|

Recursion in Python — simple detailed explanation

1) What is recursion? (simple)

Recursion is when a **function calls itself** to solve a problem by breaking it into smaller, similar subproblems.

Short phrase: *solve big problem by solving a smaller version of the same problem, then combine results*.

2) Two easy analogies

1. **Russian nesting dolls (matryoshka)**
   Each doll contains a smaller doll. To get the smallest one you open each one in order, then put them back together. Recursion opens smaller problems until you reach the smallest (base case), then closes/combines results on the way back.
2. **Asking a friend to pass a message down a line**
   You tell the first friend: “If you can’t handle this, ask the next friend the same question.” Eventually someone says “I can handle it” (base case) and the answer is passed back along the chain (returns back up).

3) The two essential parts of any recursive function

- **Base case** — the simplest input/value for which the answer is known immediately; stops the recursion.
- **Recursive case** — the rule that reduces the problem into one or more smaller subproblems and calls the same function on them.

If you forget the base case or the problem doesn't get smaller, recursion keeps going and causes an error.

4) Clear Python example: factorial

Problem: `factorial(n) = n * (n-1) * ... * 1`, with `factorial(0) = 1`.

```python
def factorial(n):
    # base case
    if n == 0:
        return 1
    # recursive case
    return n * factorial(n - 1)

print(factorial(4))  # prints 24
```

Trace for `factorial(4)` (what happens under the hood):

- `factorial(4)` → needs `4 * factorial(3)`
- `factorial(3)` → needs `3 * factorial(2)`
- `factorial(2)` → needs `2 * factorial(1)`
- `factorial(1)` → needs `1 * factorial(0)`
- `factorial(0)` → base case → returns `1`
- Now the returns happen up the chain:
  - `factorial(1)` returns `1 * 1 = 1`
  - `factorial(2)` returns `2 * 1 = 2`
  - `factorial(3)` returns `3 * 2 = 6`
  - `factorial(4)` returns `4 * 6 = 24`

5) Visualize with debug prints (helps understanding)

```python
def factorial(n):
    print("call:", n)
    if n == 0:
        print("base case reached:", n)
        return 1
    result = n * factorial(n - 1)
    print("returning", result, "for", n)
    return result

print(factorial(4))
```

Expected printed flow:

```
call: 4
call: 3
call: 2
call: 1
call: 0
base case reached: 0
returning 1 for 1
returning 2 for 2
returning 6 for 3
returning 24 for 4
24
```

6) Example that shows the *cost* of naive recursion: Fibonacci

Naive recursive Fibonacci:

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

This works but is **very slow** for larger `n` because it recomputes the same values many times (time ≈ exponential in `n`).

Better (memoized) version:

```python
from functools import lru_cache

@lru_cache(None)
def fib(n):
    if n <= 1:
        return n
    return fib(n-1) + fib(n-2)
```

Memoization caches results, turning exponential-time into linear-time.

7) Recursion depth, RecursionError, and tail recursion

- Each recursive call uses a stack frame (memory). Python has a recursion depth limit (commonly **around 1000** in CPython). If you recurse deeper, you'll get

`RecursionError: maximum recursion depth exceeded` .

- You can check/change the limit:

```
import sys
print(sys.getrecursionlimit())   # often 1000
sys.setrecursionlimit(2000)      # be careful: can crash interpreter if set too high
```

- **Tail recursion** (when the recursive call is the last action) can be optimized by some languages, but **CPython does not perform tail-call optimization**. So tail-recursive functions still consume stack frames in Python.

Example tail-style factorial (still uses stack in CPython):

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

8) When recursion is natural and useful

- Working with **trees** (binary trees, ASTs) — each node processes subtrees recursively.
- **Divide-and-conquer** algorithms (merge sort, quick sort).
- **Backtracking** problems (permutations, combinations, Sudoku, N-Queens).
- Parsing nested structures (like JSON or nested parentheses).

When the problem is naturally nested or hierarchical, recursion makes the code simple and clear.

9) When to avoid recursion in Python

- Very deep linear loops (Python recursion depth and overhead make loops often preferable).
- Performance-critical tight loops where function-call overhead matters.
- When recursion yields exponential behavior (like naive Fibonacci) — consider dynamic programming or memoization.

10) Common mistakes and debugging tips

- **Missing/incorrect base case** → infinite recursion → `RecursionError` .
- **Not making progress** (the arguments don't get smaller) → infinite recursion.
- **Using recursion with heavy slicing** (e.g., `lst[1:]`) can cause extra memory/time costs; prefer passing an index or using iterators.

- **Debugging tips:** add `print` statements, test base case directly, use small inputs, step through with a debugger or visualize the call stack.

11) Small practice tasks (with solutions)

1. Sum of list (simple):

```
def sum_list(lst):
    if not lst:           # base case: empty list
        return 0
    return lst[0] + sum_list(lst[1:])
```

Note: `lst[1:]` creates a new list each call — fine for learning, but for large lists prefer index-based recursion.

2. Sum with index (avoids slicing):

```
def sum_list_idx(lst, i=0):
    if i == len(lst):
        return 0
    return lst[i] + sum_list_idx(lst, i+1)
```

3. Reverse a string:

```
def reverse(s):
    if s == "":
        return ""
    return s[-1] + reverse(s[:-1])
```

12) Quick summary (tl;dr)

- **Recursion = function calling itself** to solve a smaller piece of the same problem.
- Always have a **base case** and a **recursive case** that makes progress toward the base case.
- Use recursion for **natural hierarchical problems** (trees, divide-and-conquer, backtracking). Avoid for extremely deep linear tasks in Python because of depth limits and overhead.
- Debug by printing calls/returns, test the base case, and prefer memoization for overlapping subproblems.