Recursion in Python

Last Updated : 20 Sep, 2025

Recursion is a programming technique where a function calls itself either directly or indirectly to solve a problem by breaking it into smaller, simpler subproblems.

In Python, recursion is especially useful for problems that can be divided into identical smaller tasks, such as mathematical calculations, tree traversals or divide-and-conquer algorithms.

Working of Recursion

A **recursive function** is just like any other Python function except that it calls itself in its body. Let's see basic structure of recursive function:

```
def recursive_function(parameters):
    if base_case_condition:
        return base_result
    else:
        return recursive_function(modified_parameters)
```

Recursive function contains two key parts:

- **Base Case:** The stopping condition that prevents infinite recursion.
- **Recursive Case:** The part of the function where it calls itself with modified parameters.

Examples of Recursion

Let's understand recursion better with the help of some examples.

Example 1: Factorial Calculation

This code defines a recursive function to calculate factorial of a number, where function repeatedly calls itself with smaller values until it reaches the base case.

```
def factorial(n):
    if n == 0:  # Base case
        return 1
    else:       # Recursive case
        return n * factorial(n - 1)

print(factorial(5))
```

Output

Explanation:

- **Base Case:** When **n == 0**, recursion stops and returns **1**.
- **Recursive Case:** Multiplies **n** with the factorial of **n-1** until it reaches the base case.

Example 2: Fibonacci Sequence

This code defines a recursive function to calculate nth Fibonacci number, where each number is the sum of the two preceding ones, starting from 0 and 1.

```
def fibonacci(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fibonacci(n-1) + fibonacci(n-2)

print(fibonacci(10))
```

Output

```
55
```

Explanation:

- **Base Cases:** If n == 0, the function returns 0. If n == 1, the function returns 1. These two cases are necessary to stop the recursion.

- **Recursive Case:** function calls itself twice with decrements of n (i.e., fibonacci(n-1) and fibonacci(n-2)), summing results of these calls.

Types of Recursion in Python

Recursion can be broadly classified into two types: **tail recursion** and **non-tail recursion**. The main difference between them is related to what happens after recursive call.

- **Tail Recursion:** The recursive call is the last thing the function does, so nothing happens after it returns. Some languages can optimize this to work like a loop, saving memory.
- **Non-Tail Recursion:** The function does more work after the recursive call returns, so it can't be optimized into a loop.

This code compares tail recursion and non-tail recursion using two versions of factorial function one with an accumulator (tail-recursive) and one with multiplication after recursive call (non-tail-recursive).

```python
def tail_fact(n, acc=1):
    # Base case
    if n == 0:
        return acc
    # Tail recursive call with an accumulator
    else:
        return tail_fact(n-1, acc * n)

def nontail_fact(n):
    # Base case
    if n == 1:
        return 1
    # Non-tail recursive call because the multiplication happens after the call
    else:
        return n * nontail_fact(n-1)

# Example usage
print(tail_fact(5))
print(nontail_fact(5))
```

Output

```
120
120
```

Explanation:

- **def tail_fact(n, acc=1):** - Defines a tail-recursive factorial function with an accumulator acc to store intermediate results.
- **if n == 0: return acc** - Base case: when n reaches 0, return the accumulated result.
- **return tail_fact(n-1, acc * n)** - Tail-recursive call: multiplies acc by n before the call, so no extra work is left after recursion.
- **def nontail_fact(n):** - Defines a non-tail-recursive factorial function.
- **if n == 1: return 1** - Base case: factorial of 1 is 1.
- **return n * nontail_fact(n-1)** - Non-tail call: multiplication happens after the recursive call returns, so more work remains after recursion.

Recursion vs Iteration

Recursion:

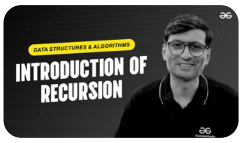
It is often more intuitive and easier to implement when the problem is naturally recursive, like tree traversals. It can lead to solutions that are easier to understand compared to iterative ones.

Iteration:

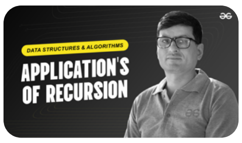
Iteration involves loops (for, while) to repeat the execution of a block of code. It is generally more memory-efficient as it does not involve multiple stack frames like recursion.

When to Avoid Recursion

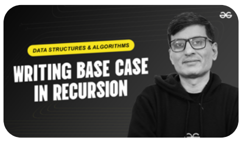
- When the problem can be solved easily with loops.
- When recursion depth is large enough to risk a stack overflow.
- When performance is critical and function call overhead matters.



Introductio of Recursion



Applicatior of Recursic



Writing base case in Recursion

Introduction of Recursion

Visit Course

Suggested Quiz

4 Questions

Predict output of the following program Python def function(n): if n==4: return n else: return 2*function(n+1) print(function(2))

- (A) 4
- (B) 8
- (C) 16
- (D) Runtime Error

Login to View Explanation

1/4

< Previous Next >

Comment

S shardu... + Follow

85