



a function that calls itself, on a smaller version of the same problem, until one call finally can't



The base case and the recursive case

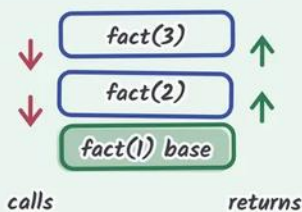
- a recursive function calls itself, directly or through another function, to solve a smaller version of the same problem
- the **base case** is the condition answered directly, with no further call
- the **recursive case** breaks the problem down and calls the function again on a smaller piece of it
- every recursive function needs both parts: a base case alone never calls itself, a recursive case alone never stops

asked rarely, scores big

Infinite recursion and RecursionError

- a base case that's never reached, or a recursive call that never actually shrinks the problem, calls forever
- Python has a built-in limit on how deep calls can stack, roughly a thousand by default, and stops with a RecursionError once it's crossed
- hitting that limit is usually a sign the base case is wrong, missing, or unreachable, not that the limit needs raising

Tracing fact(3), call by call



three calls stack downward on the left in red, the base case turns things around, green arrows carry the answers back up

fact(3), what actually happens

fact(3) calls fact(2), waiting for its answer before it can multiply anything

fact(2) calls fact(1), waiting the same way

fact(1) is the base case, it returns 1 with no further call

fact(2) resumes, computes 2 times 1, returns 2

fact(3) resumes, computes 3 times 2, returns 6

Calls stack, then unwind *

- each call gets its own space to remember its own local variables and its own place in the calculation
- a call waiting on another call sits paused, it can't finish until the one it called returns
- returns happen in the reverse order the calls were made, the last call in is the first call to finish

Recursive vs Iterative

Recursive

- calls itself on a smaller version of the problem
- often reads closer to the problem's own definition, shorter to write
- each waiting call uses extra memory on the call stack, deep recursion can run out

Iterative

- repeats with a loop, no repeated function calls
- uses a small, fixed amount of memory no matter how many repetitions
- can look less like the problem's own mathematical definition

every recursive solution has an iterative twin, either one earns full marks if it's correct

Why choose one over the other

- recursion suits problems naturally defined in terms of smaller versions of themselves, like a factorial or a tree
- a loop is usually the safer choice when the depth could get large, since it never risks a RecursionError
- clarity is a real trade-off, some solutions are dramatically shorter and easier to trust when written recursively

