Welcome to the CoGrammar Recursion and Object-Orientated Programming Lecture

The session will start shortly...

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

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

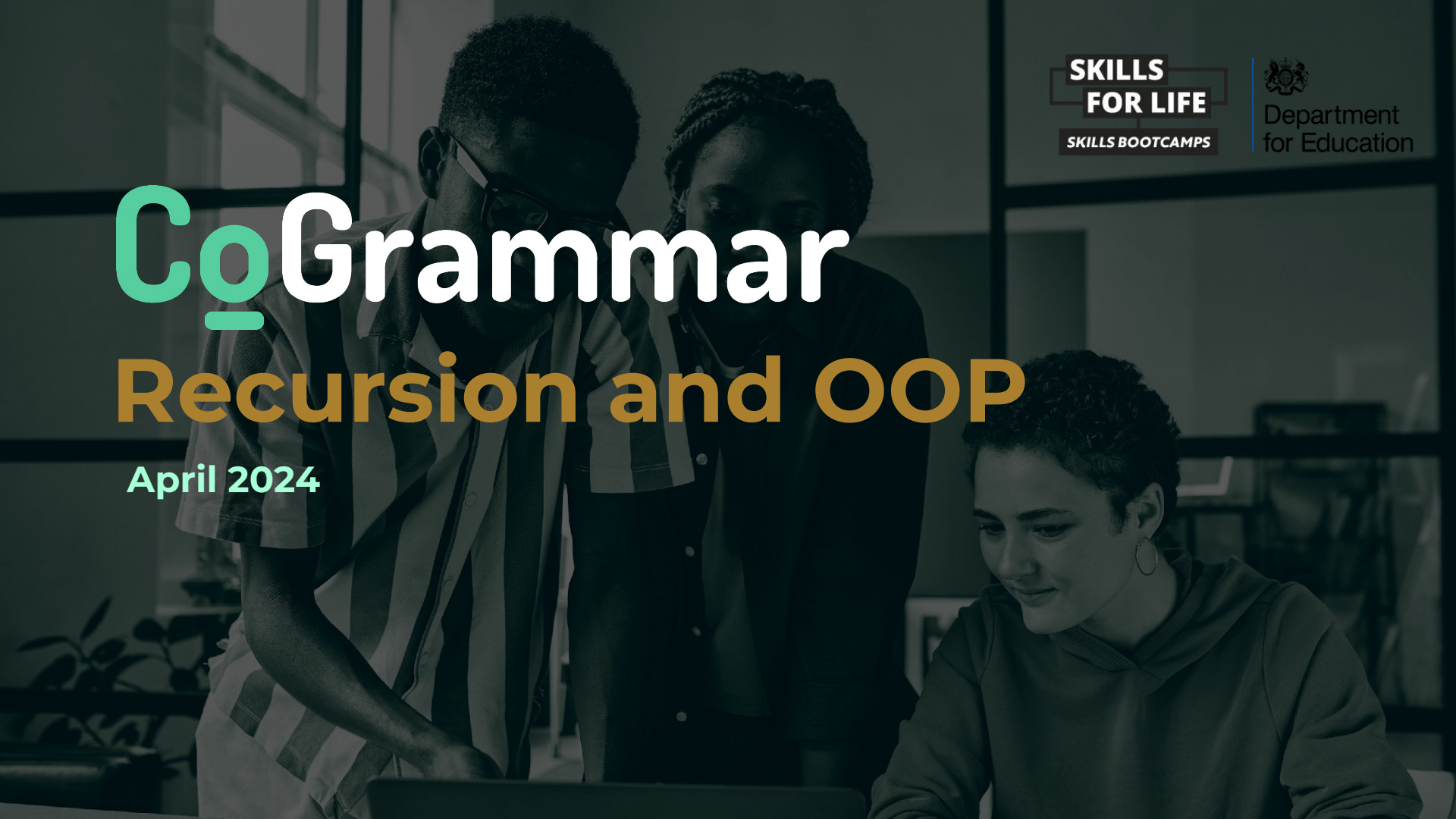
Skills Bootcamp Progression Overview

✅ 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


SKILLS
FOR LIFE
SKILLS BOOTCAMPS
Department
for Education
CoGrammar
Recursion and OOP
April 2024

Learning Objectives

- Understand and apply the concept of recursion in Python and JavaScript, including base cases and recursive steps for problem-solving.
- Identify and resolve common issues associated with recursion, such as stack overflows, by optimizing recursive functions or using iterative solutions.
- Explain the principles of object-oriented programming (OOP), including encapsulation, inheritance, and polymorphism, and how they facilitate code reuse and modularity.

Learning Objectives

- Implement classes and objects in Python and JavaScript, demonstrating the use of constructors, methods, and attributes.
- Apply runtime polymorphism in OOP through method overriding and interface implementation to enable dynamic method dispatch.

Recursion

A method of solving a computational problem where the solution depends on solutions to smaller instances of the same problem

- Consider the following problem:

> Imagine you are back in school and you are helping a new student figure out the ropes of the classroom. A note passes by and you decide to explain the note passing protocol to the new student.

Recursion

- We could explain the process to the student as follows:

1. Take the note from the student behind you.
2. Check the name on the note:
    a. If the name on the note is your name, open the note.
    b. If the name on the note is not your name:
        i. Pass the note on to the person in front of you.
        ii. Repeat steps 1 and 2.

Recursion

- We can convert these instructions to code using an iterative approach:

```python
students = ["Zahra", "Moumita", "Anri", "Julien"]

def passNote (destination):
    found = False
    i = 0

    while (not found):
        if (i == len(students)):
            return "Destination not found"
        elif (destination == students[i]):
            found = True
            return "Note delivered"

        i += 1
```

```javascript
let students = ["Zahra", "Moumita", "Anri", "Julien"];

function passNote (destination){
    let found = False;
    i = 0;

    while (!found){
        if (i == len(students))
            return "Destination not found";
        else if (destination == students[i]){
            found = True;
            return "Note delivered";
        }


        i += 1;
    }
}
```

CoGrammar

Recursion

- Alternatively, we could use recursion, which involves calling a function within itself:

```python
students = ["Zahra", "Moumita", "Anri", "Julien"]

def passNote (destination, location):
    if (location == len(students)):
        return "Destination not found"
    elif (destination == students[location]):
        return "Note delivered!"
    else:
        return passNote(destination, location + 1)
```

```javascript
let students = ["Zahra", "Moumita", "Anri", "Julien"];

function passNote (destination, location) {
    if (location == len(students))
        return "Destination not found";
    else if (destination == students[location])
        return "Note delivered!";
    else
        return passNote(destination, location + 1);
}
```

Recursion

- A recursive function has these two components:
  - **Base case**: A condition that stops the recursive calls.
  - **Recursive step**: The step where the function calls itself with a smaller input.

Stack Overflow

Occurs when the number of function calls added to the stack is more than the stack's maximum limit.

- In our previous lecture, we looked at **the Stack and the Heap.**
- We saw that **function calls**, are added to the **stack** and are kept their until the function has **returned or completed execution**.
- Recursive functions that are **too deep** or that **have no base case** can result in stack overflow.

```
function overflow () {
    return overflow (); }
```

CoGrammar

Stack Overflow

- In cases where we **do have a defined base case**, the following methods can be used to prevent stack overflow:
  - **Limit the depth of recursion**: keep track of the number of recursive calls and stop the function once a maximum is reached.
  - **Tail recursion**: in our recursive function, we ensure that the recursive call is the last statement executed.
    - This optimization **does not** work for Python and for many JavaScript compilers, since it does not help the call stack.
  - **Convert to an iterative solution**: all recursive solutions can be converted into iterative solutions, which may be more complex

Stack Overflow

- Consider the following code, how could we prevent possible cases of stack overflow:

```python
def sum (n):
    if (n <= 0):
        return 0
    else:
        return n + sum (n-1)
```

```javascript
function sum (n){
    if (n <= 0)
        return 0;
    else
        return n + sum (n-1);
}
```

Let’s Breathe!

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

CoGrammar

Object-Oriented Programming

A programming paradigm based on the concept of objects which store data in the form of attributes and code in the form of methods.

- Consider a scenario where you may want to store the information of several students in a class.
  - Each student has multiple sets of data pertaining to them.
  - There are some functions that we may need to perform for each students which involves the data pertaining to them.
- We could implement this using multiple arrays/lists, dictionaries or maps to store all the data but this could become confusing

Object-Oriented

- What if we could define a new data type: **“Student”**
- We can do this using **objects** in JavaScript and Python.
- In order to create objects, we create a “template” or “blueprint” for the object using **classes**.
- In this blueprint, we outline the different **attributes** that the object has and the different **methods** defined for the object.
- In JavaScript, objects can be created using **object literal notation** and **class notation**. For simplicity, we will only be using class notation.

Object-Oriented

- We use the **class** keyword to create a new class, followed by the name of the class.
- We use a **constructor function** to define anything that needs to take place when the object is first instantiated.
  - This includes any **attributes** that need to be defined, which we store using the **this (JS)** keyword or **self (Python)** parameter.
  - In Python, **self** has to be passed into every function in the class as the first parameter but does not have to be included when the function is actually called.

Object-Oriented Programming

```python
class Student:
    def __init__(self, name, age, grade):
        self.name = name
        self.age = age
        self.grade = grade
```

```javascript
class Student {
    constructor (name, age, grade) {
        this.name = name;
        this.age = age;
        this.grade = grade;
    }
}
```

Object-Oriented

- We define methods in our classes the same way that we would define functions.
- To reference any of the class' attributes we use **this** or **self**.

```python
def sayMyName (self):
    print("Hi, my name is " + self.name)
```

```javascript
sayMyName () {
    console.log(`Hi, my name is ${this.name}`);
}
```

Object-Oriented

- To create a new object of a certain class, we use the name of the object and pass in any parameters needed by constructor.
- We access the attributes and methods of the object using a “.”

```
zahra = Student("Zahra", 23, 12)
print(zahra.name)
```

```
let zahra = new Student("Zahra", 23, 12);
console.log(zahra.name);
```

Encapsulation

A fundamental concept in OOP which involves hiding the internal details of an object and controlling how data within the object can be manipulated.

- Instead of allowing for attributes to be accessed directly, we make our attributes **private** and create **getter** and **setter methods** which can be used to modify and access the attributes.
- We use a "_" in front of the name of an attribute to change the **visibility** of the attribute.

Encapsulation

```
class User:
    def __init__(self, username, password, accessCode):
        self.username = username
        self._password = password
        self._accessCode = accessCode

    def getAccess(self, username, password):
        if (self.username == username):
            if (self._password == password):
                return self._accessCode
            else:
                return "Incorrect Password"
        else:
            return "Incorrect Username"
```

Inheritance

A mechanism where a new, child class inherits attributes and methods from an existing, parent class.

- Inheritance allows for classes to be created based on an existing class, which **shares** attributes and methods.
- A child class can have its own methods and attributes as well.
- In JavaScript, this is implemented using **prototypes**, all objects are said to have a **prototype** and attribute and method calls are passed through the **prototype chain**, until it is found.
- In Python, the parent class is passed as a **parameter** to the child class.
- We use the **super** keyword to access the parent class within the child class.

Inheritance

```python
class Animal:
    def __init__(self, name):
        self.name = name

    def sayHi (self):
        print("Hi, I am a " + self.name)

class Mammal (Animal):
    def __init__(self, name, gestationPeriod):
        super().__init__(name)
        self.gestation = gestationPeriod

mammal = Mammal("Zebra", 12)
mammal.sayHi()
```

```javascript
class Animal {
    constructor (name) {
        this.name = name;
    }

    sayHi () {
        console.log(`Hi, I am a ${this.name}`);
    }
}
class Mammal extends Animal {
    constructor (name, gestationPeriod){
        super(name)
        this.gestation = gestationPeriod
    }
}

let mammal = new Mammal("Zebra", 12);
mammal.sayHi();
```

CoGrammar

Polymorphism

A concept that allows objects of different classes to be treated as objects of a common interface class.

- Polymorphism allows for multiple objects of different classes to have methods with the same name.
- It also allows for us to **override** methods from parent classes in a child class.

Portfolio Assignment: SE

Object-Oriented Programming Design Patterns Implementation

Objective: Implement a set of design patterns using object-oriented programming principles in Python. This project will demonstrate your understanding of OOP and design patterns.

Portfolio Assignment: SE

Requirements:

- Choose at least three design patterns from the following: Singleton, Factory Method, Observer, Strategy, or Composite.
- Implement the selected design patterns in Python, focusing on clean and modular code.
- Integrate the design patterns into a sample application (e.g., a simple game, a data processing tool).
- Provide documentation for each implemented design pattern, explaining its purpose and usage.
- Test the sample application to verify that the design patterns are functioning as expected.

Portfolio Assignment: DS

Recursive Neural Network for Text Classification

Objective: Implement a recursive neural network (RNN) in Python for text classification tasks. This project will showcase your understanding of recursion, OOP, and natural language processing (NLP) techniques.

Portfolio Assignment: DS

Requirements:

- Implement a recursive neural network using Python, utilising libraries such as TensorFlow or PyTorch.
- Use a publicly available text classification dataset (e.g., sentiment analysis, topic classification).
- Use OOP principles to organise the code and represent the neural network architecture.
- Train the RNN model on the dataset and evaluate its performance using appropriate metrics.
- Provide detailed documentation of the implementation, including the dataset used, model architecture, training process, and evaluation results.

Portfolio Assignment: WD

Interactive Recursive Tree Visualizer

Objective: Create a web application that allows users to visualise and interact with recursive tree structures. This project will demonstrate your understanding of recursion and frontend development skills.

Portfolio Assignment: WD

Requirements:

- Implement a recursive function in JavaScript to generate tree structures.
- Use HTML, CSS, and JavaScript to create an interactive visualization of the generated trees.
- Allow users to modify the tree structure (e.g., add/remove branches) through user interactions.
- Utilize OOP principles in JavaScript to represent tree nodes as objects.
- Update the visualization dynamically as users modify the tree structure.
- Provide clear documentation on how to use the application.

Additional Resources

- GeeksForGeeks - Top 50 Recursion Interview Questions
- LinkedIn - Best practices for avoiding Stack Overflow
-

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