



# CoGrammar

## Lecture 11: Recursion

**SKILLS  
FOR LIFE**

**SKILLS BOOTCAMPS**



Department  
for Education

## Lecture Housekeeping

---

- The use of disrespectful language is prohibited in the questions, this is a supportive, learning environment for all - please engage accordingly.  
**(FBV: Mutual Respect.)**
- 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 Open Classes.  
You can submit these questions here: [Open Class Questions](#)

## Lecture Housekeeping cont.

---

- For all **non-academic questions**, please submit a query: [www.hyperiondev.com/support](https://www.hyperiondev.com/support)
- Report a **safeguarding** incident: [www.hyperiondev.com/safeguardreporting](https://www.hyperiondev.com/safeguardreporting)
- We would love your **feedback** on lectures: [Feedback on Lectures](#)

# Lecture Objectives

- Explore what Recursion is and how to use it.
- Explore what Recursion is and how to use it.
- Explore what Recursion is and how to use it.

# What is recursion?

Recursion occurs when a function makes a call to itself.

- ★ Used when we have a certain set of operations that need to perform repeatedly on changing values.
- ★ Let's us perform complex operations in a straightforward manner
- ★ Allows us to work with different data structure like Linked Lists and Graphs

```
def factorial(n):  
    if (n == 0):  
        return 1  
  
    return n * factorial(n - 1)
```

# Recursion vs Iteration

Both recursion and iteration can be used to perform the same operations, so it's up to the programmer to decide which approach their code will benefit from most

- ★ **Recursion:**

- The code is easier to write and understand

- Allows you to work with advanced data structures like Linked Lists, Trees and Graphs

- ★ **Iteration:**

- Less resource intensive

- Smaller learning curve

# Recursion vs Iteration

## Recursion

```
def partial_sum_recursive(n):  
    if (n == 1):  
        return 1  
  
    return n + partial_sum(n - 1)
```

## Iterative

```
def partial_sum_iterative(n):  
    total = 0  
  
    while n > 0:  
        total += n  
        n -= 1  
  
    return total
```

# Creating Recursive Functions

There are 3 main features of a recursive function that we need to look at, within each feature, we need to answer a question.

- ★ **Arguments** - What values are we taking in, and how will one or more of these values change in order for our function to meet the base case.
- ★ **Base Case** - What condition/s do our arguments need to meet in order to exit the function.
- ★ **Recursive Call** - What is our goal output from each function call.



# Creating Recursive Functions: Arguments

We use parameters/arguments to represent the values that we want to perform our operations on and also the values that we want to test against the base case. We need to remember the following:

- You can think of the argument like the *i* in `for i in values`, for each iteration of the `for` loop, the value of *i* will represent the value we need to work with for a particular iteration
- At least one argument needs to change on each recursive call, this argument will need to be tested against our base case so that we can exit the function.

# Creating Recursive Functions: Base Case

The base case tests the condition that needs to be met in order for us to exit our recursive function, once the condition has been met, we need to return a value that will help us get our desired output. Our base cases should:

- Account for the change in the arguments.
- Should return a value that will help us get our desired output.

# Creating Recursive Functions: Recursive Call

Our recursive call need to meet the requirements of our desiged output, we can make the call alongside a calculation or on it's own. Some important things to remember:

- We need to have a return after we make our recursive call.
- The argument/s that are being checked in the base case need to change in a manner that allows them to be tested correctly
- We can perform our operations while calling our function, or assign the results of the function call to a variable and use that in a calculation before returning the output

# Let's Breathe

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

## Real Life Example

Pretend you are seated in an exam room where all of the students are seated in one long straight line. You need to add your seat number to the exam paper, but the organisers forgot to provide that detail, and there are no numbers on the seats. You decide to ask the student in front of you for their seat number as you know that your seat number would be their seat number + 1, but it turns out they also don't know their seat number. It turns out that no one knows their seat number. so after you initially asked the person in front of you, each person after that did the same until the first person in line was reached. Seeing that they had no one in front of them, they could positively assert that they are the first person and their seat must be seat number 1.

# Visualising the Solution

`seat = 1 + person_ahead`

`seat = 1 + person_ahead`

`seat = 1 + person_ahead`

`seat = 1 + person_ahead`

`seat = 1`



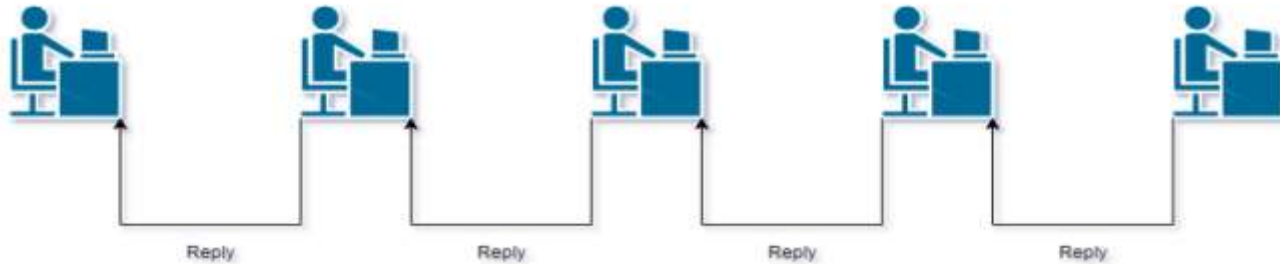
`seat = 1 + (4) = 5`

`seat = 1 + (3) = 4`

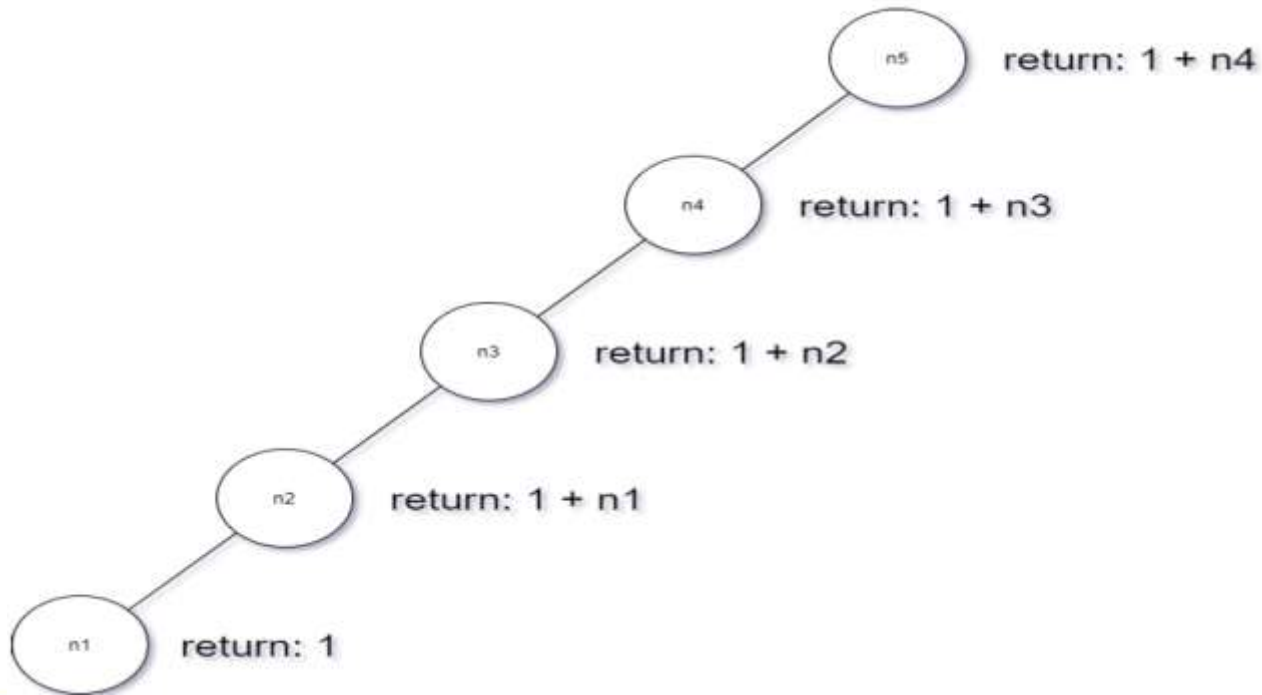
`seat = 1 + (2) = 3`

`seat = 1 + (1) = 2`

`seat = 1`



# Visualising using a tree





# The Code

The example seating example would make use of a linked list if we wanted to code it out properly, but for this example, we will just pass a normal list and only check if there are any students ahead before returning the seating positions:

```
def get_seating_position(student_asking: list):  
    if (len(student_asking) == 1):  
        return 1  
  
    return 1 + get_seating_position(student_asking[1::])
```

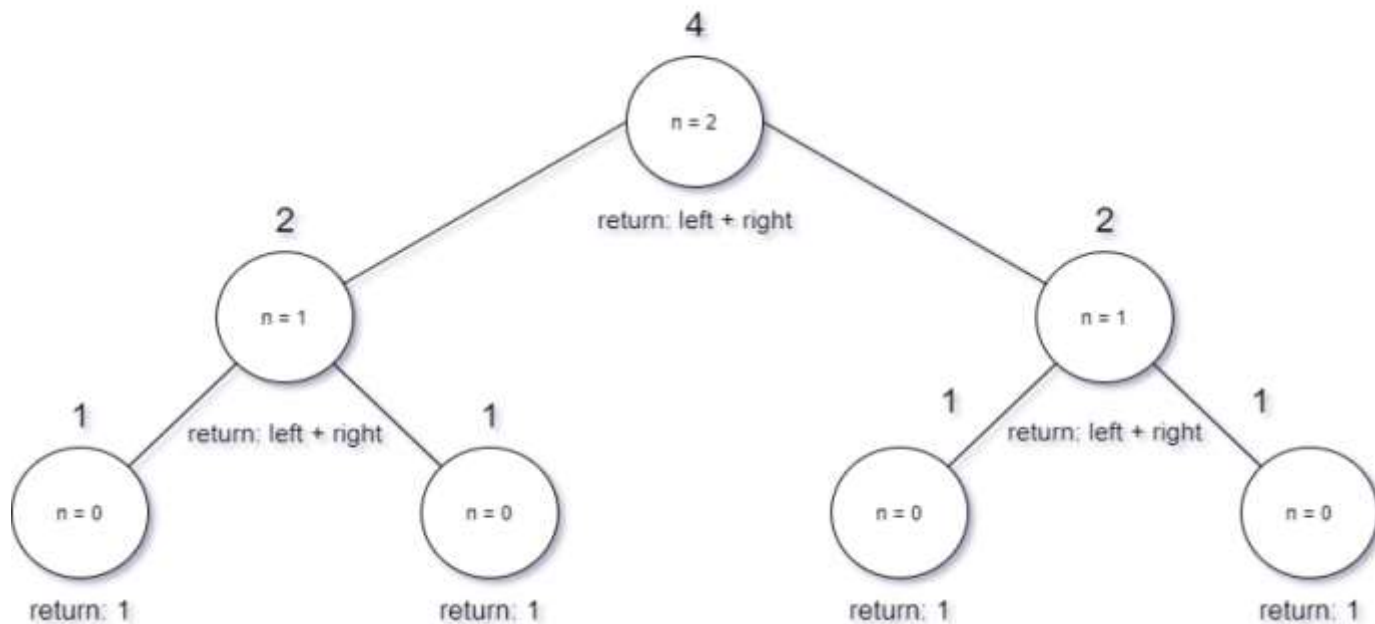


# Several Recursive Calls

- ★ In mathematics, we know that  $2^n$  would perform  $2 \times 2^n$  times, but we also know that if we have 1 as an exponent, we will just get the base number. so we can also look at  $2 \times 2$  as  $(2^1) \times (2^1)$
- ★ Further more, we know that any base with an exponent of 0 will return 1, so we can look at the value 2 as  $(2^0) + (2^0)$
- ★ With this knowlegde, we can build our own recursive function for calculating exponents, but first, let's look at the example graph

# Exponents Tree

total =  $2^n$  where  $n = 2$



# Code

```
def two_to_the_power_of(n):  
    if (n == 0):  
        return 1  
  
    return two_to_the_power_of(n - 1) + two_to_the_power_of(n - 1)
```

**OR**

```
def two_to_the_power_of(n):  
    if (n == 0):  
        return 1  
  
    left = two_to_the_power_of(n - 1)  
    right = two_to_the_power_of(n - 1)  
  
    return left + right
```

# Why Visualise with Tree?

Each node represents the function, and each child that is directly linked to a node represents the amount of times the function calls itself.

- ★ Tree are a good way to visualise our recursive functions as we can map out the depth of our recursion for each input that we want to test.
- ★ By using a graph, we are able to identify a base case for our operations.
- ★ We are able to identify the calculation that needs to be performed when returning values within the function.

# Issues with Recursion

Looking at our graphs, and also keeping in mind that each function needs to wait for the function that it has called before it can return it's own value, we can see how inefficient recursion can be. This problem is made worse when we have more recursive calls in a single function. There are two main issues that you might run into when performing recursion:

- ★ **Hanging application:**  
Happens when you too many functions running at the same time.
- ★ **Stack Overflow error**  
happens when your recursive calls create too many objects in memory that you run out of the allotted memory for the application.

# Mitigating the Issues

## Hanging Application:


- ★ When you look at the graphs, you will often spot patterns, there will be many instances where the same values are being passed
- ★ Using a technique known as memoization, we can store past checks and return their result whenever our arguments have similar values.

## StackOverflow Error:

- ★ This issue is best handled with a good base case.
- ★ Making sure that your base case covers all possibilities is key, if you know that your output might not be an exact number, it's better to use less than or greater than operators.



# Which of these is not a benefit of recursion?

- A. It helps shorten code blocks
  - B. It makes our code run faster
  - C. Let you work with advanced data structures
- 

# Which error might you commonly encounter with Recursion when your Base Case isn't set up well?

- A. Bad Argument Error
- B. Syntax Error
- C. The need to drink more coffee
- D. Stack Overflow



# CoGrammar

## Q & A SECTION

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



# CoGrammar

Thank you for joining!