Welcome to this CoGrammar lecture: Recursion

The session will start shortly...

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

Software Engineering Session Housekeeping

- For all **non-academic questions**, please submit a query: www.hyperiondev.com/support
- We would love your **feedback** on lectures: Feedback on Lectures

Software Engineering Session 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 throughout the session, should you wish to ask any follow-up questions.
- 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

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

Rafiq Manan

Charlotte Witcher

Nurhaan Snyman

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 Progression Overview

To be eligible for a certificate of completion, students must fulfil three specific criteria. These criteria ensure a high standard of achievement and alignment with the requirements for the successful completion of a Skills Bootcamp.

✓ Criterion 1 - Meeting Initial Requirements

Criterion 1 involves specific achievements within the first two weeks of the program.
To meet this criterion, students need to:

- Attend a minimum of 7-8 hours per week of guided learning (lectures, workshops, or mentor calls) within the initial two-week period, for a total minimum of 15 guided learning hours (GLH), by no later than 15 September 2024.
- Successfully complete the Initial Assessment by the end of the first 14 days, by no later than 15 September 2024.

Skills Bootcamp Progression Overview

✓ Criterion 2 - Demonstrating Mid-Course Progress

Criterion 2 involves demonstrating meaningful progress through the successful completion of tasks within the first half of the bootcamp.
To meet this criterion, students should:

- Complete 42 guided learning hours and the first half of the assigned tasks by the end of week 7, no later than 20 October 2024.

CoGrammar

Skills Bootcamp Progression Overview

✓ Criterion 3 - Demonstrating Post-Course Progress

Criterion 3 involves showcasing students' progress after completing the course.
To meet this criterion, students should:

- Complete all mandatory tasks before the bootcamp's end date. This includes any necessary resubmissions, no later than 22 December 2024.
- Achieve at least 84 guided learning hours by the end of the bootcamp, 22 December 2024.

CoGrammar


SKILLS
FOR LIFE
SKILLS BOOTCAMPS
Department
for Education
CoGrammar
Recursion

Poll

1. **What is recursion in programming?**
   A. A method of solving problems iteratively.
   B. A method of solving problems using loops.
   C. A function that calls itself.
   D. A function that calls another function.

Poll

2. **When should you use recursion?**
   A. When the problem can be easily solved using loops.
   B. When the problem can be divided into smaller, similar sub problems.
   C. When the problem requires complex data structures.
   D. When the problem cannot be solved using any other method.

Learning Outcomes

- Describe the concept of recursion and its role in programming.
- Describe the concept of iteration and its role in programming.
- Identify when recursion is an appropriate solution and when it may not be.
- Implement recursive functions to solve problems.

What is Recursion?

- Recursion is a programming technique where a function calls itself to solve a problem by breaking it down into smaller, similar sub problems.
- This self-referential approach allows for elegant and concise solutions to certain types of problems.
- In recursion, a base case is typically defined to provide a stopping condition for the recursive calls. When the base case is reached, the recursion unwinds, and the function returns results back up the call stack.

Why Recursion?

- Recursion offers simplicity, modularity, and flexibility in solving certain types of problems.
- It allows for concise and elegant code, promotes code reuse, and is particularly effective for tackling problems with repetitive, self-similar structures.
- While it may not be suitable for every problem, recursion is a valuable tool in a programmer's toolkit, enabling the solution of complex problems with clarity and efficiency.

What is Iteration?

- Iteration is a fundamental programming concept that involves repeating a set of instructions or a process multiple times until a specific condition is met.
- Iteration provides a way to execute code repeatedly, often with slight variations or modifications each time.
- In iteration, a loop structure is commonly used to achieve repetition.
- Iteration involves executing a block of code repeatedly until a certain condition is satisfied. This allows for the efficient handling of repetitive tasks and is essential for automating processes in programming.

Types of Iteration

- **Count-controlled Iterations**

  Where the number of repetitions is predetermined based on a fixed count or iteration variable. For example, a loop may be set to execute a certain number of times specified by a loop counter or a predefined limit.

- **Condition-controlled Iterations**

  Where the repetition continues until a specific condition evaluates to false. The condition is typically based on the evaluation of a boolean expression, such as checking for the end of a data stream or the satisfaction of a particular condition.

Why Iteration?

- Iterations excel in providing efficiency, readability, and direct control over execution in a broader range of situations.
- Iterations provide a versatile alternative to recursion, especially in scenarios where simplicity, modularity, and flexibility are not the primary concerns.
- Iterations typically offer better performance and predictable resource usage compared to recursion, making them suitable for handling large datasets or deep levels of nesting.

Recursion vs Iteration

- Recursion and iteration (loops) can be used to achieve the same results. However, unlike loops, which work by explicitly specifying a repetition structure, recursion uses continuous function calls to achieve repetition.
- Recursion is a somewhat advanced topic and problems that can be solved with recursion can also most likely be solved by using simpler looping structures.
- Recursion is a useful programming technique that, in some cases, can enable you to develop natural, straightforward, simple solutions to otherwise difficult problems.

Recursion vs Iteration ...

- The following guidelines will help you to decide which method to use depending on a given situation:
  - **When to use recursion?**

    When compact, understandable, and intuitive code is required and where you want to avoid the need for explicit variable state management.
  - **When to use iteration?**

    When there is limited memory and faster processing is required and where more direct control over the flow of execution is required.

The Case for Recursion

- Recursion is suitable for solving problems that exhibit repetitive, self-similar structures, such as:
  - factorial calculation
  - Fibonacci sequence generation
  - tree traversal (visiting all the nodes in a tree data structure)
- Recursion requires careful handling of base cases to avoid infinite recursion or too many recursive calls, which can lead to stack overflow errors.

Recursive Functions

- Normally a recursive function uses conditional statements to determine whether or not to call the function recursively.
- The main benefits of recursion are:
  - compactness of code,
  - ease of understanding the code,
  - and having fewer variables.

Main Components

- **Base Case**

  The function returns a value when a certain condition is satisfied, without any other recursive calls.

- **Recursive Case**

  The function calls itself with an input that is a step closer to the base case.

Base Case Component

- Base cases are the terminating conditions that stop the recursion and prevent the function from infinitely calling itself.
- These are the simplest instances of the problem that can be solved directly without further recursion.
- Without base cases, the recursive function would continue indefinitely, leading to stack overflow errors or infinite loops.

Recursive Case Component

- Recursive cases define how the function calls itself with modified inputs to solve smaller instances of the same problem.
- In recursive cases, the function applies the same algorithm to a reduced or modified version of the original problem.
- By breaking down the problem into smaller sub problems and solving each sub problem recursively, the function gradually approaches the base case(s).

Recursive Function Structure

```
def recursive_function(input):
    # Base case(s)
    if base_condition(input):
        # Return the result directly
        return base_result

    # Recursive case(s)
    else:
        # Modify the input and make a recursive call
        modified_input = modify_input(input)
        recursive_result = recursive_function(modified_input)

        # Further processing of the recursive result
        final_result = process_result(recursive_result)
        return final_result
```

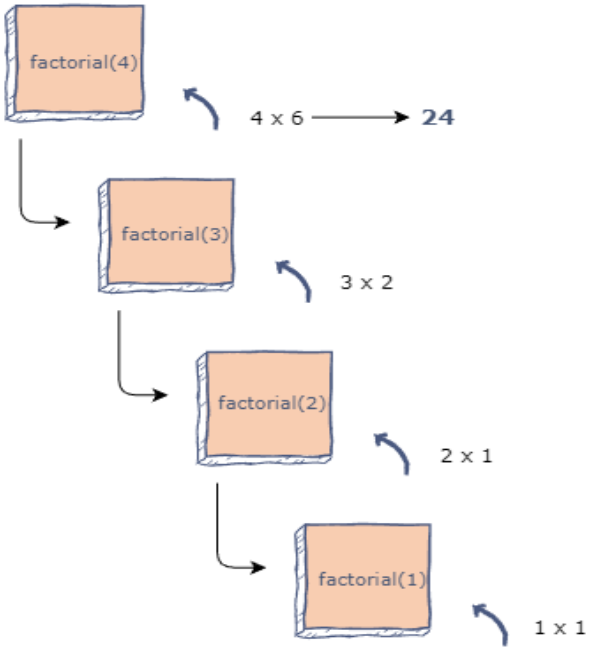
Recursive Function Structure ...

- The function first checks for base cases using if statements.
- If the base condition is met, the function returns the base result directly.
- If the base condition is not met, the function proceeds to the recursive case(s).
- It modifies the input parameters and makes a recursive call to itself with the modified input.
- The process continues recursively until the base case(s) are reached, at which point the recursion unwinds and returns the final result back up the call stack.

Recursive Function Example

- Computing Factorials
  - Many mathematical functions can be defined using recursion. A simple example is a factorial function.
  - The factorial function, n! describes the operation of multiplying a number by all positive integers less than or equal to itself (excluding zero).
  - For example: 4! = 4 * 3 * 2 * 1

Factorials Diagram



Factorials Code

```python
def factorial(num):
    if num == 1:
        return 1
    else:
        return num * factorial(num-1)
```

Let’s take a short break

CoGrammar

Let’s get coding!

CoGrammar

Poll

1. **What is a base case in a recursive function?**
   A. The case where the function calls itself.
   B. The case where the function returns a value without making further recursive calls.
   C. The case where the function returns None.
   D. The case where the function has reached the maximum recursion depth.

Poll

2. **What is the main advantage of using recursion in programming?**
   A. Improved performance compared to iterative solutions.
   B. Simplicity and elegance of code.
   C. Ability to solve any problem regardless of complexity.
   D. Greater control over program flow.

Conclusion and Recap

CoGrammar

Conclusion and Recap

- By combining base cases and recursive cases, recursive functions effectively break down complex problems into simpler sub problems and solve them iteratively until reaching a termination condition, providing an elegant and efficient approach to problem-solving in programming.

Conclusion and Recap

```
def recursive_function(input):
    # Base case(s)
    if base_condition(input):
        # Return the result directly
        return base_result

    # Recursive case(s)
    else:
        # Modify the input and make a recursive call
        modified_input = modify_input(input)
        recursive_result = recursive_function(modified_input)

        # Further processing of the recursive result
        final_result = process_result(recursive_result)
        return final_result
```

Conclusion and Recap

- **Use recursion:**

  When compact, understandable, and intuitive code is required and where you want to avoid the need for explicit variable state management.

- **Use iteration:**

  When there is limited memory and faster processing is required and where more direct control over the flow of execution is required.

Learner Challenge – Option 1

- *Write a recursive function to reverse a string. For example, reversing "recursion" should return "noisrucer".*

1. Write a function that takes a string and returns it in reverse using recursion
2. Questions to Reflect:

- How does the recursion break down the string into smaller pieces?
- What happens if the base case is not defined properly?
- How would this problem be handled iteratively? What benefits or challenges does recursion provide here?

Learner Challenge – Option 2

- *Write a recursive function to find the greatest common divisor (GCD) of two numbers using Euclid's algorithm. The GCD of two numbers is the largest number that divides both without leaving a remainder.*

1. **Write a function to find the GCD of two integers using recursion.**
2. **Questions to Reflect:**

- How does the recursive function reduce the problem of finding the GCD?
- Why is the base case important in preventing infinite recursion?
- Can you compare this approach to finding the GCD iteratively? What advantages does the recursive approach offer?

Questions and Answers

CoGrammar

Thank you for attending

SKILLS FOR LIFE
SKILLS BOOTCAMPS

Department for Education

CoGrammar