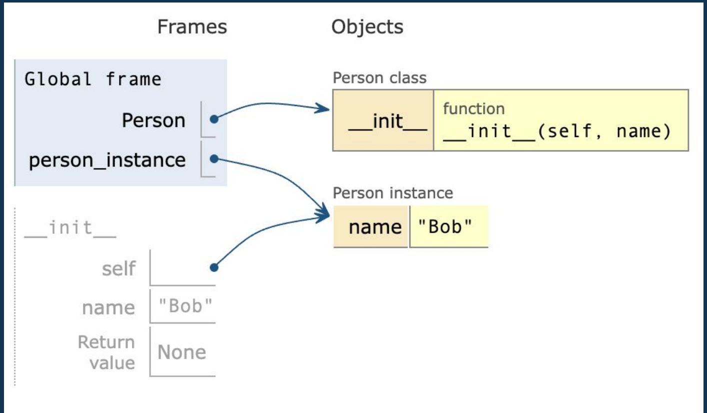
The Heap

The heap is a region of memory used by programming languages to store global variables and supports dynamic memory allocation.

- Unlike the stack, the heap is **not managed automatically by the CPU but by the Python memory manager.** When objects are created, they are placed in the heap and **remain until they are no longer needed**, at which point **the garbage collector reclaims the memory**.

CoGrammar

The Heap



The Person class instance **person_instance with its attribute name set to "Bob" is stored in the heap**, while the reference to this object is held in the stack within the global frame.

```python
class Person:
    def __init__(self, name):
        # 'name' attribute is stored in the heap as part of the object
        self.name = name
person_instance = Person('Bob')
```

CoGrammar

Pass by Reference and Pass by Value

CoGrammar

Pass by Reference and Value

- Depending on the **type of object** we **pass in the function**, the function behaves differently.

- **Pass by reference:** Some languages **handle function arguments as references to existing variables, mutable objects**
  - **Pass:** provide an argument to a function.
  - **By reference**: the **argument passing to the function** is a **reference to a variable** that already **exists** in memory rather than an independent copy of that variable. All operations performed on this reference will directly affect the variable to which it refers.

- **Pass by value:** handle them as **independent values, immutable objects.**

CoGrammar

Examples

Pass by value: The value 10 was copied to **b** when it was declared, and changes to **a** do not affect **b**. They are independent of each other, each occupying its own memory space.

Pass by reference: **obj1** and **obj2** point to the same memory space, both references to the same object.

```python
#Pass by value
a = 10
b = a

a = 20

print(a) #Outputs: 20
print(b) #Outputs: 10
```

```javascript
// Pass by value
let a = 10;
let b = a;

a = 20;

console.log(a); // Outputs: 20
console.log(b); // Outputs: 10
```

```python
# Pass by reference
obj1 = {'value': 10}
obj2 = obj1

obj1['value'] = 20

print(obj1['value'])  #Outputs: 20
print(obj2['value'])  #Outputs: 20
```

```javascript
// Pass by reference
let obj1 = { value: 10 };
let obj2 = obj1;

obj1.value = 20;

console.log(obj1.value); // Outputs: 20
console.log(obj2.value); // Outputs: 20
```

CoGrammar

Value Types and Reference Types

- **Value types** in Python, such as **numbers** and **booleans**, are **stored directly in the variable;** they are **copied** when the variable is assigned to another variable or passed to a function.

- **Reference types**, such as **lists** and **class instances**, **store a reference to the object's memory address.** When assigning or passing these, **only the reference is copied**, not the object itself.

```python
# Value type: Copied when passed to a function
def increment(number):
    number += 1
    return number

# Reference type: The reference is passed, original list can be modified
def append_to_list(lst):
    lst.append(4)

lst = [1, 2, 3]
append_to_list(lst)
number = 10
increment(number)
```

CoGrammar

Recursion and Stack Overflow

- **Recursion** is a common technique in Python where a function calls itself. If it goes too deep (each call adds a new frame to the stack), causes **stack overflow**. This is because there's a limit to the size of the stack, which, when exceeded, **causes the program to crash**.

- **Stack overflow** can be **mitigated** by
  - Increasing the maximum recursion depth
  - More robust: Convert recursive algorithms into their iterative counterparts, thereby **using the heap instead of the stack** and avoiding the risk of overflow.

More details: Week 7 session

CoGrammar

In Python, where are the references to heap-allocated objects stored?

A. In the Heap

B. In the Stack

C. In the CPU Registers

D. On the Disk

In Python, where are the references to heap-allocated objects stored?

A. In the Heap

B. In the Stack

C. In the CPU Registers

D. On the Disk

CoGrammar

What is a primary characteristic of memory allocation in the heap compared to the stack?

A. Faster access times

B. Memory is automatically managed

C. Slower allocation and deallocation of memory

D. Only supports primitive data types

CoGrammar

What is a primary characteristic of memory allocation in the heap compared to the stack?

A. Faster access times

B. Memory is automatically managed

C. Slower allocation and deallocation of memory

D. Only supports primitive data types

CoGrammar

Portfolio Assignment Reviews

[Submit you solutions here!]

Provide a README file, how to run it, and examples of output.

CoGrammar

Portfolio Assignment: SE

Project: Scheduler to manage task execution

Objective: 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.

CoGrammar

Provide a README file, how to run it, and examples of output.

Portfolio Assignment: SE

Requirements:

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

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

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

CoGrammar

Provide a README file, how to run it, and examples of output.

Portfolio Assignment: DS

Project: Cafeteria queues and stacks

Objective: The school cafeteria serves brown (0) and white (1) bread toasted sandwiches to students. Students queue to get their preferred bread from the sandwiches kept in a stack. If the student at the front of the queue prefers the type of sandwich that is on top of the sandwich stack, they take the sandwich and leave the queue. Otherwise, they pass on the sandwich and move to the end of the queue. This continues until none of the queue students want to take the top sandwich and are thus unable to eat. Return the number of students that are unable to eat.

CoGrammar

Provide a README file, how to run it, and examples of output.

Portfolio Assignment: WD

Project: Matching HTML tags

Objective: Uses stacks to match HTML tag in an HTML document. A simple opening HTML tag has the form “<name>” and the corresponding closing tag has the form “</name>”. Write a program to use stack to match the HTML tag and validate the input file.

Provide a README file, how to run it, and examples of output.

CoGrammar

Further Learning

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

- GeeksforGeeks - Stacks Complexity Analysis

- GeeksforGeeks - Queues Complexity Analysis

- Real Python - Memory Management in Python

- Scout APM Blog - Python Memory Management: The Essential Guide

CoGrammar

Q & A SECTION

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

Thank you for attending

SKILLS FOR LIFE
SKILLS BOOTCAMPS

Department for Education

CoGrammar