

# Recursion

Recursion is a powerful concept in computer science, where a function calls itself to solve a problem.

Its powerful concept where a function calls itself to solve a problem.



Before we start exploring recursion in-depth, let's quickly review the basic structure of a function in Python. It consists of two main portions

the base case

the recursive case

Base Case:

The base case is the condition that allows the recursion to stop. Every recursive function must have a base case. This is the condition under which the function stops calling itself and returns a result. The base case prevents infinite recursion.

Recursive Case:

The recursive case is the part where the function calls itself, i.e. Inside the function, there will be a call to the same function, but with a smaller or modified version of the original problem.

The recursive case works towards reaching the base case and, in turn, solving the overall problem.

Imagine you want to make a stack of pancakes.



To create the stack, you'll pour a ladle of pancake batter onto a hot pan, and when the pancake is cooked on one side, you'll flip it over.

You'll pour a ladle of pancake batter onto a hot pan, and when the pancake is cooked on one side, you'll flip it over



pouring batter

cooking

flipping

You'll repeat this process until you have a delicious stack of pancakes. This repetitive process of pouring batter, cooking, and flipping represents the concept of recursion. Each time you make a pancake, you follow the same steps (pouring, cooking, flipping) until the stack is complete.

In programming, recursion follows a similar idea. A function calls itself to break down a problem into smaller, more manageable tasks. This recursive process continues until a base case is reached, where the function stops calling itself, just like you stop making pancakes when the stack is done.

Let's understand it with a classic example of a recursive function to calculate the factorial of a number.

Factorial is a mathematical operation, It is used to find the product of all positive integers up to a given number.

For example:

The factorial of 5 is calculated as  $5 \times 4 \times 3 \times 2 \times 1$ .

where  $4 \times 3 \times 2 \times 1$  can be written as a factorial of 4

So the generalised formula for factorial is :  $\text{factorial}(n) = n * \text{factorial}(n-1)$

Here is the code to calculate the factorial of a number:

```
• def factorial(n):  
    # Base case: factorial of 0 or 1 is 1  
    if n == 0 or n == 1:  
        return 1  
    else:  
        facto=n * factorial(n-1)  
        # Recursive case:  $n! = n * (n-1)!$   
        return facto
```

Now let's calculate the factorial of 5.

(Add this code below to the above code in Python)

- ```
result = factorial(5)

print("Factorial of 5 is:", result)
```

Now, let's manually track the value of n at each step when calculating the factorial of 5:

`factorial(5)` calls `factorial(4)` because  $n * \text{factorial}(n-1)$  implies  $5 * \text{factorial}(5-1)$ .

`factorial(4)` calls `factorial(3)` because  $4 * \text{factorial}(4-1)$ .

`factorial(3)` calls `factorial(2)` because  $3 * \text{factorial}(3-1)$ .

`factorial(2)` calls `factorial(1)` because  $2 * \text{factorial}(2-1)$ .

`factorial(1)` returns 1, as it hits the base case.

Essentially, in this, when we enter 5, first, the function checks for the base case. As 5 is neither 0 nor 1, it does not return 1 and moves on to the recursive case. In the recursive function, we can see that to calculate the factorial of 5, the function has to calculate the factorial of 4, for which the function has to calculate the factorial of 3, and so on, till the recursive case is equal to the base case of 1, which happens after the calculation of 2 factorial and then the recursive case stops.

You can see how the recursion works by breaking the problem into smaller subproblems until it reaches the base case.