



# Paso por Valor y Referencia en Python

Una guía completa para entender cómo Python maneja los argumentos en funciones y cómo esto afecta a nuestro código.

# Agenda

## Conceptos fundamentales

Variables como referencias a objetos

Tipos mutables vs inmutables

## Ejemplos prácticos

Comportamiento con diferentes tipos de datos

Casos especiales y soluciones

## Buenas prácticas

Cómo evitar efectos secundarios

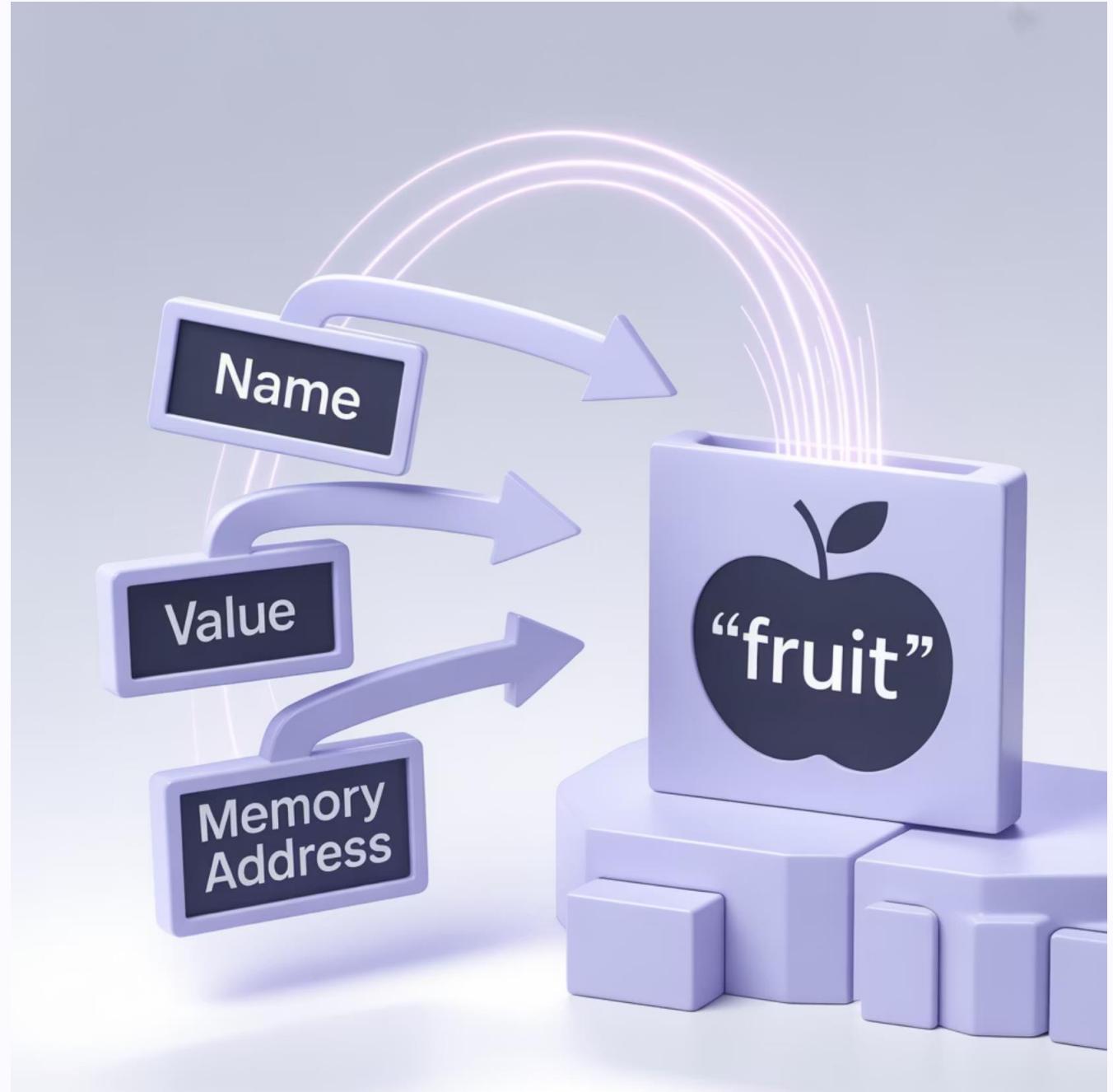
Recomendaciones para el código limpio

# Concepto Clave: Todo es un Objeto

En Python, **todos los datos son objetos**, y las variables son simplemente **referencias** a esos objetos.

Cuando pasamos una variable como argumento a una función:

- Se pasa una referencia al objeto, no una copia del valor
- El comportamiento depende del tipo de objeto (mutable o inmutable)
- Python no tiene un verdadero "paso por valor" como otros lenguajes



# Tipos de Datos en Python

## Tipos Inmutables

- int (enteros)
- float (decimales)
- str (cadenas)
- bool (booleanos)
- tuple (tuplas)
- frozenset

No pueden ser modificados después de su creación

## Tipos Mutables

- list (listas)
- dict (diccionarios)
- set (conjuntos)
- Objetos de clases personalizadas

Pueden ser modificados después de su creación

Esta distinción es fundamental para entender el comportamiento de los argumentos en funciones.

# Comportamiento de Paso de Argumentos

## Efecto de Paso por Valor

Ocurre con tipos **inmutables** (int, str, etc.)

Si modificamos el parámetro dentro de la función, se crea un nuevo objeto

La variable original fuera de la función **no se ve afectada**

## Efecto de Paso por Referencia

Ocurre con tipos **mutables** (list, dict, etc.)

Si modificamos el contenido del objeto, estamos modificando el objeto original

Los cambios **sí son visibles** fuera de la función

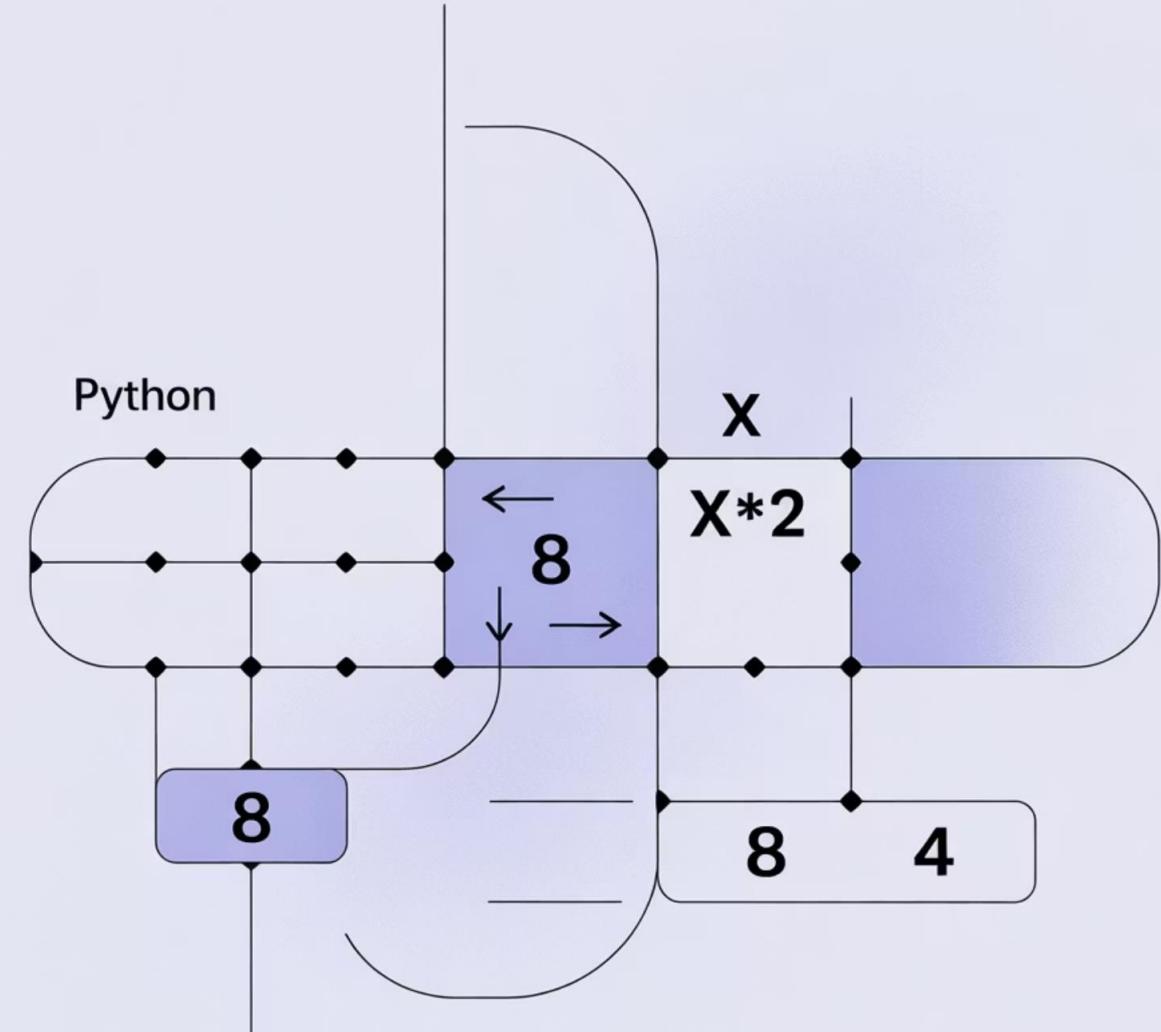
- ❑ ⚠ Técnicamente, Python siempre pasa referencias a objetos, pero el efecto observable depende de si el objeto es mutable o inmutable.

# Ejemplo: Tipos Inmutables (Enteros)

```
def duplicar(x):
    x = x * 2
    print("Dentro de la función:", x)
a = 10
duplicar(a)
print("Fuera de la función:", a)
```

Salida:

Dentro de la función: 20Fuera de la función: 10



Conclusión: Aunque `x` cambió dentro de la función, la variable `a` no se modificó afuera, porque

# Ejemplo: Tipos Mutables (Listas)

```
def agregar_elemento(lista):
    lista.append(4)
    print("Dentro de la función:", lista)
mi_lista = [1, 2, 3]
agregar_elemento(mi_lista)
print("Fuera de la función:", mi_lista)
```

Salida:

```
Dentro de la función: [1, 2, 3, 4]Fuera de la función: [1, 2, 3, 4]
```



Conclusión: La función modificó la lista original porque las listas son mutables y ambas

# El Caso Especial: Reasignación vs Modificación



## Reasignación

```
def cambiar_lista(lista):
    lista = [9, 9, 9] # Reasignación
    print("Dentro:", lista)
```

La variable local *lista* ahora apunta a un nuevo objeto

La variable original no cambia

## Modificación

```
def modificar_lista(lista):
    lista[0] = 99 # Modificación
    print("Dentro:", lista)
```

Se modifica el contenido del objeto original

El cambio es visible fuera de la función

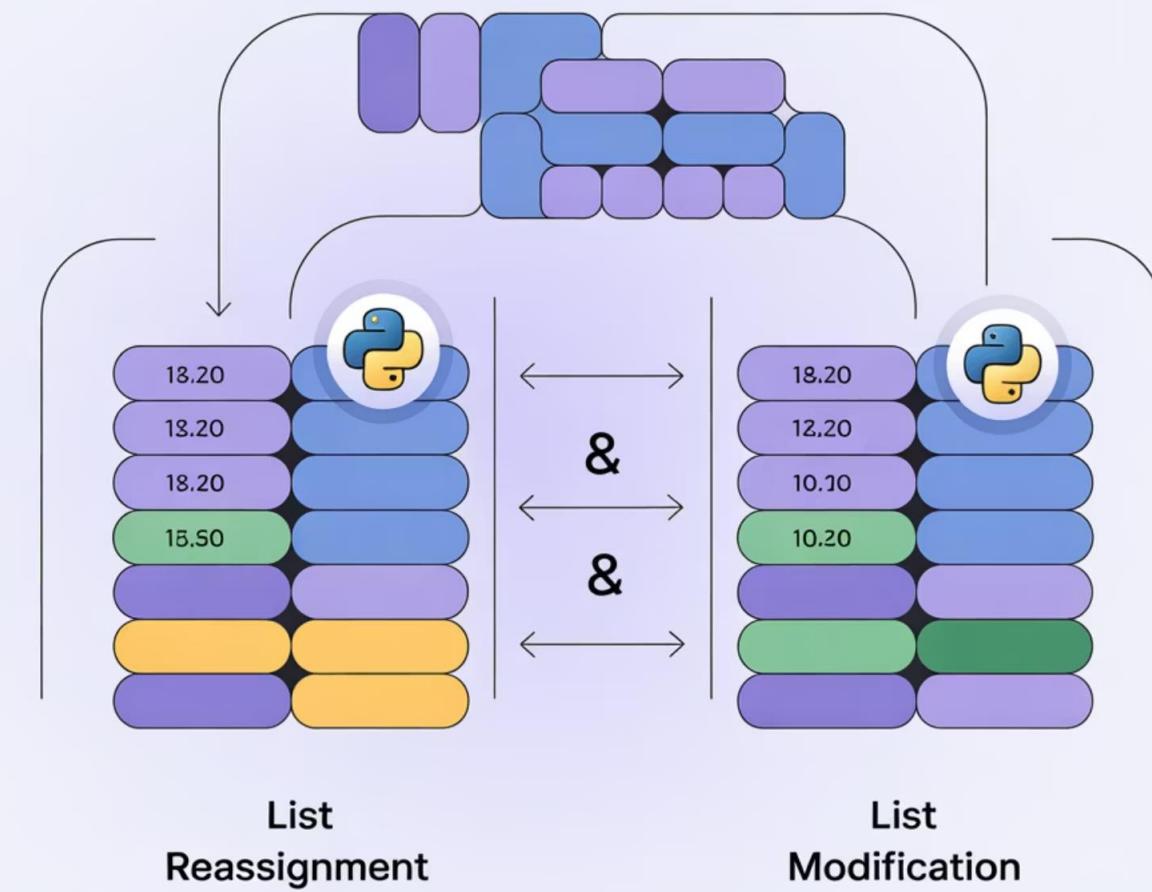
# Demostración: Reasignación vs Modificación

```
original = [1, 2, 3]
cambiar_lista(original)
print("Después de cambiar_lista:", original)
modificar_lista(original)
print("Después de modificar_lista:", original)
```

Salida:

```
Dentro (reasignación): [9, 9, 9]
Después de cambiar_lista: [1, 2, 3]
Dentro (modificación): [99, 2, 3]
Después de modificar_lista: [99, 2, 3]
```

## Python Memory Management



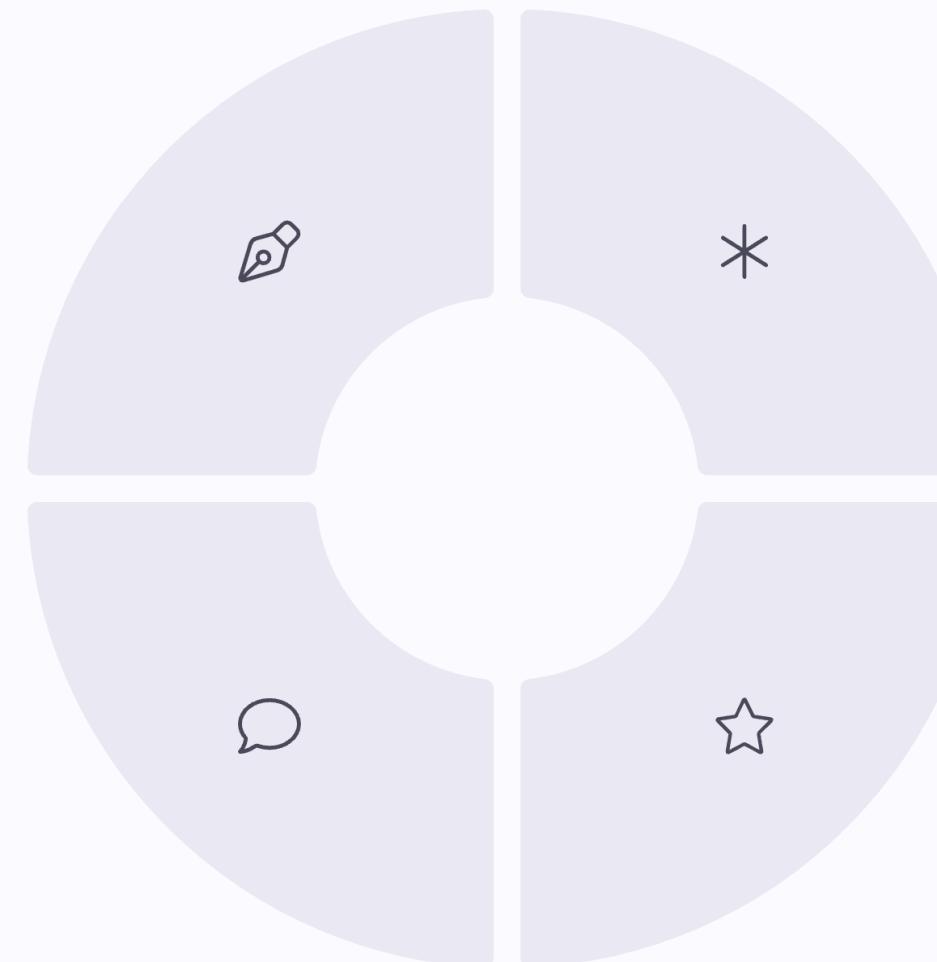
Clave para entender: La reasignación crea una nueva referencia local, mientras que la

# Visualización del Comportamiento

**Inmutables**  
int, float, str, bool, tuple  
No se pueden modificar  
Se crean nuevos objetos

## Reasignación

Crea nueva referencia  
No afecta al objeto original



**Referencias**  
Todas las variables son referencias  
Los argumentos pasan referencias

**Mutables**  
list, dict, set  
Se pueden modificar  
Cambios visibles globalmente

# Caso Especial: Parámetros Predeterminados

¡Cuidado con los mutables como valores por defecto!

```
def aniadir(elemento, lista=[]):
    lista.append(elemento)
    return listaprint(aniadir(1))
# [1]
print(aniadir(2))
# [1, 2] ¡Sorpresa!
```

Los valores predeterminados se evalúan **solo una vez** cuando se define la función, no cada vez que se llama.

Solución correcta:

```
def aniadir(elemento, lista=None):
    if lista is None:
        lista = []
    lista.append(elemento)
    return listaprint(añadir(1))
# [1]
print(añadir(2))
# [2] ¡Correcto!
```

Usar *None* como valor predeterminado y crear el objeto mutable dentro de la función.

# Evitar Efectos Secundarios

---

Usar copy() para listas y diccionarios

```
def procesar(lista):
    copia = lista.copy() # Copia superficial
    copia.append(4)
    return copia
original = [1, 2, 3]
resultado = procesar(original)
# original sigue siendo [1, 2, 3]
```

Usar deepcopy() para estructuras anidadas

```
import copy
def procesar_complejo(datos):
    copia = copy.deepcopy(datos)
    copia[0][0] = 99
    return copia
original = [[1, 2], [3, 4]]
resultado = procesar_complejo(original)
# original sigue siendo [[1, 2], [3, 4]]
```

# Buenas Prácticas

## ■ Evitar modificar argumentos mutables

Si necesitas modificar un objeto mutable, considera devolver una nueva versión en lugar de modificar el original.

## ■ Usar nombres claros para parámetros

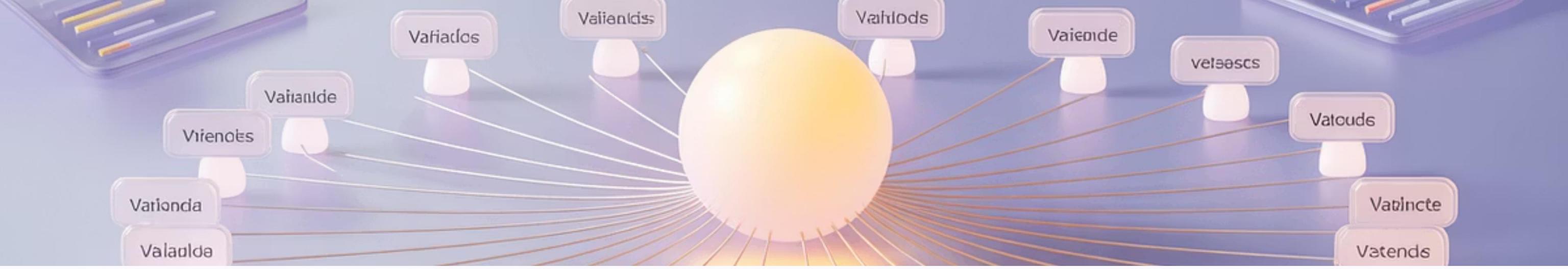
Elige nombres que indiquen el propósito y posible modificación, como `data_to_process` vs `original_data`.

## ■ Documentar el comportamiento esperado

Indica claramente en la documentación de tus funciones si modifican los argumentos que reciben.

## ■ Conocer la biblioteca estándar

Familiarízate con funciones que modifican *in-place* (como `list.sort()`) versus las que devuelven nuevos objetos (como `sorted()`).



# Resumen: Paso de Argumentos en Python

# Conceptos Clave

- Python siempre pasa referencias a objetos
  - El comportamiento observable depende de si el objeto es mutable o inmutable
  - La reasignación no afecta al objeto original
  - La modificación directa sí afecta al objeto original (si es mutable)

## Próximos Pasos

- Experimentar con diferentes tipos de datos
  - Practicar con `copy()` y `deepcopy()`
  - Revisar el código existente y optimizarlo
  - Explorar casos más complejos como funciones anidadas