



Guía Completa sobre Recursividad en Python

💡 ¿Qué es la recursividad?

La **recursividad** es una técnica de programación en la cual una **función se llama a sí misma** para resolver un problema. Cada llamada recursiva resuelve una **parte del problema**, y el conjunto de llamadas sucesivas permiten alcanzar la solución final.

🧠 ¿Cómo funciona?

Una función recursiva tiene dos componentes clave:

1. Caso base (o condición de corte):

Es la condición que **detiene la recursión**, evitando que el proceso se repita infinitamente.

2. Paso recursivo:

Es la parte donde la función **se llama a sí misma** con un problema más pequeño o una versión modificada del mismo.

💡 Ejemplo simple: cuenta regresiva

```
def cuenta_regresiva(n):
    if n == 0:
        print("¡Despegue!")
    else:
        print(n)
        cuenta_regresiva(n - 1)

cuenta_regresiva(5)
```

Salida:

```
5
4
3
2
1
¡Despegue!
```

Diagrama conceptual de la recursividad

Imaginemos la función `cuenta_regresiva(3)` como una pila de llamadas:

```
cuenta_regresiva(3)
└ cuenta_regresiva(2)
    └ cuenta_regresiva(1)
        └ cuenta_regresiva(0) → imprime "¡Despegue!"
```

Cada llamada queda “en pausa” hasta que termina la más interna, y luego se deshacen en orden inverso.

Ejemplo clásico: factorial de un número

La función factorial se define matemáticamente así:

- $\text{factorial}(0) = 1$
- $\text{factorial}(n) = n \times \text{factorial}(n-1)$

Implementación en Python:

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

print(factorial(5)) # Resultado: 120
```

Comparación entre recursión y bucle

Con bucle:

```
def factorial_iterativo(n):
    resultado = 1
    for i in range(1, n + 1):
        resultado *= i
    return resultado
```

Con recursión:

Más elegante, pero puede consumir más memoria (por el uso de la pila de llamadas).

⚠ Consideraciones importantes



- La **recursión debe tener un caso base claro**, o se producirá un error de recursión infinita (`RecursionError`).
- Python tiene un **límite de profundidad recursiva** (por defecto es 1000 llamadas).

```
import sys
print(sys.getrecursionlimit()) # Generalmente 1000
```

🔍 Ejemplo útil: suma de una lista

```
def suma_lista(lista):
    if not lista:
        return 0
    else:
        return lista[0] + suma_lista(lista[1:])

print(suma_lista([1, 2, 3, 4])) # Resultado: 10
```

🌲 Ejemplo más complejo: Fibonacci

```
def fibonacci(n):
    if n == 0 or n == 1:
        return n
    else:
        return fibonacci(n - 1) + fibonacci(n - 2)

print(fibonacci(6)) # Resultado: 8
```

Nota: Esta versión no es eficiente para valores grandes. Se pueden aplicar técnicas como **memoización** para mejorarla.

💼 ¿Cuándo usar recursividad?

- Cuando el problema tiene una **estructura repetitiva anidada** o se puede **dividir en subproblemas similares**.
 - Es útil en árboles, estructuras jerárquicas, algoritmos de búsqueda, y más.
-

❖ ¿Qué es el stack o pila de llamadas?

Cuando se llama a una función en Python, el programa guarda la **información de esa llamada en una pila**, conocida como **call stack** (pila de llamadas).

Cada vez que una función **se llama a sí misma**, se agrega un nuevo **marco de ejecución** (stack frame) al tope de la pila. Cuando una función termina, ese marco se **elimina (pop)** de la pila.



💡 Imaginá la pila como una torre de cajas:

- Cada caja representa una llamada a la función.
- Cuando la función se llama a sí misma, se **apila una nueva caja**.
- Cuando la llamada termina, se **quita la caja superior**.

⚠ ¿Por qué puede llenarse el stack con la recursividad?

La recursividad **agrega una nueva capa al stack** por cada llamada, sin reutilizar las anteriores.

Si hay muchas llamadas recursivas, la pila **crece demasiado** y se produce un error llamado:

RecursionError: maximum recursion depth exceeded

En cambio, en un **bucle for**, todo ocurre en el **mismo marco de ejecución**, sin usar más espacio en la pila. Por eso un bucle puede repetirse millones de veces sin problema.

⌚ Ejemplo gráfico

🌀 Recursión (stack se llena):

```
factorial(3)
└ factorial(2)
    └ factorial(1)
        └ factorial(0)
```

Cada una de estas funciones **espera que termine la siguiente** para poder continuar.

⌚ Bucle for (misma pila, sin crecer):

```
def factorial_iterativo(n):
    resultado = 1
    for i in range(1, n + 1):
        resultado *= i
```

El ciclo se ejecuta en **una sola función activa**, sin agregar marcos a la pila.

En el siguiente link encontrarás un artículo relacionado con la temática.

<https://medium.com/swlh/visualizing-recursion-6a81d50d6c41>