

# Funciones en Python: De la Sintaxis a la Arquitectura

Dominando el pilar fundamental del código modular y eficiente.



Blandskron

# El Fundamento del Software Moderno: Abstracción y Reutilización

Más que un bloque de código, una función es la unidad fundamental de **abstracción**, **reutilización** y **diseño** en Python. Es un objeto de ‘primera clase’ que encapsula lógica y define ámbitos.

## Sin Funciones



### Código Duplicado



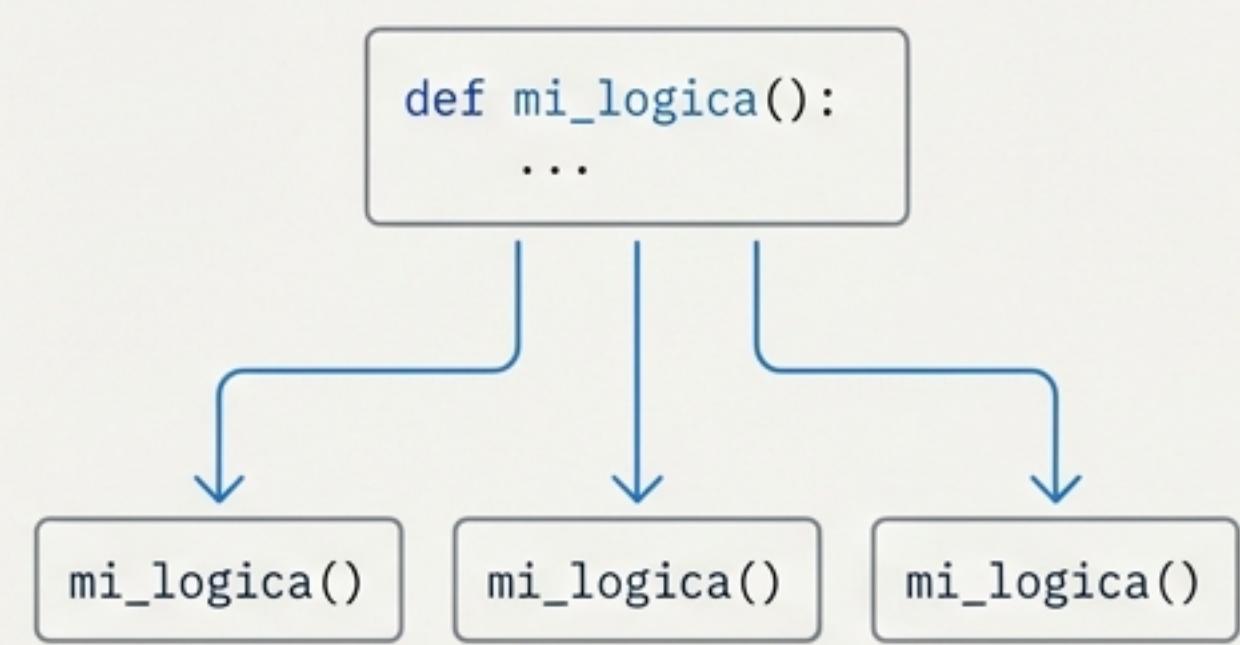
Incrementa la deuda técnica, propenso a errores, difícil de mantener.

El principio **DRY (Don't Repeat Yourself)** definir la lógica una vez para ser invocada infinitas veces, garantizando coherencia y facilitando el mantenimiento.

## Con Funciones



### Código Modular



Reutilizable, legible, fácil de probar y mantener.

# Definición vs. Invocación: El Plano Frente a la Ejecución

## La Definición (`def`) - "El Plano"

```
def mi_funcion():
    ...
```

La sentencia `def` es ejecutable. No corre el código, sino que crea el objeto-función.

## Pasos del Intérprete

1.  **Compila** el cuerpo de la función a bytecode.
2.  **Crea** un nuevo objeto de función que encapsula el bytecode.
3.  **Asigna** este objeto a un nombre en el *namespace* actual.



## La Invocación (`()`) - "La Ejecución"

```
resultado = mi_funcion()
```

El operador de llamada `()` activa la ejecución de la lógica encapsulada.

## Mecánica Interna

1.  **Crea** un nuevo marco de pila (stack frame) en la memoria.
2.  **Transfiere** el control del programa a la función.
3.  **Ejecuta** el código hasta encontrar `return` o el final del bloque. El marco de pila se destruye al terminar.

# El Flujo de Datos: Parámetros y Argumentos

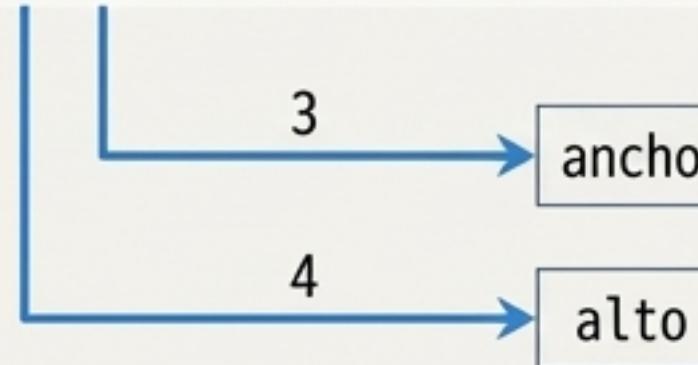
Una función recibe datos de entrada ('argumentos') que se asignan a sus variables internas ('parámetros'). Python ofrece una sintaxis flexible para este mapeo.

## 1. Posicionales

El orden importa. El argumento se asigna al parámetro en la misma posición.

```
def area_rectangulo(ancho, alto):
    return ancho * alto

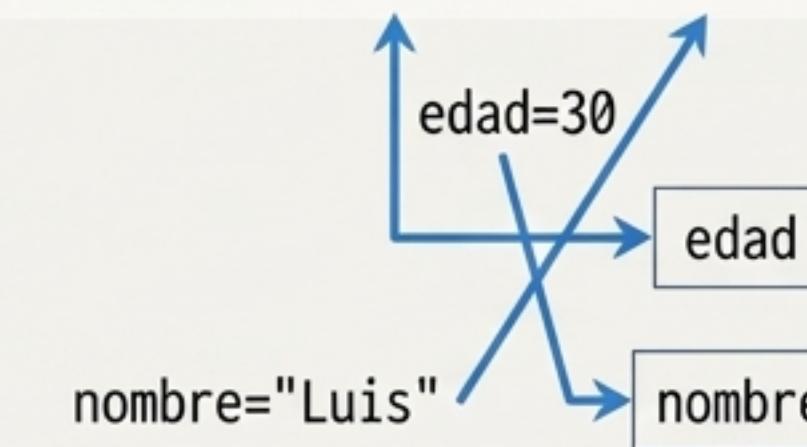
area_rectangulo(3, 4) # ancho=3, alto=4
```



## 2. Por Nombre (Keyword)

El orden no importa. La asignación es explícita.

```
def crear_usuario(nombre, edad): ...
crear_usuario(edad=30, nombre="Luis")
```



## 3. Por Defecto

Proporcionan un valor de respaldo si no se pasa un argumento.

```
def saludar(nombre, saludo="Hola"): ...
print(saludar("María"))      # Usa "Hola"
print(saludar("Pedro", "Buenos días"))
# Usa "Buenos días"
```

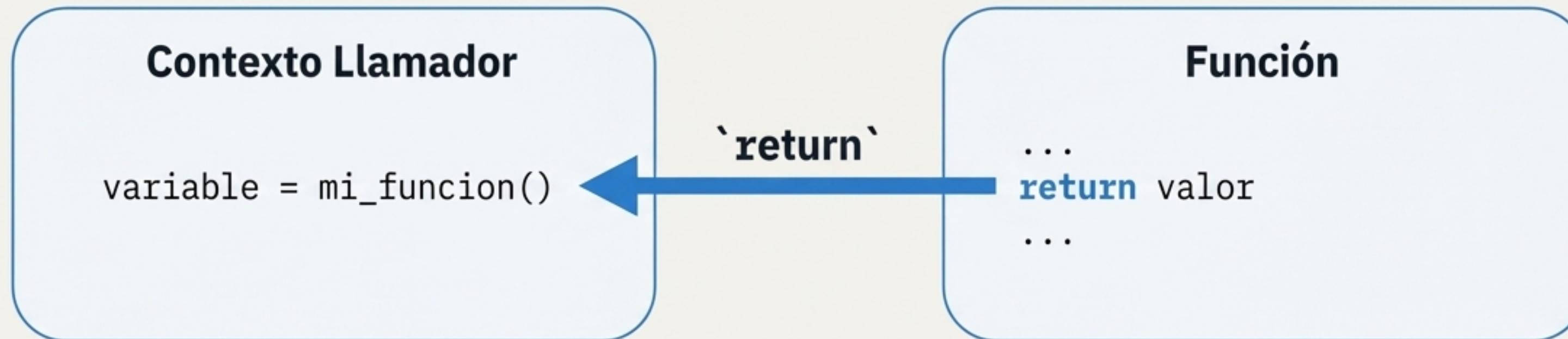
```
def saludar(nombre, saludo="Hola"): ...
```

Opcional

\*Nota de Diseño: Los parámetros con valores por defecto deben ir después de los posicionales.

# El Viaje de Vuelta: La Sentencia `return`

La sentencia `return` finaliza la ejecución de la función y devuelve un valor al contexto que la llamó.



## 1. Retorno Único

Devuelve un solo objeto de cualquier tipo.

```
def sumar(a, b):
    return a + b

resultado = sumar(5, 7) # resultado es 12
```

## 2. Retorno Múltiple

Python empaqueta automáticamente múltiples valores en una **tupla**.

```
def stats(nums):
    return sum(nums), max(nums)

total, mayor = stats([10, 20, 30]) # Desempaquetado elegante
```

## 3. Retorno Implícito

Si una función termina sin una sentencia `return`, devuelve `None` por defecto.

```
def saludo_simple():
    print("Hola")

valor = saludo_simple() # valor es None
```

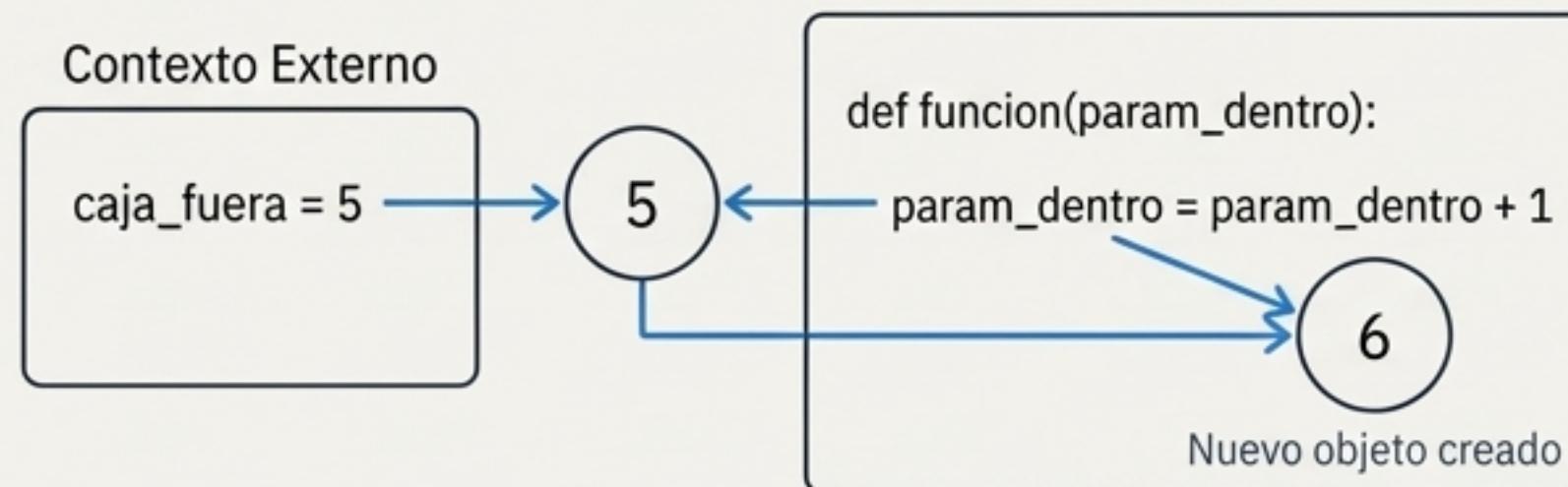
# El Modelo de Memoria: 'Llamada por Referencia de Objeto'

En Python, las variables no contienen valores, son **nombres que apuntan a objetos en memoria**. Al llamar a una función, se copia la **referencia** al objeto, no el objeto en sí. El comportamiento observable depende de la **mutabilidad** del objeto.

## Objetos Inmutables (int, str, tuple)

Al intentar modificar el valor (e.g., `x = x + 1`), Python crea un **nuevo objeto** y reasigna el nombre local a él. El objeto original permanece intacto.

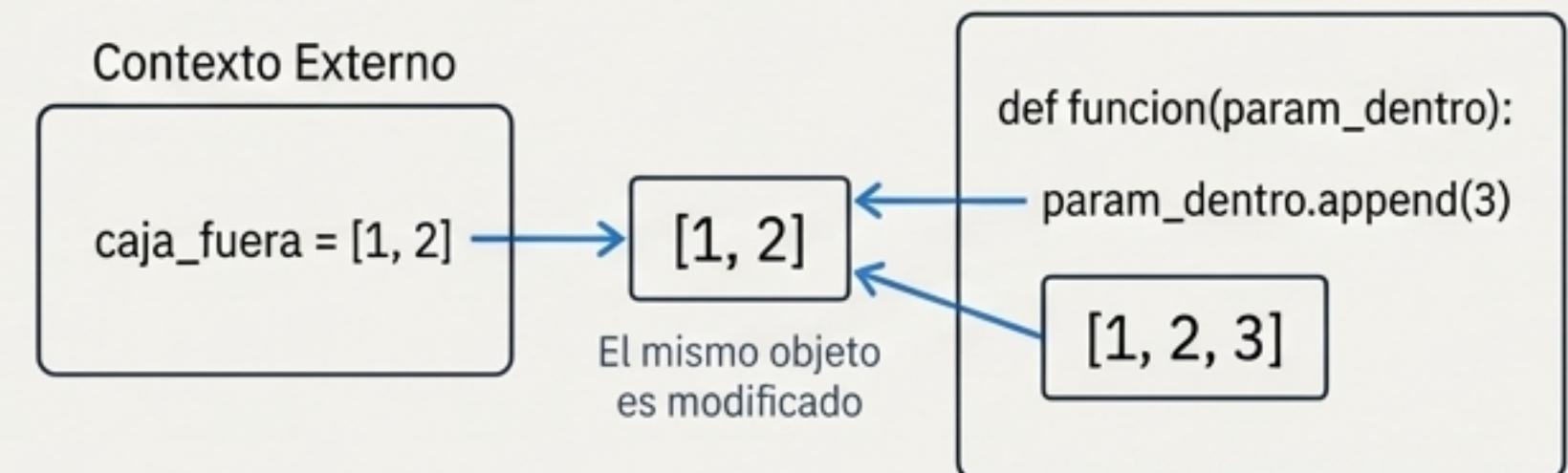
Resultado: Simula el "paso por valor". La variable externa **no cambia**.



## Objetos Mutables (list, dict)

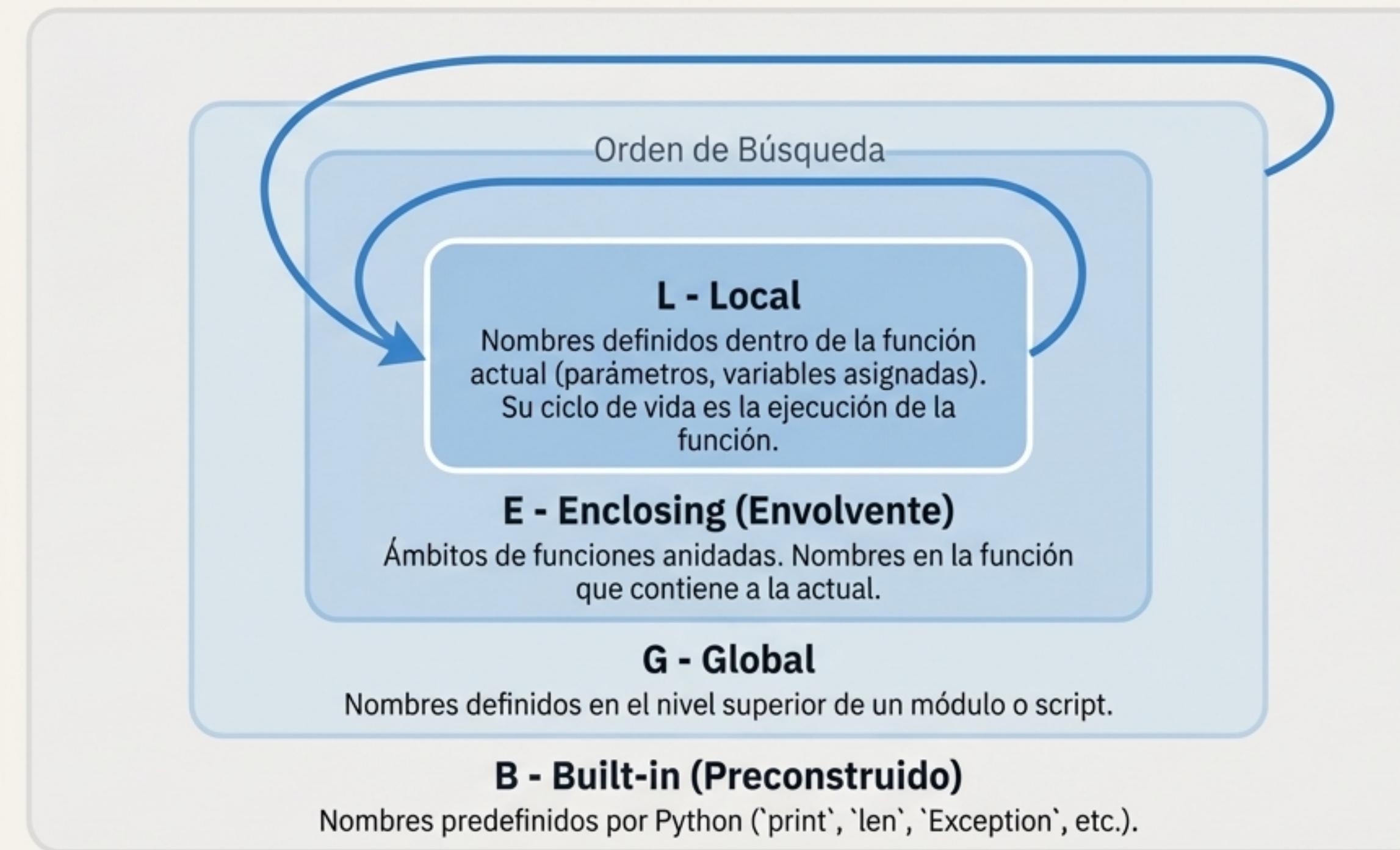
Al modificar el objeto *in-situ* (e.g., `lista.append(3)`), se modifica el **mismo objeto** al que apunta la variable externa.

Resultado: Simula el "paso por referencia". La variable externa **sí cambia**.



# El Universo de las Variables: La Regla de Resolución LEGB

Cuando se usa un nombre de variable, Python lo busca en una secuencia estricta de ámbitos (scopes). Si no lo encuentra, lanza un `NameError`.



# Global vs. Local: Aislamiento vs. Acoplamiento

## Concepto de "Sombreado" (Shadowing)

Una variable local con el mismo nombre que una global la "oculta" dentro de la función. Python siempre prioriza el ámbito más interno (Local).

## El Anti-Patrón: Modificar Variables Globales

Para modificar una global, se debe usar la palabra clave `global`, pero es una práctica desaconsejada.

### Problemas Arquitectónicos:

- Acoplamiento Fuerte:** La función depende de un estado externo, lo que dificulta su reutilización.
- Dificultad en Pruebas (Testing):** El estado compartido crea "efectos fantasma" entre pruebas.
- Problemas de Concurrencia:** El acceso simultáneo en aplicaciones multihilo puede causar corrupción de datos.



### Mal Diseño - Alto Acoplamiento

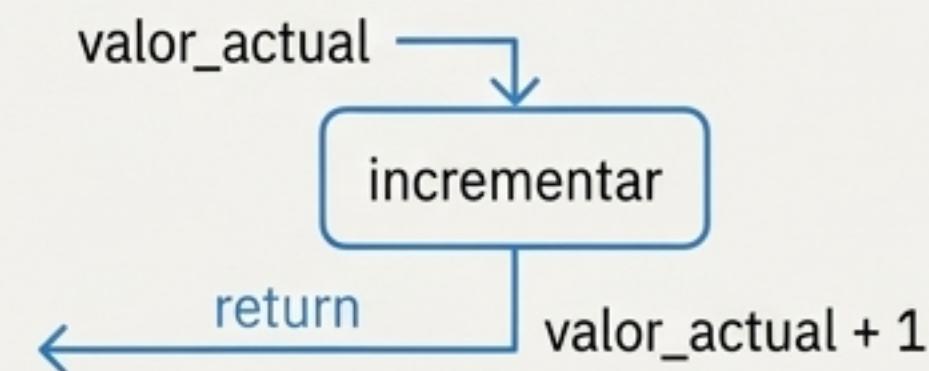
```
# Mala práctica
contador = 0
def incrementar():
    global contador
    contador += 1
```



### Buen Diseño - Función Pura

```
# Buena práctica
def incrementar(valor_actual):
    return valor_actual + 1

contador = 0
contador = incrementar(contador)
```



# Flexibilidad Infinita con `\*args` y `\*\*kwargs`

Para diseñar funciones que acepten un número arbitrario de argumentos.

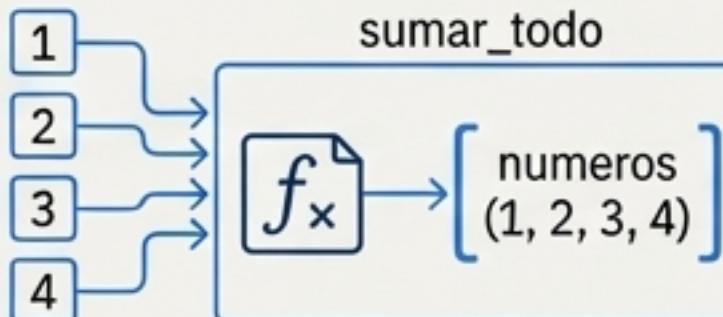
## \*args: Número Variable de Argumentos Posicionales

**Mecánica:** Agrupa todos los argumentos posicionales extra en una **tupla**.

**Caso de Uso:** Ideal para funciones de agregación o que aplican una operación a múltiples elementos.

```
def sumar_todo(*numeros): # numeros es una tupla
    total = 0
    for n in numeros:
        total += n
    return total
```

```
sumar_todo(1, 2, 3, 4) # Devuelve 10
```



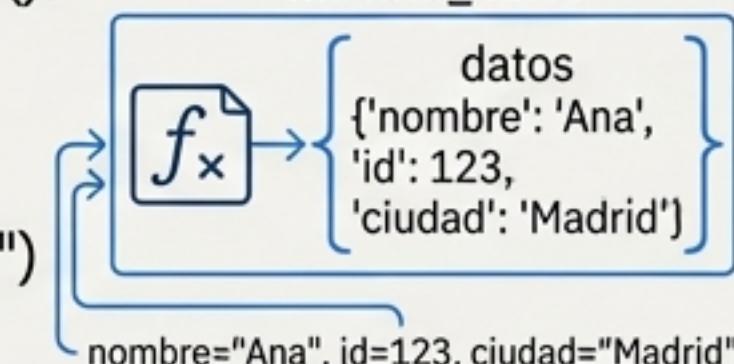
## \*\*kwargs: Número Variable de Argumentos por Nombre

**Mecánica:** Agrupa todos los argumentos por nombre extra en un **diccionario**.

**Caso de Uso:** Fundamental para decoradores o funciones que deben pasar configuraciones transparentemente.

```
def mostrar_datos(**datos): # datos es un diccionario
    for clave, valor in datos.items():
        print(f"{clave}: {valor}")

mostrar_datos(nombre="Ana",
              id=123, ciudad="Madrid")
```



# El Ecosistema de Funciones: Built-in, Usuario y Lambda

Característica	Funciones Built-in	Funciones de Usuario (def)	Funciones Anónimas (lambda)
Origen	Intérprete de Python	Código del programador	Expresión en tiempo de ejecución
Implementación	Generalmente en C (rápido)	En Python (bytecode)	En Python (bytecode)
Sintaxis	Nombre predefinido	Sentencia <code>def</code>	Palabra clave <code>lambda</code>
Capacidad	Tareas comunes y optimizadas	Lógica compleja, bloques de código, múltiples sentencias	<b>Una sola expresión</b> , no sentencias
Uso Ideal	Operaciones de alto rendimiento (e.g., <code>sum()</code> , <code>sorted()</code> )	Lógica de negocio, algoritmos, estructuración de código	Callbacks, claves de ordenamiento, funciones de orden superior ( <code>map</code> , <code>filter</code> )

# Manual de Estilo: Escribiendo Funciones Robustas y Profesionales

Un buen diseño de funciones es la diferencia entre un script frágil y una aplicación robusta.

## Lista de Verificación de Calidad

### Responsabilidad Única

Una función debe hacer una sola cosa, y hacerla bien. Si su nombre requiere la conjunción 'y', probablemente deba dividirse.

### Documentación Clara (Docstrings)

Explica el \*qué hace la función, sus parámetros (`Args:`) y su valor de retorno (`Returns:`), siguiendo el estándar PEP 257.

### Pistas de Tipos (Type Hinting)

Aumenta la claridad y permite la validación estática del código. Hace las interfaces explícitas.

```
def suma(a: int, b: int) -> int:
```

### Evitar Efectos Secundarios (Inmutabilidad)

Prefiere retornar un nuevo objeto a modificar un parámetro mutable. Esto hace que las funciones sean predecibles y fáciles de probar.

```
# Preferir esto:  
nueva_lista = sorted(mi_lista)
```

```
# A esto:  
mi_lista.sort()
```

### Minimizar Dependencia Global

Pasa datos explícitamente a través de parámetros y resultados a través de `return`.

# Síntesis: De Script Lineal a Arquitectura Modular

**\*\*Problema\*\*:** Procesar una lista de datos de temperatura: filtrar valores, convertir unidades y formatear un reporte.

## Antes (Script “Spaghetti”)

Acoplamiento y Lógica Mezclada

```
# TODO: Refactorizar este código
datos = [20, 15, -5, 30.5, 'N/A']

reporte = "Reporte de Temperaturas:\n"

for temp in datos:
    if isinstance(temp, (int, float)) and temp > 0:
        # Conversión y formato en el mismo lugar
        temp_f = (temp * 9/5) + 32
        reporte += f"- {temp}°C -> {temp_f:.1f}°F\n"

print(reporte)
```



## Después (Arquitectura Funcional)

Responsabilidad Única y Reutilización

```
def limpiar_datos(lista_cruda: list) -> list:
    # ...lógica de limpieza...

def convertir_a_fahrenheit(celsius: float) -> float:
    return (celsius * 9/5) + 32

def formatear_reporte(datos_proc: list) -> str:
    # ...lógica de formateo...

# Orquestación clara
datos_crudos = [20, 15, -5, 30.5, 'N/A']
datos_limpios = limpiar_datos(datos_crudos)
print(formatear_reporte(datos_limpios))
```



# **Dominar las funciones es dominar la esencia de la programación en Python.**

Son el átomo del diseño de software: la clave para construir sistemas escalables, mantenibles y eficientes.



**Blandskron**