Recursion in Python

Recursion is when a function calls itself. Recursion is a common mathematical and programming concept. It means that a function calls itself. This has the benefit of meaning that you can loop through data to reach a result.

Recursion in Python is a method where a function calls itself to solve a smaller part of a larger problem. It's commonly used when a problem can be broken down into similar subproblems. Each recursive call should bring the problem closer to a base case, which stops the recursion.

A simple recursive function that counts down from 5:

```
def countdown(n):
  if n <= 0:
    print("Done!")
  else:
    print(n)
    countdown(n - 1)

countdown(5)
```

Output:

```
5
4
3
2
1
Done!
```

Explanation:

Let's understand what's happening step by step:

1. The function countdown(n) takes one argument — a number n.
2. It checks if n <= 0.
   - If yes → it prints "Done!" → this is the base case (stopping condition).
3. If n is greater than 0 → it prints the current value of n and then calls itself again with n - 1.
   - This is the recursive case (repeating step).

Base Case and Recursive Case

Every recursive function must have two parts:

- A base case - A condition that stops the recursion
- A recursive case - The function calling itself with a modified argument

Recursion in Python

Without a base case, the function would call itself forever, causing a stack overflow error.

How recursion works here:
When we call countdown(5), this happens:

Function Call	Action	Next Call
countdown(5)	prints 5	countdown(4)
countdown(4)	prints 4	countdown(3)
countdown(3)	prints 3	countdown(2)
countdown(2)	prints 2	countdown(1)
countdown(1)	prints 1	countdown(0)
countdown(0)	prints Done!	stops

The recursion keeps reducing n by 1 each time until n becomes 0, at which point the base case stops further calls.

Why to use Recursion?
- Solves problems with a repeating or self-similar structure (e.g., trees, graphs, factorial, Fibonacci).
- Breaks complex problems into smaller, easier subproblems (divide and conquer).
- Makes code simpler and cleaner for problems naturally defined recursively.
- Essential for tree and graph traversal algorithms (e.g., depth-first search).
- Used in mathematical computations like permutations, combinations, and power sets.

Solved Example 1: Find Sum of First N Natural Numbers

Program:
```
def find_sum(n):
    if n == 0:
        return 0
    else:
        return n + find_sum(n - 1)

print("Sum =", find_sum(5))
```

Output:
Sum = 15

Explanation:

- This program adds all numbers from 1 to 5 using recursion.
- Each time the function calls itself with a smaller number until n becomes 0.
- The sum is then calculated as:
- 5 + 4 + 3 + 2 + 1 + 0 = 15

Solved Example 2: Find Factorial of a Number

Program:

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

print("Factorial =", factorial(5))
```

Output:

Factorial = 120

Explanation:

- Factorial means multiplying a number by all numbers before it.
- factorial(5) → 5 * 4 * 3 * 2 * 1 = 120.
- Recursion stops when n == 1.

Solved Example 3: Print Numbers from 1 to N

Program:

```
def print_numbers(n):
    if n == 0:
        return
    print_numbers(n - 1)
    print(n)

print_numbers(5)
```

Output:

1
2
3
4
5

Explanation:

- This program prints numbers in **increasing order** because the recursive call (print_numbers(n-1)) happens **before** printing n.

- When recursion starts to return (unwind), the numbers print one by one.

Solved Example 4: Find the Power of a Number

Program:

```
def power(base, exp):
    if exp == 0:
        return 1
    else:
        return base * power(base, exp - 1)

print("2^5 =", power(2, 5))
```

Output:

2^5 = 32

Explanation:

- To find 2^5, the function multiplies 2 by itself 5 times.
- power(2,5) → 2 * power(2,4) → 2 * 2 * 2 * 2 * 2 = 32.
- The base case (if exp == 0) returns 1, which stops recursion.

What is the Fibonacci Series?

The **Fibonacci sequence** is a series of numbers where:
Each number is the sum of the two numbers before it.

Example:

0, 1, 1, 2, 3, 5, 8, 13, 21, ...
That means:

- 0 + 1 = 1
- 1 + 1 = 2
- 1 + 2 = 3
- 2 + 3 = 5

...and so on.

Fibonacci Series using Recursion

Program:

```
def fibonacci(n):
    if n <= 1:
        return n
    else:
        return fibonacci(n - 1) + fibonacci(n - 2)

# Print first 10 Fibonacci numbers
for i in range(10):
```

```
print(fibonacci(i), end=" ")
```

Output:
0 1 1 2 3 5 8 13 21 34

Explanation:
- Base cases:

When n <= 1, function just returns n (i.e., 0 or 1).
- Recursive case:

Each call adds the two previous Fibonacci numbers:
fibonacci(n) = fibonacci(n-1) + fibonacci(n-2)
- The loop prints the first 10 terms.

Fibonacci Series using a While Loop (Non-Recursive)
Program:

```
n = 10  # number of terms
a, b = 0, 1
count = 0

while count < n:
    print(a, end=" ")
    a, b = b, a + b
    count += 1
```

Output:
0 1 1 2 3 5 8 13 21 34

Explanation:
- a and b store the previous two Fibonacci numbers.
- In each iteration, a becomes the next term and b becomes the sum.
- Continues until the desired number of terms is printed.

What is Recursion with Lists?

Recursion with lists means using a recursive function to perform an operation (like summing, printing, or searching) on elements of a list, **without using loops**.
Each recursive call usually:
- Works on **one element** (like the first or last one),
- Then calls itself again on the **remaining list**,
- Until the list becomes **empty** — which is the **base case**.

Basic Structure

```
def function_name(list):
    if base_condition:   # usually when list is empty
```

```
    return something
  else:
    return some_operation(list[0], function_name(list[1:]))
```

Solved **Example 1: Sum of All Elements in a List**

Program:

```
def list_sum(numbers):
    if len(numbers) == 0:       # Base case: if list is empty
        return 0
    else:
        return numbers[0] + list_sum(numbers[1:])  # Add first element + sum of
rest

nums = [2, 4, 6, 8]
print("Sum of list elements:", list_sum(nums))
```

Output:

```
Sum of list elements: 20
```

Explanation:

- The function keeps adding the first number and calling itself on the rest of the list.

→ 2 + (4 + (6 + (8 + 0))) = 20

- The recursion stops when the list becomes empty (len(numbers) == 0).

Solved Example 2: Print All Elements of a List Recursively

Program:

```
def print_list(items):
    if not items:   # Base case: empty list
        return
    print(items[0])
    print_list(items[1:])  # Recursive call for rest of list

my_list = ["Apple", "Banana", "Cherry"]
print_list(my_list)
```

Output:

```
Apple
Banana
Cherry
```

Explanation:

- The function prints the first element, then calls itself on the remaining list.
- It continues until the list becomes empty.

Solved Example 3: Find Maximum Element in a List Using Recursion

Program:

```
def find_max(lst):
    if len(lst) == 1:      # Base case: only one element
        return lst[0]
    else:
        max_rest = find_max(lst[1:])     # Find max of remaining list
        return lst[0] if lst[0] > max_rest else max_rest

numbers = [10, 25, 8, 99, 56]
print("Maximum element:", find_max(numbers))
```

Output:

Maximum element: 99

Explanation:

- Compares the first element with the max of the remaining list.
- The largest value is returned step-by-step.