📘 Chapter 15: Recursion – Functions That Call Themselves

🎯 What You Will Learn

- Understand what recursion is and how it works step-by-step
- Differentiate between base case and recursive case
- Visualize how the function call stack behaves
- Solve classic problems using recursion (factorial, Fibonacci, sum, reverse string)
- Avoid common recursion traps (like infinite loops and wrong base cases)
- Know when to use recursion vs iteration

🧠 Why This Concept Matters

Recursion allows a function to **call itself** to solve a problem in smaller steps. It's a powerful technique used in:

- Traversing **trees**, **graphs**, and **directories**
- Breaking down problems like **Fibonacci**, **factorials**, **string operations**
- Real-world algorithms (e.g., QuickSort, MergeSort, backtracking)

Many programming tasks are **naturally recursive**, and learning recursion unlocks elegant, clean solutions.

> Think of recursion like nested Russian dolls — each function opens a smaller version of itself until a base case is reached.

🔷 Concept Introduction

What is Recursion?

A **recursive function** is one that calls itself **within its own definition** to solve a smaller part of the same problem.

It consists of two main parts:

- **Base Case** – the stopping condition
- **Recursive Case** – the part where the function calls itself

```python
def countdown(n):
    if n == 0:
        print("Blastoff!")            # Base Case
    else:
        print(n)
        countdown(n - 1)              # Recursive Case

countdown(5)
```

Output:

```
5
4
3
2
1
Blastoff!
```

🔁 10 Must-Know Recursion Patterns

1. Recursive Countdown

Counts down from `n` to 1, then prints "Done". Base case: `n <= 0`.

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

2. Factorial of a Number

Calculates `n!` recursively. Base case: `factorial(0) = 1`.

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

3. Sum of a List

Returns the sum of all elements in a list by adding the first element to the sum of the rest.

```python
def list_sum(lst):
    if not lst:
        return 0
    return lst[0] + list_sum(lst[1:])
```

4. Fibonacci Sequence

Returns the `n`th Fibonacci number. Very inefficient without memoization.

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

5. Reverse a String

Builds the reversed string by calling itself on the substring and appending the first character at the end.

```python
def reverse(s):
    if len(s) == 0:
        return ""
    return reverse(s[1:]) + s[0]
```

6. Check if Palindrome

Returns `True` if the string reads the same forward and backward, by checking first and last characters recursively.

```python
def is_palindrome(s):
    if len(s) <= 1:
        return True
    if s[0] != s[-1]:
        return False
    return is_palindrome(s[1:-1])
```

7. Count Digits in Number

Counts the number of digits in an integer by dividing by 10 recursively.

```python
def count_digits(n):
    if n < 10:
        return 1
    return 1 + count_digits(n // 10)
```

8. Flatten a Nested List

Recursively unpacks nested lists into a single flat list.

```
def flatten(lst):
    result = []
    for item in lst:
        if isinstance(item, list):
            result.extend(flatten(item))
        else:
            result.append(item)
    return result
```

9. Find Maximum in List

Compares the first item to the max of the rest of the list recursively.

```
def max_list(lst):
    if len(lst) == 1:
        return lst[0]
    sub_max = max_list(lst[1:])
    return lst[0] if lst[0] > sub_max else sub_max
```

10. Binary Search Recursively

Efficiently searches a **sorted list** for a target using divide-and-conquer.

```
def binary_search(arr, target, low, high):
    if low > high:
        return -1
    mid = (low + high) // 2
    if arr[mid] == target:
        return mid
    elif arr[mid] > target:
        return binary_search(arr, target, low, mid - 1)
    else:
        return binary_search(arr, target, mid + 1, high)
```

🔼 Top 5 Advanced Recursive Ideas

1. Tail Recursion (Python doesn’t optimize it)

A special case of recursion where the recursive call is the **last** thing in the function. It's memory-efficient in some languages, but **Python does NOT optimize** tail calls.

```
def tail_sum(n, acc=0):
    if n == 0:
        return acc
    return tail_sum(n - 1, acc + n)
```

2. Mutual Recursion

When **two or more functions** call each other in a cycle. Useful in problems like checking even/odd recursively.

```
def is_even(n):
    if n == 0:
        return True
    return is_odd(n - 1)

def is_odd(n):
    if n == 0:
        return False
    return is_even(n - 1)
```

3. Recursion + Memoization

Improves efficiency by **caching** results of previous calls to avoid redundant work. Great for problems like Fibonacci.

```
memo = {}
def fib(n):
    if n in memo:
        return memo[n]
    if n <= 1:
        memo[n] = n
    else:
        memo[n] = fib(n-1) + fib(n-2)
    return memo[n]
```

4. Tree Traversal (Preorder)

Uses recursion to **visit all nodes** in a tree structure. Preorder means: visit current node → left → right.

```
def traverse_tree(node):
    if node is None:
        return
    print(node.value)
    traverse_tree(node.left)
    traverse_tree(node.right)
```

5. Recursion Limit

Python sets a **default limit (~1000)** on recursion depth to prevent a crash. Exceeding it raises a `RecursionError`.

```
import sys
sys.setrecursionlimit(2000)
```

🧪 Mini Quiz (10 Questions)

1. What two parts make up every recursive function?
2. What happens if there's no base case?
3. Predict the output:

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

4. What does this return?

```python
def f(n):
    if n == 1:
        return 1
    return n + f(n - 1)

print(f(4))
```

5. Is this valid?

```python
def f():
    return f()
```

6. What's the base case in factorial recursion?
7. What will `reverse("abc")` return?
8. Which is more memory efficient: recursion or iteration?
9. When should you avoid recursion in Python?
10. What error do you get from excessive recursion?

🧩 Basic Practice Problems (10)

1. Write a recursive function to count from `n` to `1`.
2. Create a function to return the sum of digits in a number.
3. Write a function that calculates factorial using recursion.
4. Reverse a list recursively.
5. Write a recursive power function: `power(x, n)`.
6. Check if a string is a palindrome using recursion.
7. Print elements of a list one by one recursively.
8. Calculate the nth Fibonacci number recursively.
9. Find the minimum in a list using recursion.

10. Count how many even numbers are in a list recursively.

🚀 Intermediate Practice Problems (5)

1. Flatten a nested list using recursion.
2. Implement recursive binary search.
3. Find GCD of two numbers using recursion.
4. Create a recursive directory crawler (conceptual).
5. Count how many times a value appears in a nested structure.

🔧 Debug Challenges

1. Identify the error:

```
def loop(n):
    print(n)
    loop(n + 1)
```

2. Fix the missing base case:

```
def hello():
    print("Hello")
    hello()
```

3. What's wrong here?

```
def fib(n):
    return fib(n - 1) + fib(n - 2)
```

4. Find the logic error:

```
def reverse(s):
    if len(s) == 0:
        return s
    return s + reverse(s[1:])
```

5. Fix and explain:

```
def sum_list(lst):
    return lst[0] + sum_list(lst)
```

✅ Summary: When to Use Recursion

- When the problem can be divided into smaller similar subproblems

- When working with **nested** or **tree-like** structures
- When iterative logic becomes complex or unreadable
- When the recursive solution is more natural to express
- Avoid if Python’s call stack depth becomes a bottleneck

📋 TL;DR Technique Table

#	Pattern	Base Case	Recursive Call Example
1	Factorial	`if n == 0`	`n * factorial(n - 1)`
2	Countdown	`if n == 0`	`countdown(n - 1)`
3	Fibonacci	`if n <= 1`	`f(n-1) + f(n-2)`
4	Reverse String	`if len(s)==0`	`reverse(s[1:]) + s[0]`
5	Sum of List	`if not lst`	`lst[0] + sum(lst[1:])`

🔥 Bonus

- Learn how `functools.lru_cache` improves recursive calls
- Use `sys.getrecursionlimit()` to view stack limit
- Visualize recursion with `Python Tutor`: https://pythontutor.com