Day 17: Recursion in Python

🟢 What is Recursion?

Recursion is a programming technique where a function **calls itself** to solve a problem by breaking it into smaller subproblems. It continues until it reaches a **base case**, which stops the recursion.

🟢 Key Topics Covered Today

1. **Recursive Functions**
2. **Understanding the Base Case**
3. **Common Examples of Recursion**
    - Factorial Calculation
    - Fibonacci Series
    - Sum of Digits
    - Reverse a String
    - Tower of Hanoi
4. **Recursion vs Iteration**
5. **Advantages & Disadvantages of Recursion**
6. **Optimizing Recursion (Memoization & Tail Recursion)**
7. **Handling Recursion Errors**

---

1. Recursive Functions in Python

A **recursive function** is a function that calls itself within its own definition.

◆ Structure of a Recursive Function

A recursive function must have two parts:

1. **Base Case** → The condition that stops the recursion.
2. **Recursive Case** → The function calls itself with a modified argument.

◆ Example: Simple Recursive Function

```
def countdown(n):
  if n <= 0:  # Base case
    print("Blast off!")
    return
  else:
    print(n)
    countdown(n - 1)  # Recursive case


countdown(5)
```

◆ Output:

```
5
4
3
2
1
Blast off!
```

🔹 **Explanation:**

- The function keeps calling itself, reducing n by 1 each time.
- When n reaches 0, it stops.

---

2. Understanding the Base Case

A **base case** is essential in recursion to **prevent infinite recursion**. If no base case exists, the function will keep calling itself indefinitely and cause a RecursionError.

🔹 **Example Without Base Case (Infinite Recursion)**

```
def infinite():
  print("This will go on forever!")
  infinite()


infinite()
```

❌ **This will result in:**

RecursionError: maximum recursion depth exceeded

- Python has a **recursion limit** (default is 1000).
- We can modify this limit using sys.setrecursionlimit() (not recommended).

---

3. Common Examples of Recursion

🟢 **Example 1: Factorial Calculation**

Factorial (n!) of a number n is:

n!=n×(n−1)×(n−2)×…×1n! = n \times (n-1) \times (n-2) \times ... \times 1
Base Case: 0!=1\text{Base Case: } 0! = 1 Recursive Case: n!=n×(n−1)!\text{Recursive Case: } n! = n \times (n-1)!

🔹 **Recursive Function for Factorial**

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


print(factorial(5))  # Output: 120
```

🔹 **Dry Run for factorial(5)**

```
factorial(5) = 5 * factorial(4)
factorial(4) = 4 * factorial(3)
factorial(3) = 3 * factorial(2)
factorial(2) = 2 * factorial(1)
factorial(1) = 1 * factorial(0)
factorial(0) = 1  # Base case
```

Final computation:

5 * 4 * 3 * 2 * 1 = 120

🟢 Example 2: Fibonacci Series

Fibonacci numbers are calculated as:

F(n)=F(n−1)+F(n−2)F(n) = F(n-1) + F(n-2) Base Cases: F(0)=0,F(1)=1\text{Base Cases: } F(0) = 0, F(1) = 1

🔹 Recursive Function for Fibonacci Series

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


print(fibonacci(6))  # Output: 8
```

🔹 Dry Run for fibonacci(6)

fibonacci(6) = fibonacci(5) + fibonacci(4)

fibonacci(5) = fibonacci(4) + fibonacci(3)

fibonacci(4) = fibonacci(3) + fibonacci(2)

...

This **redundant computation** can be optimized using **memoization** (covered later).

🟢 Example 3: Sum of Digits

Find the sum of digits of a number **using recursion**.

🔹 Recursive Function for Sum of Digits

```
def sum_of_digits(n):
  if n == 0:
    return 0
  return n % 10 + sum_of_digits(n // 10)

print(sum_of_digits(1234))  # Output: 10 (1+2+3+4)
```

🟢 Example 4: Reverse a String

🔹 Recursive Function for String Reversal

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

print(reverse_string("hello"))  # Output: "olleh"
```

🟢 **Example 5: Tower of Hanoi**

The **Tower of Hanoi** is a classic problem where you move disks from **source peg to destination peg using an auxiliary peg**.

🔹 **Recursive Function for Tower of Hanoi**

```
def tower_of_hanoi(n, source, destination, auxiliary):
  if n == 1:
    print(f"Move disk 1 from {source} to {destination}")
    return
  tower_of_hanoi(n-1, source, auxiliary, destination)
  print(f"Move disk {n} from {source} to {destination}")
  tower_of_hanoi(n-1, auxiliary, destination, source)

tower_of_hanoi(3, 'A', 'C', 'B')
```

---

4. Recursion vs Iteration

Feature	Recursion	Iteration
Function Calls	Uses function calls repeatedly	Uses loops (for/while)
Memory Usage	Uses more memory (stack calls)	Uses less memory
Speed	Can be slow (repeated calls)	Faster for large loops
Readability	Simpler, elegant for problems like Fibonacci	More complex but efficient

---

5. Optimizing Recursion (Memoization & Tail Recursion)

🟢 **Memoization (Caching Results)**

To avoid redundant computations in Fibonacci:

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

print(fibonacci_memo(50))  # Optimized!
```

🟢 **Tail Recursion**

If the recursive call is **the last operation**, it's **tail recursion**.

```
def factorial_tail(n, acc=1):
  if n == 0:
    return acc
  return factorial_tail(n-1, n*acc)

print(factorial_tail(5))  # Output: 120
```

---

6. Handling Recursion Errors

If recursion is too deep, Python raises:

RecursionError: maximum recursion depth exceeded

To increase the limit (not recommended):

import sys

sys.setrecursionlimit(2000)

🟢 Summary of Key Learnings

✅ **Recursive Functions** → Functions calling themselves
✅ **Base Case** → Stops recursion
✅ **Factorial & Fibonacci** → Classic examples
✅ **Recursion vs Iteration** → Pros & Cons
✅ **Optimizations** → Memoization & Tail Recursion

Mastering **recursion** is essential for **data structures, algorithms, and problem-solving**!