



## ✨ Funciones en Python ✨


### ✅ ¿Qué es una función?

Las funciones se presentan como una herramienta para transformar problemas complejos en soluciones claras y manejables. Una función es un bloque de código que se encarga de cumplir una tarea específica. Al encapsular la lógica necesaria para abordar un determinado problema, la función se comporta como una especie de “caja negra”: se le proporcionan ciertos datos de entrada y, sin revelar los detalles de su proceso interno, devuelve el resultado esperado.


Toda función se compone de tres elementos fundamentales:


 **Entrada** (Argumentos): Son los datos que se le suministran a la función para realizar su tarea. Estos pueden ser números, textos o incluso estructuras más complejas.


 **Proceso** (Cuerpo): Es el conjunto de instrucciones o cálculos que la función lleva a cabo sobre la información recibida para producir un resultado.


 **Salida** (Retorno): Este es el resultado final que se obtiene después de procesar la entrada. En algunos casos, las funciones se usan simplemente para ejecutar una acción sin generar un resultado explícito.

## + Ventajas de usar funciones

 **Modularidad**: Divide problemas grandes en partes pequeñas permitiendo abordar cada una de ellas de manera aislada, facilitando así la comprensión y el mantenimiento del código.

 **Reutilización**: Llama la misma función en varios lugares evitando duplicar esfuerzos y fomenta un estilo de programación más limpio y organizado.

 **Abstracción**: Oculta detalles internos, exponiendo solo la interfaz permitiendo trabajar con funciones sin necesidad de conocer cómo están implementadas internamente. Solo nos interesa qué hacen, no cómo lo hacen.


 **Mantenimiento**: Si en algún momento necesitás cambiar cómo funciona esa parte del programa, solo debés modificar el código dentro de esa función, sin tocar el resto del sistema. Esto es esencial cuando los proyectos crecen o cuando el trabajo se hace en equipo.

## Tipos de parámetros de función

### Parámetros por valor

La función recibe una **copia del valor original**. Es decir, cualquier cambio que se haga dentro de la función **no afecta la variable externa**.

Este comportamiento es típico con **tipos de datos inmutables** como `int`, `float`, `str` y `tuple`.


 Lo que sucede dentro de la función queda “encapsulado”: la variable original fuera de la función **no se entera** de los cambios.


```
def incrementar(x):  
    x += 1 # cambia la copia local  
    return x  
  
n = 5  
print(incrementar(n)) # → 6  
print(n)               # → 5 (sin modificar)
```

### Parámetros por referencia


En este caso, se pasa una **referencia al objeto original**, no una copia. Esto significa que **los cambios hechos dentro de la función sí afectan** a la variable externa.

 Este comportamiento es típico con **tipos de datos mutables** (`list`, `dict` o `set`)

 La función puede modificar directamente el contenido del objeto, porque ambas (la función y el programa principal) apuntan al **mismo espacio en memoria**.

 **Cuidado:** este comportamiento puede ser útil, pero también puede provocar errores si no se controla bien.

```
def agregar_elemento(lista, elemento):  
    lista.append(elemento) # modifica la lista original  
  
mi_lista = [1, 2, 3]  
agregar_elemento(mi_lista, 4)  
print(mi_lista) # → [1, 2, 3, 4]
```

 En Python, todos los parámetros se pasan como referencias a objetos, pero los tipos inmutables (`int`, `str`, `tuple`) se comportan como si fueran por valor.

## Funciones integradas vs definidas

Python ofrece dos grandes grupos de funciones que podés usar en tus programas:

### Funciones integradas (built-in)

Son funciones que ya vienen incluidas en el lenguaje. No necesitás definir las, solo llamarlas.

```
print("Hola")      # Muestra texto por pantalla
input("Nombre: ")  # Solicita datos al usuario
len("Python")      # Devuelve la longitud de una cadena
```

### Funciones definidas por el usuario

Son funciones que se pueden crear para **resolver una tarea específica** u **organizar mejor tu código**.

```
def salta_renglon():
    """Imprime una línea en blanco."""
    print()

def saludar(nombre):
    """Muestra un saludo personalizado."""
    print(f"Hola {nombre}, ¡bienvenido(a)!")
```

## Múltiples parámetros

Las funciones pueden recibir **más de un parámetro** para trabajar con varios datos al mismo tiempo. Esto permite comparar, combinar o realizar cálculos entre valores distintos.

```
def obtenerresto(a, b):
    """Retorna a % b."""
    return a % b


def obtenerresto_sin_mod(a, b):
    """Resto sin usar %: a - (b * (a // b))."""
    return a - (b * (a // b))

def es_multiplo(x, y):
    """True si x es múltiplo de y."""
    return obtenerresto(x, y) == 0

print(obtenerresto(10, 3))      # → 1
print(obtenerresto_sin_mod(10, 3)) # → 1
print(es_multiplo(12, 3))      # → True
```

## Composición de funciones


La **composición de funciones** ocurre cuando **se usa el resultado de una función como argumento de otra**. Es decir, **se encadenan** las llamadas, y Python ejecuta primero la más interna. Este mecanismo permite escribir código más compacto, expresivo y, muchas veces, más fácil de leer.


 **Consejo:** si hay muchas funciones anidadas, puede dificultar la lectura. En esos casos, usar variables auxiliares puede ser más claro.


```
def siguiente(n):  
    return n + 1  
def doble(n):  
    """Retorna 2 * n."""  
    return n * 2  
  
# Doble del siguiente de 4:  
print(doble(siguiente(4))) # → 10  
# Conversión y composición:  
valor = int(input("Introduce un entero: "))  
print("El siguiente es:", siguiente(valor))
```


## Buenas prácticas al trabajar con funciones


Aplicar buenas prácticas en tus funciones no solo mejora la calidad del código, sino que también lo hace más legible, mantenible y profesional.

 **Nombres descriptivos :** Elegí nombres que indiquen con claridad lo que hace la función. Ejemplo: `calcular_promedio()` es mejor que `func1()`.

 **Una responsabilidad por función:** Cada función debe tener una única tarea bien definida. Si una función hace demasiadas cosas, conviene dividirla.

 **Docstrings claros:** Agregá una breve descripción al principio de cada función usando comillas triples (`""" """`). Explicá qué hace, qué parámetros recibe y qué devuelve.

 **Evitar código duplicado:** Si copiás y pegás el mismo bloque varias veces, es señal de que podés convertirlo en una función reutilizable.

 **Probar con casos límite:** No te quedes solo con los “casos normales”. Probalas también con valores extremos, vacíos o inesperados para asegurarte de que la función es robusta.

## Teoría adicional: ámbitos y closures

Cada variable “**vive**” en un **determinado ámbito** o “alcance”. Esto define **dónde puede usarse** y **quién tiene acceso a ella**.

### ♦ **Ámbito local**

Las variables declaradas dentro de una función **sólo existen dentro de esa función**. No pueden ser accedidas desde fuera.

### ♦ **Ámbito global**

Las variables creadas fuera de cualquier función son **globales**, es decir, accesibles desde todo el archivo.


```
a = 10 # global

def cambiar():
    global a
    a = 20

cambiar()
print(a) # → 20
```

### **Closures (cierres):**

Un **closure** es una función **definida dentro de otra** que **recuerda el entorno en el que fue creada**, incluso después de que la función externa haya terminado de ejecutarse.

 Esto permite que la función interna **mantenga acceso a variables locales** de la función que la contiene, como si estuvieran “guardadas” en su memoria.

```
def crear_multiplicador(n):
    def multiplicar(x):
        return x * n # 'n' queda recordado
    return multiplicar

doble = crear_multiplicador(2)
triple = crear_multiplicador(3)

print(doble(5)) # → 10
print(triple(5)) # → 15
```