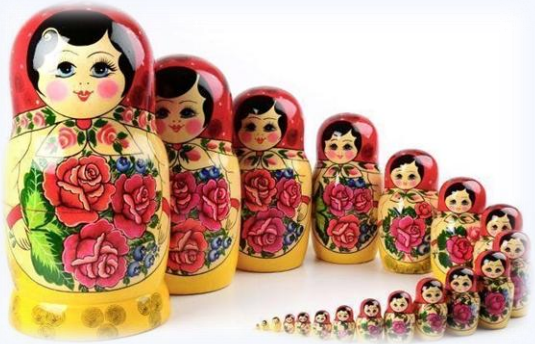
Recursion

Introduction

- **Recursion** is a programming technique where a function **calls itself to solve a problem** by breaking it down into smaller, similar sub problems.

Imagine a set of nested dolls:

- To get to the smallest doll, you keep opening the outer dolls one by one.
- Similarly, a recursive function keeps calling itself until it reaches the “smallest” or simplest version of the problem.



Source: Image from **Medium** Article - **Recursion: The Nesting Doll of Programming** by Chandler Hanson

Components Of Recursion

A well-formed recursive function has **two critical components**:

1. Base Case

- This is the **stopping condition**. When met, the function returns a simple, direct answer (no further self-calls).
- Without a base case, the function would **call itself indefinitely**, leading to **infinite recursion**.

2. Recursive Case

- This is where the function calls itself, but with "**smaller**" or **simpler** arguments. Each recursive call moves the problem **closer to the base case**.

Components Of Recursion

Python

```python
def example_recursive(n):
    if n == 0:
        return some_value
    else:
        return example_recursive(n - 1)
```

Base Case

Recursive Case

Recursion Visualized!

Python

```python
def sum_to_n(n):

    if n == 1:
       return 1
    else:
       return n + sum_to_n(n - 1)

print(sum_to_n(3))
```

Recursion Visualized!

Python

```
print(sum_to_n(3))
```

```
def sum_to_n(3):
    if n == 1: return 1
    else:      return 3 + sum_to_n(2)
```

```
                       def sum_to_n(2):
                         if n == 1: return 1
                           else:      return 2 + sum_to_n(1)
```

```
                                                def sum_to_n(1):
                                                   if n == 1: return 1
                                                   else: return n + sum_to_n(n - 1)
```

Recursion Visualized!

Python

```
print(6)
```

Output

6

Example: 1 – Factorial Function

Python

```python
def factorial(n):

    if n == 0 or n == 1:   # 1. Base Case
        return 1

    else:
        return n * factorial(n - 1) # 2. Recursive Case

print(factorial(5))
```

Output

```
120
```

Base Case: 0! = 1

Recursive Case: n! = n × (n−1)!

Example: 2 – Fibonacci Sequence

Python

```python
def fibonacci(n):
    if n == 0:   # 1. Base Cases
        return 0
    elif n == 1:
        return 1
    else:
        return fibonacci(n - 1) + fibonacci(n - 2) # 2. Recursive Case
print(fibonacci(6))
```

Output

```
8
```

Base Case: fib(0) = 0 & fib(1) = 1

Recursive Case: fib(n) = fib(n–1) + fib(n–2)

When to use Recursion

- The problem is **naturally defined** in terms of **smaller subproblems**.
- You want code that closely follows a **mathematical or logical definition**.
- The input size is **small-to-moderate**, and performance is **not** critical.

When *NOT* to use Recursion

- The required depth of **recursive calls** may **exceed** Python's **recursion limit** (~1,000 by default) – and raises **RecursionError**. (You can increase the limit, but it's risky.)
- You need **maximum performance** for large inputs.
- The problem **doesn't decompose** *naturally* into **smaller identical steps**.

Mistakes to avoid

- **Missing Base Case**: The function never stops calling itself. Always test the simplest input first (e.g., n = 0 or n = 1) to ensure the base case works.
- **Wrong Base Case**: If your base case condition is too strict then smaller inputs slip through and can cause infinite recursion.
- **Not Returning the Recursive Call**: Returned values will always be discarded. Always return your recursive calls if you need their result.

Iteration over Recursion

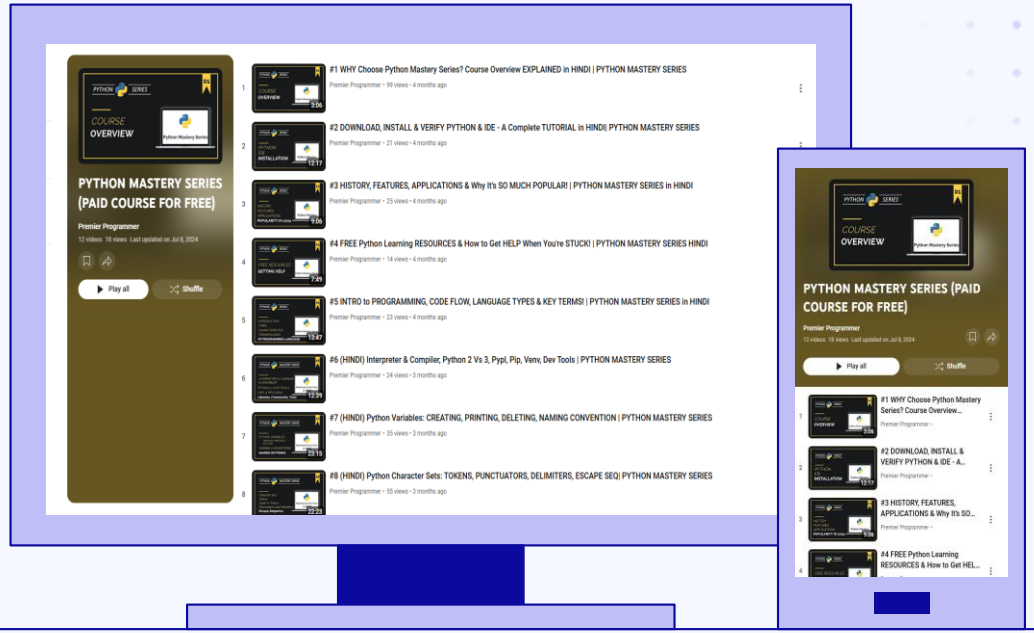
- **Performance Concerns**: If you need to process very large lists or numbers and risk hitting recursion limits, a loop is safer.
- **Simplicity**: Sometimes, iteration can be just as simple with a for or while loop.

Practice Set - 3

Download Link in **Description** and **Pinned Comment**

WATCH

Level up your coding with each episode in this focused Python series.



Next Video!

Practice Set - 3
Solution