Python Recursion

Introduction

Recursion is a programming technique where a function **calls itself** to solve a problem.

At first, this can sound confusing:

> How can a function call itself without running forever?

The answer is that a recursive function needs a **base case** that tells it when to stop.

A simple example:

```
def countdown(number):
    if number == 0:
        return

    print(number)
    countdown(number - 1)

countdown(5)
```

Output:

```
5
4
3
2
1
```

The function keeps calling itself with a smaller value until it reaches:

`number == 0`

That is the **base case**.

---

1. What Is Recursion?

Recursion happens when a function calls itself.

Basic structure:

```
def function():
    function()
```

But this alone is dangerous because the function would keep calling itself forever.

A useful recursive function normally contains two important parts:

```
Recursive Function
□
□□□ Base Case
□   □□□ Stops the recursion
□
□□□ Recursive Case
    □□□ Calls the function again with a smaller/simpler problem
```

---

2. The Two Essential Parts

Every recursive solution should make you think about two questions:

1. When should the function stop?

This is the **base case**.

2. How does the problem become smaller?

This is the **recursive case**.

Example:

```
def countdown(number):
    if number == 0:
        return

    print(number)
    countdown(number - 1)
```

Base case:

```
if number == 0:
    return
```

Recursive case:

```
countdown(number - 1)
```

---

3. First Simple Example

```
def countdown(number):
```

```
    if number == 0:
        return

    print(number)
    countdown(number - 1)

countdown(3)
```

Output:

```
3
2
1
```

Let's follow it:

```
countdown(3)
    
countdown(2)
    
countdown(1)
    
countdown(0)
    
STOP
```

The function eventually reaches the base case.

---

4. Why Does Recursion Stop?

The recursive call must move toward the base case.

In:

```
countdown(number - 1)
```

the value gets smaller:

```
3 → 2 → 1 → 0
```

At `0`, the function stops.

This gives us a very important rule:

> A recursive function must make progress toward its stopping condition.

---

5. What Happens Without a Base Case?

Consider:

```
def hello():
    print("Hello")
    hello()

hello()
```

The function keeps calling itself:

```
hello()
 →
hello()
```

```
 
hello()
 
hello()
 
...
```

Eventually Python stops the program with a:

```
RecursionError
```

This is called **infinite recursion**.

---

6. The Call Stack

To understand recursion, you need to understand the **call stack**.

When a function is called, Python keeps track of that function call.

For example:

```
def countdown(number):
    if number == 0:
        return

    print(number)
    countdown(number - 1)

countdown(3)
```

The calls build up like this:

```
countdown(3)
countdown(2)
countdown(1)
countdown(0)
```

When `countdown(0)` finishes, Python returns to the previous call.

Then:

```
countdown(0) finishes
      ↓
countdown(1) finishes
      ↓
countdown(2) finishes
      ↓
countdown(3) finishes
```

This is why recursion has two phases:

```
Going deeper
    ↓
Base case
    ↓
Returning back
```

---

7. Recursion Flow

Consider:

```
def countdown(number):
    if number == 0:
        return
```

```
    print("Start:", number)

    countdown(number - 1)

    print("End:", number)

countdown(3)
```

Output:

```
Start: 3
Start: 2
Start: 1
End: 1
End: 2
End: 3
```

Notice something important.

The first `print()` happens while recursion is going deeper.

The second `print()` happens while recursion is returning.

---

8. Visualizing the Call Stack

For:

```
countdown(3)
```

Python creates calls:

```
┌──────────────────────┐
│ countdown(1)         │
└──────────────────────┘
┌──────────────────────┐
│ countdown(2)         │
└──────────────────────┘
┌──────────────────────┐
│ countdown(3)         │
└──────────────────────┘
```

Then `countdown(1)` reaches the base case.

The calls are removed one by one.

```
countdown(1) → removed
countdown(2) → removed
countdown(3) → removed
```

The stack is therefore closely connected to recursion.

---

9. Recursive Factorial

One of the classic recursion examples is **factorial**.

The factorial of a number is:

`5! = 5 × 4 × 3 × 2 × 1`

So:

`5! = 120`

Mathematically:

`n! = n × (n - 1)!`

This definition naturally describes recursion.

---

10. Factorial Using Recursion

```
def factorial(n):
    if n == 0:
        return 1

    return n * factorial(n - 1)

print(factorial(5))
```

Output:

```
120
```

The base case is:

```
if n == 0:
    return 1
```

The recursive case is:

```
return n * factorial(n - 1)
```

---

11. Understanding Factorial Step by Step

When we call:

`factorial(5)`

Python evaluates:

`5 × factorial(4)`

Then:

`5 × 4 × factorial(3)`

Then:

`5 × 4 × 3 × factorial(2)`

Then:

`5 × 4 × 3 × 2 × factorial(1)`

Then:

`5 × 4 × 3 × 2 × 1 × factorial(0)`

The base case gives:

`factorial(0) = 1`

Now the results return:

`1`
`□`
`1 × 1 = 1`
`□`
`2 × 1 = 2`

```
3 × 2 = 6
4 × 6 = 24
5 × 24 = 120
```

---

12. Factorial With a Loop

The same problem can be solved without recursion:

```
def factorial(n):
    result = 1

    for number in range(1, n + 1):
        result *= number

    return result

print(factorial(5))
```

Output:

```
120
```

Both approaches work.

This leads to an important question:

> Should every repetitive problem use recursion?

No.

Recursion is useful when the problem itself has a recursive structure.

---

13. Recursion vs Loops

Loop

```
for number in range(5, 0, -1):
    print(number)
```

Recursion

```
def countdown(number):
    if number == 0:
        return

    print(number)
    countdown(number - 1)
```

A loop is often simpler for straightforward repetition.

Recursion becomes especially useful when dealing with **nested or hierarchical structures**.

---

14. Recursive Sum

Suppose we want to calculate:

```
1 + 2 + 3 + 4 + 5
```

We can define:

`sum(5) = 5 + sum(4)`

Then:

`sum(4) = 4 + sum(3)`

And so on.

Python:

```
def recursive_sum(n):
    if n == 0:
        return 0

    return n + recursive_sum(n - 1)

print(recursive_sum(5))
```

Output:

```
15
```

---

15. Recursive Sum Flow

```
recursive_sum(5)
        ↓
5 + recursive_sum(4)
        ↓
5 + 4 + recursive_sum(3)
```

```
      ↓
5 + 4 + 3 + recursive_sum(2)
      ↓
5 + 4 + 3 + 2 + recursive_sum(1)
      ↓
5 + 4 + 3 + 2 + 1 + recursive_sum(0)
      ↓
15
```

The base case:

```
n == 0
```

returns:

```
0
```

---

16. Recursively Counting Down

A very simple use of recursion:

```
def count_down(n):
    if n < 1:
        return

    print(n)
    count_down(n - 1)

count_down(5)
```

Output:

```
5
4
3
2
1
```

---

17. Recursively Counting Up

We can also print while returning:

```
def count_up(n):
    if n == 0:
        return

    count_up(n - 1)
    print(n)

count_up(5)
```

Output:

```
1
2
3
4
5
```

Why?

The function first reaches:

```
count_up(0)
```

Then the calls return:

```
1
2
3
4
5
```

---

18. Recursively Printing a String

We can process characters recursively.

```
def print_characters(text, index=0):
    if index == len(text):
        return

    print(text[index])
    print_characters(text, index + 1)

print_characters("Python")
```

Output:

```
P
y
t
h
o
n
```

The index moves toward the end of the string.

---

19. Recursive String Reversal

We can reverse a string recursively.

```
def reverse_text(text):
    if len(text) <= 1:
        return text

    return reverse_text(text[1:]) + text[0]

print(reverse_text("Python"))
```

Output:

```
nohtyP
```

The idea is:

```
Python
→
ython + P
→
thon + y + P
→
hon + t + y + P
...
```

---

20. Recursive List Sum

We can recursively calculate the sum of a list.

```
def list_sum(numbers):
    if not numbers:
        return 0

    return numbers[0] + list_sum(numbers[1:])

numbers = [10, 20, 30]

print(list_sum(numbers))
```

Output:

```
60
```

The base case is:

```
if not numbers:
    return 0
```

When the list becomes empty, recursion stops.

---

21. Recursion With Lists

Consider:

```
numbers = [10, 20, 30, 40]
```

The recursive process becomes:

```
[10, 20, 30, 40]
 
```

```
10 + [20, 30, 40]
↓
10 + 20 + [30, 40]
↓
10 + 20 + 30 + [40]
↓
10 + 20 + 30 + 40 + []
↓
60
```

This is useful for understanding how recursion reduces a problem into smaller versions of itself.

---

22. Searching Recursively

Suppose we want to check whether a value exists in a list.

```
def contains(numbers, target):
    if not numbers:
        return False

    if numbers[0] == target:
        return True

    return contains(numbers[1:], target)

numbers = [10, 20, 30, 40]

print(contains(numbers, 30))
```

Output:

```
True
```

The function checks one item and then asks the smaller list to solve the remaining problem.

---

23. The Recursive Problem-Solving Pattern

A useful pattern is:

```
1. Solve the smallest possible case.
2. Identify the base case.
3. Reduce the problem.
4. Call the function on the smaller problem.
5. Combine the result if necessary.
```

For example:

```
def recursive_sum(numbers):
    if not numbers:
        return 0

    return numbers[0] + recursive_sum(numbers[1:])
```

Think:

```
Current problem
      
Take one piece
      
Solve the smaller problem
      
Combine the answer
```

---

24. Fibonacci Sequence

Another classic recursion example is Fibonacci.

The sequence begins:

`0, 1, 1, 2, 3, 5, 8, 13...`

Each number is the sum of the previous two.

Mathematically:

`F(n) = F(n - 1) + F(n - 2)`

with base cases:

`F(0) = 0`
`F(1) = 1`

---

25. Recursive Fibonacci

```
def fibonacci(n):
    if n == 0:
        return 0

    if n == 1:
        return 1

    return fibonacci(n - 1) + fibonacci(n - 2)

print(fibonacci(6))
```

Output:

```
8
```

The recursive structure is:

```
fibonacci(n)
    □
fibonacci(n - 1)
    +
fibonacci(n - 2)
```

This creates multiple recursive branches.

---

26. Why Fibonacci Recursion Can Be Slow

Consider:

```
fibonacci(5)
```

It repeatedly calculates the same values.

For example:

```
fibonacci(5)
□□□ fibonacci(4)
□   □□□ fibonacci(3)
□   □□□ fibonacci(2)
□□□ fibonacci(3)
    □□□ fibonacci(2)
    □□□ fibonacci(1)
```

Notice that:

`fibonacci(3)`
`fibonacci(2)`

can be calculated multiple times.

This makes the simple recursive implementation inefficient for larger values.

---

27. Recursion With Memoization

We can store previously calculated results.

```
def fibonacci(n, memo=None):
    if memo is None:
        memo = {}

    if n in memo:
        return memo[n]

    if n == 0:
        return 0

    if n == 1:
        return 1

    memo[n] = fibonacci(n - 1, memo) + fibonacci(n - 2, memo)

    return memo[n]

print(fibonacci(40))
```

The dictionary remembers results that have already been calculated.

This technique is called **memoization**.

---

28. Recursion and Nested Data

Recursion becomes particularly useful when data has a nested structure.

Example:

```
data = [
    "Python",
    ["JavaScript", "React"],
    ["Flutter", ["Dart", "Firebase"]]
]
```

The structure contains lists inside lists.

A recursive function can process each level.

```
def print_items(data):
    for item in data:
        if isinstance(item, list):
            print_items(item)
        else:
            print(item)

print_items(data)
```

Output:

```
Python
JavaScript
React
Flutter
Dart
Firebase
```

The function does not need to know how deeply nested the structure is.

---

29. Why Recursion Works Well With Trees

Many real-world structures are naturally hierarchical:

```
Website
□□□ Resources
□   □□□ Python
□   □□□ JavaScript
□   □□□ DSA
□□□ Blog
□   □□□ Tutorials
□   □□□ Guides
□□□ Projects
```

A tree contains:

```
parent
  □
children
  □
more children
```

A recursive function can process each node and then recursively process its children.

This is one of the most important practical uses of recursion.

---

30. Recursive Tree Example

A simple nested dictionary:

```
website = {
    "name": "haas.dev",
    "children": [
        {
            "name": "Resources",
            "children": [
                {"name": "Python"},
                {"name": "JavaScript"}
            ]
        }
    ]
}
```

We can recursively print it:

```
def print_tree(node):
    print(node["name"])

    for child in node.get("children", []):
        print_tree(child)

print_tree(website)
```

Output:

```
haas.dev
Resources
Python
JavaScript
```

The function works at every level of the tree.

---

31. Practical haas.dev Example

Imagine the haas.dev resource system has categories:

```
resources = {
    "Programming": {
        "Python": {
            "Functions": {},
            "Data Structures": {}
        },
        "JavaScript": {
            "Functions": {},
            "Arrays": {}
        }
    }
}
```

A recursive function could explore every category:

```
def explore(data, level=0):
    for name, children in data.items():
        print("  " * level + name)

        if children:
            explore(children, level + 1)
```

```
explore(resources)
```

Output:

```
Programming
  Python
    Functions
    Data Structures
  JavaScript
    Functions
    Arrays
```

The same function can handle deeper levels without knowing their exact depth.

---

32. Base Case Design

Choosing the correct base case is one of the most important parts of recursion.

Suppose:

```
def countdown(n):
    if n == 0:
        return

    print(n)
    countdown(n - 1)
```

The base case is:

```
n == 0
```

Ask:

> What is the smallest version of this problem that I can solve immediately?

For factorial:

`0! = 1`

For list sum:

`empty list = 0`

For string reversal:

`one-character string = itself`

---

33. Recursive Case Design

After identifying the base case, ask:

> How can I make the current problem smaller?

Examples:

`countdown(n - 1)`

`recursive_sum(numbers[1:])`

`factorial(n - 1)`

`print_characters(text, index + 1)`

Each call moves closer to the base case.

---

34. Common Mistake: No Base Case

Incorrect:

```
def count(n):
    print(n)
    count(n - 1)
```

There is no stopping condition.

Correct:

```
def count(n):
    if n == 0:
        return

    print(n)
    count(n - 1)
```

---

35. Common Mistake: Wrong Base Case

Consider:

```
def count(n):
```

```
    if n == 10:
        return

    print(n)
    count(n - 1)
```

If we start with:

```
count(5)
```

the values become:

```
5
4
3
2
1
0
-1
-2
...
```

The function is moving away from `10`.

The base case exists, but the recursive call never reaches it.

The recursive step must move **toward** the base case.

---

36. Common Mistake: Forgetting to Change the Input

Incorrect:

```
def count(n):
    if n == 0:
        return

    print(n)
    count(n)
```

The argument never changes.

It stays:

```
n
n
n
n
...
```

Correct:

```
count(n - 1)
```

The problem becomes smaller.

---

37. Common Mistake: Returning Too Early

Consider:

```
def countdown(n):
    if n == 0:
        return
```

```
    return

    print(n)
    countdown(n - 1)
```

The function stops immediately after the first `return`.

The recursive call is never reached.

Remember that `return` exits the current function.

---

38. Recursion Depth

Python limits how deeply function calls can be nested.

You can see the approximate recursion limit with:

```
import sys

print(sys.getrecursionlimit())
```

A recursive function that goes too deep can cause:

```
RecursionError: maximum recursion depth exceeded
```

This is one reason recursion should not automatically be used for very large linear problems.

---

39. Recursion vs Iteration

Feature	Recursion	Iteration
Main mechanism	Function calls itself	Loop repeats
Uses call stack	Yes	Usually less stack usage
Good for trees	Yes	Possible but often less natural
Good for simple counting	Usually unnecessary	Often simpler
Can be elegant	Yes	Yes
Risk of recursion depth	Yes	No recursion-depth issue
Performance	Depends on problem	Often efficient
Best for	Naturally recursive structures	Straight repetition

---

40. When Should You Use Recursion?

Recursion is especially useful when the problem naturally contains smaller versions of itself.

Examples:

- tree traversal
- directory traversal
- nested data
- divide-and-conquer algorithms
- depth-first search
- backtracking
- some mathematical problems
- recursive data structures

---

41. When Should You Use a Loop?

A loop is often better when you simply need to repeat something.

For example:

```
for number in range(1, 101):
    print(number)
```

There is no strong reason to use recursion for this.

The loop is straightforward and avoids unnecessary function calls.

---

42. Recursion in Problem Solving

When facing a recursive problem, use this checklist:

Step 1: Identify the smallest problem

What input can be solved immediately?

Step 2: Create the base case

```
if smallest_case:
    return result
```

Step 3: Reduce the problem

Make the input smaller or simpler.

Step 4: Call the function again

```
function(smaller_problem)
```

Step 5: Combine results

If necessary:

```
return current_value + recursive_result
```

---

43. Mini Project: Recursive Folder Explorer

Imagine a folder structure:

```
folders = {
    "project": {
        "src": {
            "components": {},
            "pages": {}
        },
        "assets": {},
```

```
        "tests": {}
    }
}
```

Create a recursive explorer:

```
def explore_folder(folder, level=0):
    for name, contents in folder.items():
        print("  " * level + name)

        if contents:
            explore_folder(contents, level + 1)

explore_folder(folders)
```

Output:

```
project
  src
    components
    pages
  assets
  tests
```

This is a practical example because real file systems and hierarchical data naturally form tree structures.

---

44. Mini Project: Recursive Resource Counter

Suppose categories contain nested resources:

```
resources = {
    "Python": {
```

```
        "Functions": 5,
        "Data Structures": 8
    },
    "JavaScript": {
        "Functions": 6,
        "Arrays": 4
    }
}
```

We can recursively count all resource values:

```
def count_resources(data):
    total = 0

    for value in data.values():
        if isinstance(value, dict):
            total += count_resources(value)
        else:
            total += value

    return total

print(count_resources(resources))
```

Output:

```
23
```

The function can handle nested dictionaries without knowing their depth.

---

45. 5 Minute Challenge

Write a recursive function:

`sum_numbers(numbers)`

that calculates the sum of:

`[5, 10, 15, 20]`

Expected result:

`50`

Requirements:

- Do not use `sum()`.
- Use a base case.
- Remove or skip one item at each recursive call.
- Return the final result.

Then try the same problem using a loop and compare the two solutions.

---

46. Exercises

Exercise 1

Create a recursive countdown function:

`countdown(5)`

Expected:

```
5
4
3
2
1
```

Exercise 2

Create:

`factorial(5)`

without using a loop.

Expected:

`120`

Exercise 3

Create:

`recursive_sum(10)`

that returns:

`55`

Exercise 4

Create a recursive function that counts the characters in a string without using `len()`.

Example:

```
count_characters("Python")
```

Expected:

```
6
```

Exercise 5

Create a recursive function that reverses:

```
"Python"
```

Expected:

```
"nohtyP"
```

Exercise 6

Create a recursive function that searches for a target value in a list.

Example:

```
contains([10, 20, 30, 40], 30)
```

Expected:

`True`

---

47. Mini Quiz

Question 1

What is recursion?

A. A loop inside a loop
B. A function calling itself
C. A variable calling itself
D. A function without arguments

Answer: B

Question 2

What is required to stop recursion?

A. `break`
B. `continue`
C. A base case
D. `pass`

Answer: C

Question 3

What happens if recursion never reaches its base case?

A. Python automatically fixes it
B. The function returns `None`
C. It can cause `RecursionError`
D. Nothing happens

Answer: C

Question 4

Which statement should generally be true about the recursive call?

A. It should move toward the base case
B. It should always increase the input
C. It should never change the input
D. It should always use a loop

Answer: A

Question 5

Which structure is naturally suited to recursion?

A. Simple counting
B. Trees and nested structures
C. Printing one message

D. Basic variable assignment

Answer: B

Question 6

What does the call stack keep track of?

A. Only variables
B. Active function calls
C. Only loops
D. Imported modules

Answer: B

---

48. Interview Questions

Beginner

1. What is recursion?
2. What is a base case?
3. What is a recursive case?
4. Why does recursion need a stopping condition?
5. What happens when a recursive function has no base case?
6. What is the call stack?

Intermediate

7. What is the difference between recursion and iteration?

8. When is recursion more appropriate than a loop?
9. What is infinite recursion?
10. What causes `RecursionError`?
11. Explain recursive factorial.
12. Explain recursive Fibonacci.
13. Why is naive recursive Fibonacci inefficient?
14. What is memoization?
15. Why are trees well suited to recursive algorithms?
16. What is the relationship between recursion and the call stack?

---

49. Cheat Sheet

Basic Structure

```
def function(value):

    if base_case:
        return result

    return function(smaller_value)
```

Countdown

```
def countdown(n):
    if n == 0:
        return

    print(n)
    countdown(n - 1)
```

Factorial

```
def factorial(n):
    if n == 0:
        return 1

    return n * factorial(n - 1)
```

Recursive Sum

```
def recursive_sum(n):
    if n == 0:
        return 0

    return n + recursive_sum(n - 1)
```

List Sum

```
def list_sum(numbers):
    if not numbers:
        return 0

    return numbers[0] + list_sum(numbers[1:])
```

Nested Data

```
def explore(data):
    for item in data:
        if isinstance(item, list):
            explore(item)
```

Remember

```
Recursion = Function calls itself

Base case
    
Stops recursion
```

Recursive case

Makes problem smaller

Call stack

Keeps track of active calls

---

50. Key Takeaways

- **Recursion** is when a function calls itself.
- Every recursive solution needs a **base case**.
- The recursive case should move toward that base case.
- Each recursive call creates another function call on the call stack.
- Once the base case is reached, the calls return one by one.
- Recursion is useful for problems that naturally contain smaller versions of themselves.
- Trees, nested data, directory structures, and some algorithms are common recursion use cases.
- Simple repetition is often easier with loops.
- Recursive Fibonacci demonstrates how recursion can become inefficient when the same work is repeated.
- **Memoization** can store previously calculated results and avoid repeated work.
- Understanding recursion is important for later topics such as **trees, graph traversal, backtracking, divide and conquer, and dynamic programming**.

Visit haas.dev for more resources and guides.

Website: https://dev-roast-app.vercel.app
Resources: https://dev-roast-app.vercel.app/resources