

Lecture 10

Recursion

Lecture 10 - Recursion

- Objectives
 - Understand a recursive definition
 - Identify base case and recursive case
 - Trace recursive calls and returns
 - Use recursion on numbers, strings, lists and nested data
 - Recognize when iteration may be simpler.

Why Recursion?

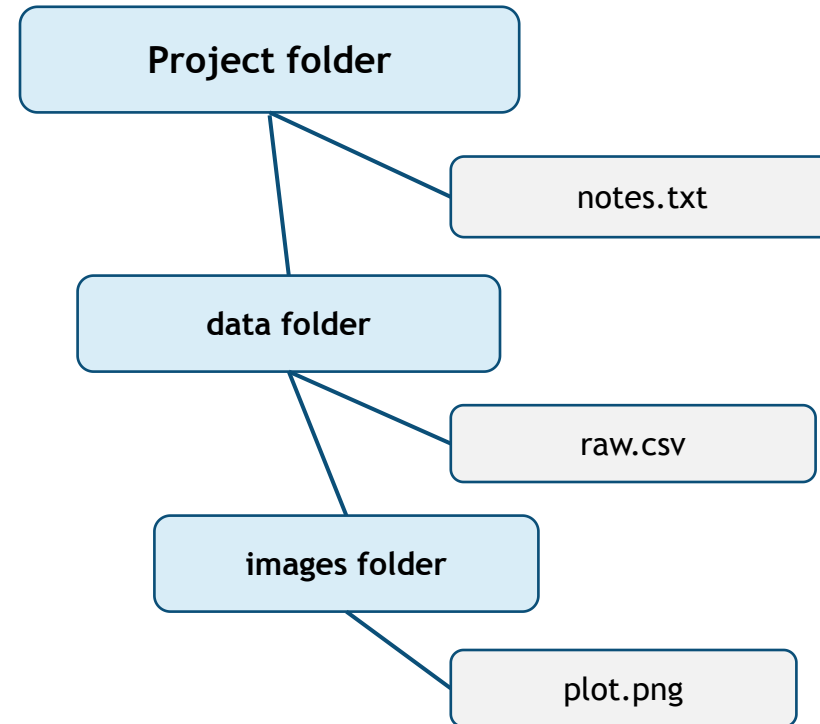
- Some problems contain smaller problems of the same type:
 - Folders contain subfolders
 - Managers supervise teams with sub-teams
 - Menus contain submenus
 - Nested data may contain more nested data
 - “*Divide and Conquer*” algorithms repeat the same operations with smaller sets
 - Binary search in ordered lists
- In these cases, the same procedure can be applied again at each level.

Recursive structure → recursive solution



Real Problem: Folders Inside Folders

- Task: count all files in a folder
 - A folder may contain files
 - It may also contain subfolders, and the number of levels is unknown
 - Each subfolder must be processed in the same way.
- Question: how deeply can folders be nested?



What Is Recursion?

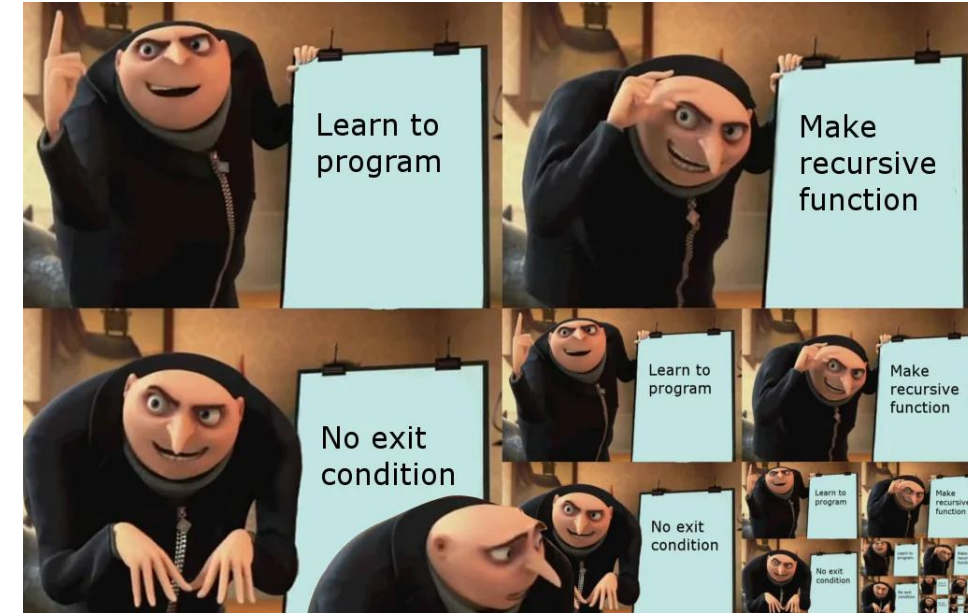
- A **recursive function** is a function that *calls itself*
- It solves a problem by reducing it to a smaller instance of the same problem
- The process must eventually stop.

```
def recursive_function(problem):  
    if simple_enough:  
        return result  
  
    return combine(  
        recursive_function(smaller_problem)  
    )
```

Same problem, smaller input.

The Two Essential Components

- Every recursive solution needs:
 - **Base case:** the case that is solved directly.
 - **Recursive case:** reduce the problem and call the function again.
- Without a base case, recursion does not stop.
- Without reduction, recursion never reaches the base case.



Base case
stop



Recursive case
smaller problem

First Example: Countdown

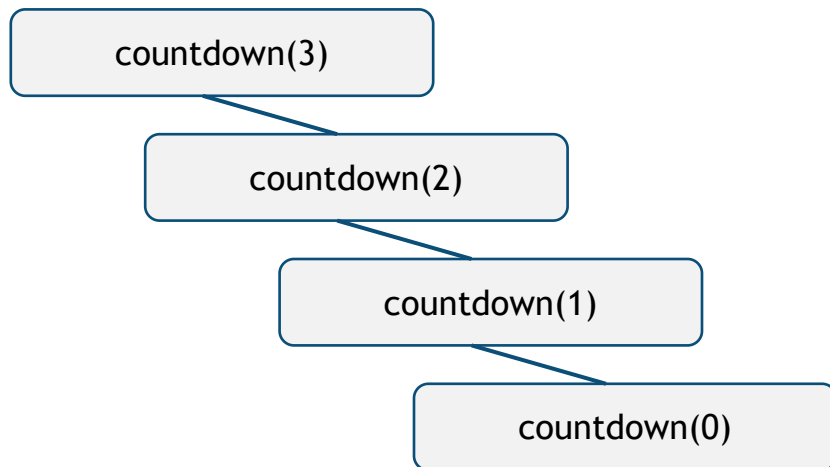
- The function prints a number and calls itself with a smaller number
- **Base case:** when n reaches 0
- **Recursive case:** print n and continue with $n - 1$

```
def countdown(n):  
    if n == 0:  
        print("Go!")  
    else:  
        print(n)  
        countdown(n - 1)  
  
countdown(3)
```

Each call receives a different n

Tracing Recursive Calls

- A recursive call pauses the current function call
- Execution enters the next call
- When the last call finishes, execution returns step by step.



Output:

3

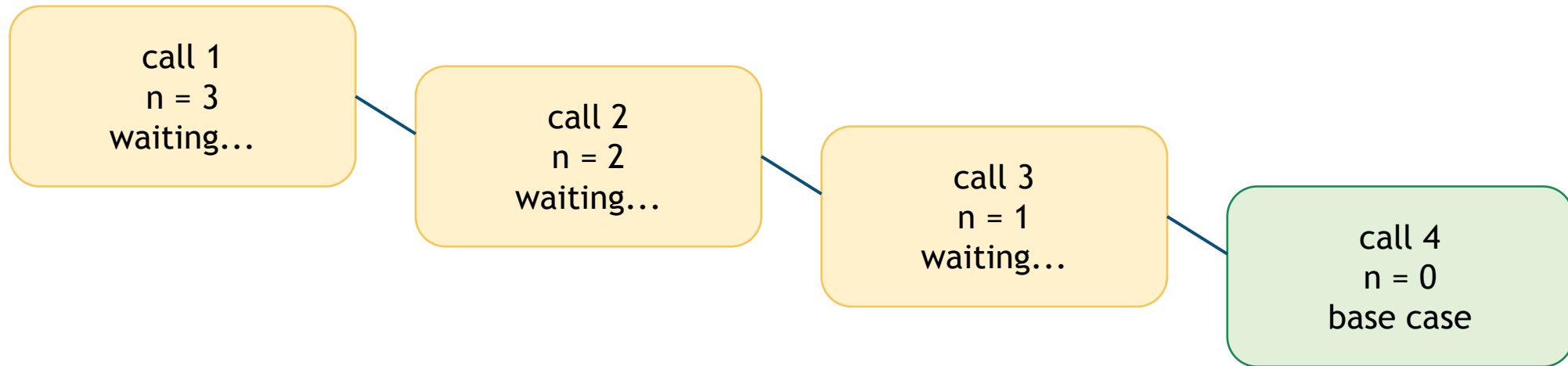
2

1

Go!

Each Call Has Its Own Local State

- Each recursive call is a separate function call
- Parameters and local variables belong to that call
- This is the same scope rule already seen for functions.



Returning Values Through Recursion

- Some recursive functions compute a result
- The base case returns a direct value
- Each pending call uses the value returned by the smaller call.

```
def factorial(n):  
    if n <= 1:  
        return 1  
  
    return n * factorial(n - 1)  
  
print(factorial(4))  # 24
```

return sends the result back to the caller.

Two Pattern: Direct Recursive Return

Direct Recursive Return

- The current call waits for the next call
- When the value comes back, it is **returned unchanged**
- **No extra computation** is pending after the recursive call.

```
def f(n):  
    if n == 0:  
        return 0  
    return f(n - 1)
```

The frame is waiting, but only to pass the value back.

f(3)
waiting for f(2), then return result

f(2)
waiting for f(1), then return result

f(1)
waiting for f(0), then return result

f(0)
base case: return 0

Return path: 0 → 0 → 0 → 0

Each frame returns exactly the value received.

Two Pattern: Using the Returned Value

Using the Returned Value

- The current call waits for the recursive result
- Its local variables must remain available
- After the next call returns, **a pending operation is completed.**

```
def f(n):  
    if n == 0:  
        return 0  
    return n + f(n - 1)
```

The answer is built while returning.

f(3)
waiting to compute: $3 + \text{result}$

f(2)
waiting to compute: $2 + \text{result}$

f(1)
waiting to compute: $1 + \text{result}$

f(0)
base case: return 0

$$\begin{aligned}f(1) &= 1 + 0 = 1 \\f(2) &= 2 + 1 = 3 \\f(3) &= 3 + 3 = 6\end{aligned}$$

Each frame uses its own n after the recursive call returns.

Designing a Recursive Function

- Ask three questions:
 - What is the simplest input I can solve directly?
 - How can I reduce the input to a smaller instance?
 - How do I combine the smaller answer with the current step?
- Then test very small cases first.



**Base case + smaller input +
combination**

Classic Recursive Definition: Fibonacci

- Fibonacci is defined using previous Fibonacci values
- **Base cases:** $F(0) = 0$ and $F(1) = 1$
- **Recursive case:** $F(n) = F(n-1) + F(n-2)$
- It shows that one call can generate two recursive calls.

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

Useful to understand the structure; not the most efficient way to compute Fibonacci sequences.

Recursion on Strings

- A string can be viewed as:
 - first character + remaining string
- Example task: count how many times a character appears
- Base case: empty string.

```
def count_char(text, target):  
    if text == "":  
        return 0  
  
    if text[0] == target:  
        return 1 + count_char(text[1:], target)  
    else:  
        return count_char(text[1:], target)
```

text[1:] is a smaller string.

Building a New Result Recursively

- Recursion is not only for numbers
- We can build a new string step by step
- At each step: keep or skip the first character.

```
def remove_spaces(text):  
    if text == "":  
        return ""  
  
    if text[0] == " ":  
        return remove_spaces(text[1:])  
  
    return text[0] + remove_spaces(text[1:])
```

Recursion on Lists

- A list can be viewed similarly:
 - first element + remaining list
- Example task: compute the sum of a list.
- Base case: empty list has sum 0.

```
def sum_list(values):  
    if len(values) == 0:  
        return 0  
  
    return values[0] + sum_list(values[1:])
```

values[1:] is a smaller list.

When Recursion Becomes Natural

It is proved that any recursive algorithm can be transformed in an iterative one using loops and possibly other auxiliary data structures.
Specifically in Python, a well-written loop is more efficient.

Why recursion, then?

- Fractals, trees, graphs, or nested formats
 - Nested structures repeat the same pattern at different levels
 - Example: an employee may manage a team of employees
 - The same procedure can be applied to each team member
- “*Divide and Conquer*” algorithms
 - Merge Sort, Quick Sort...
 - finding an optimal path in graphs (such as computing connected components using Depth-First Search)
- Elegance and readability → **Maintainability** of code

```
company = {  
    "name": "Director",  
    "team": [  
        {"name": "Manager A",  
         "team": [  
             {"name": "Alice", "team": []}  
         ]},  
        {"name": "Manager B", "team": []}  
    ]  
}
```

One employee record contains more employee records.

Common Recursion Errors

- **Missing base case** → the function never stops
- Recursive call does **not reduce the problem** → the base case is never reached
- Wrong return value → the call happens, but the result is lost
- Too many recursive calls → possible RecursionError in Python.

```
def bad_countdown(n):  
    print(n)  
    bad_countdown(n)  
  
# no progress toward 0
```

Recursion vs Iteration

Iteration

- Often simpler for linear repetitions
- Usually more efficient in Python
(no Tail Call Optimization (TCO) in Python)
- Good when the number of steps is clear.

Recursion

- Natural for hierarchical data
- Natural for recursive definitions
- Requires careful base case and reduction.

Recursion is not automatically better; use it when it makes the structure clearer.

Checklist Before Writing Recursive Code

- Can I clearly identify a smaller problem of the same type?
- Can I solve a very small case directly?
- Does every recursive call move toward that small case?
- Do I return the recursive result, or combine it correctly?
- Would a loop be simpler here?

Start with the base case and test small inputs.

Key Takeaways

- Recursion means solving a problem by calling the same function on a smaller problem
- Every recursive function needs a **base case** and a **recursive case**
- Recursive calls create multiple active function calls with their own local state
- Recursion is especially natural for nested or hierarchical structures
- In Python, prefer recursion when it improves clarity; otherwise, iteration may be simpler.