Python Recursion

Table of contents

Canonical example: factorial 1
Choosing base cases 2
Linear vs. tree recursion 3
Relationship to induction 4

A **recursive function** is a function that calls itself. Every recursion needs at least one of each:

- **Base case**: a case defined outright, with no further recursive call (`if base_case: return constant`).
- **Recursive step**: a line where the function calls itself on "smaller" input, making progress toward a base case.

Canonical example: factorial

```
def fact(n: int) -> int:
    if not n:              # base case
        return 1
    return n * fact(n-1)   # recursive step
```

Evaluation has two phases:

- **Winding**: recursive calls build up a stack of pending computations (`4 × (3 × (2 × (1 × fact(0))))`). Each call pushes an *activation record*; they complete last in, first out.
- **Unwinding**: once the base case is hit, each pending computation resolves from the inside out.

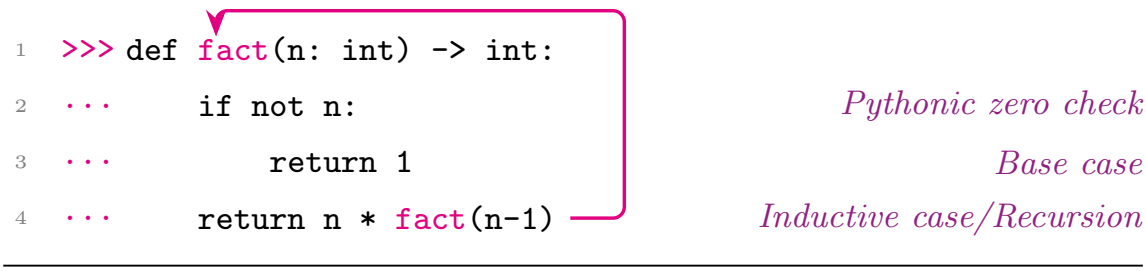

Figure 1: The anatomy of a recursive function. The purple notes on the right name each part. The two occurrences of fact are coloured magenta and joined by an arrow, to show the one thing the definition of recursion turns on: line 4 calls the function being defined on line 1.

See 2025-10-20-recursion[1] for the full winding/unwinding trace of `fact(4)` drawn out line by line.

If a base case is never reached, the recursion **bottoms out** with a `RecursionError: maximum recursion depth exceeded` once Python's call stack limit (1000 by default) is hit — `fact(-1)` does this, since `n` only gets more negative. The limit can be raised with `sys.setrecursionlimit(n)`:

```
>>> import sys
>>> sys.setrecursionlimit(1500)
```

Overflowing the stack and bottoming out are **different bugs with the same error message**. Raising the limit fixes a recursion that legitimately needs more depth than Python allows (`fact(999)` works once the limit is 1001); it never fixes a missing or unreachable base case (`fact(-1)` fails at any limit).

Choosing base cases

Type	Typical base case
`int` (counting down)	`0`
`list`	`[]`
`str`	`''` (empty string)

[1] courses/csse1001/2025-10-20-recursion.pdf

For less obvious base cases, take an example that makes a *single* recursive call and ask what the base case **must be** for that example to come out right. For example, $2^k = 2 \cdot 2^{k-1}$ leaves open what "2 multiplied 0 many times" is. Since $2^1 = 2$ and $2^1 = 2 \cdot 2^0$, the base case must be $2^0 = 1$.

The same advice stated the other way round: set the base cases to whatever makes the **singleton** case (the second-smallest one) work.

Linear vs. tree recursion

Most recursive functions make a single recursive call per invocation (linear recursion, e.g. `fact`, `is_palindrome`, `increasing`, `sum_of_digits`) and so unwind along one chain. Some make more than one (tree recursion), e.g. checking whether any sublist sums to a target:

```
def sublist_sum(xs: list[int], target: int) -> bool:
    if not xs:
        return not target
    return sublist_sum(xs[1:], target-xs[0]) or sublist_sum(xs[1:], target)
```

Two calls per invocation means 2^n leaf calls for a list of n elements. See 2025-10-20-recursion[2] for the call tree drawn out, and for the other branching examples (`num_paths`, `lcs`, `hanoi`, naive `fib`).

Tree recursion can recompute the same subproblem many times. **Dynamic programming** — caching previous results — avoids this when subproblems overlap heavily (the classic example being naive recursive Fibonacci, which is exponential without a cache but linear with one). Here the cache is a module-level dict seeded with the base cases:

```
fib_cache = {0: 1, 1: 1}
def fib(n: int) -> int:
    global fib_cache
    if n in fib_cache:
        return fib_cache[n]
    fib_cache[n] = fib(n-1) + fib(n-2)
    return fib_cache[n]
```

[2] courses/csse1001/2025-10-20-recursion.pdf

Relationship to induction

Recursion is essentially the Principle of Mathematical Induction realised as code: a base case (the $P(0)$ case) plus a step that reduces a general case to a smaller one already known to work (the $P(n) \implies P(\text{succ}(n))$ step). The recursive call *is* the appeal to the induction hypothesis — which is easiest to see in the Towers of Hanoi proof, where the hypothesis is used twice and the code makes exactly two recursive calls.

See python-scope[3] for how local variables/recursive calls interact with scope, and python-composition[4]/python-inheritance[5] for other structuring tools. See towers-of-hanoi[6] for a worked induction/recursion example, and algorithmic-complexity[7] for the vocabulary that makes "exponential without a cache, linear with one" precise — merge sort and quick sort in sorting-algorithms[8] are both recursive.

[3] courses/csse1001/python-scope.pdf
[4] courses/csse1001/python-composition.pdf
[5] courses/csse1001/python-inheritance.pdf
[6] courses/csse1001/towers-of-hanoi.pdf
[7] courses/csse1001/algorithmic-complexity.pdf
[8] courses/csse1001/sorting-algorithms.pdf