💻Clase 10 - Funciones - Parte 2

✨ Argumentos posicionales de longitud variable con `args`

> 💡 A veces, el número de argumentos posicionales que se pasarán a una función no es seguro.

En tales casos, los argumentos posicionales de longitud variable se pueden recibir usando `*args`.

✅Un solo argumento:

```
def imprimelo(*args):
    print()
    for var in args:
        print(var, end = '')

imprimelo(10)  # 1 arg, ok
```

```
10
```

✅Dos argumentos:

```
def imprimelo(*args):
    print()
    for var in args:
        print(var, end = '')

imprimelo(10, 3.14)  # 2 arg, ok
```

```
103.14
```

✅Sin argumentos:

```
def imprimelo(*args):
    print()
    for var in args:
        print(var, end = '')

imprimelo()
```

Salida:

```

```

🧪 Más ejemplos con `args` (argumentos posicionales de longitud variable)

```
def imprimelo(*args):
    print()
    for var in args:
        print(var, end = '')

imprimelo(10, 3.14, 'Sicilian')  # 3 arg, ok
```

```
103.14Sicilian
```

```
def imprimelo(*args):
    print()
    for var in args:
        print(var, end = '')

imprimelo(10, 3.14, 'Sicilian', 'Punekar')  # 4 arg, ok
```

```
103.14SicilianPunekar
```

ℹ️ ¿Qué es `args`?

> 💡 El `args` utilizado en la definición de `imprimelo()` es una **tupla** y `*` indica que contendrá **todos los argumentos pasados** a la función `imprimelo()`.

La tupla se puede iterar mediante el uso de un bucle `for`.

🧩 Uso de `*kwargs` (argumentos de palabras clave de longitud variable)

A veces, el número de argumentos de palabras clave (**keywords**) que se pasarán a una función no es seguro. En tales casos, los argumentos de longitud variable se pueden recibir usando `**kwargs`.

```
def imprimelo(**kwargs):
    print()
    for name, value in kwargs.items():
        print(name, value, end = '')

imprimelo(a = 10)  # keyword, ok
```

```
a 10
```

🧪 Más ejemplos con `*kwargs` (argumentos de palabras clave de longitud variable)

```
def imprimelo(**kwargs):
    print()
    for name, value in kwargs.items():
        print(name, value, end = '')

imprimelo(a = 10, b = 3.14)  # keyword, ok
```

```
a 10b 3.14
```

```
def imprimelo(**kwargs):
    print()
    for name, value in kwargs.items():
        print(name, value, end = '')

imprimelo(a = 10, b = 3.14, s = 'Sicilian')  # keyword, ok
```

```
a 10b 3.14s Sicilian
```

📦 `**kwargs` a partir de un diccionario

```
def imprimelo(**kwargs):
    print()
    for name, value in kwargs.items():
        print(name, value, end = '')

diccionario = {'Estudiante': 'Mary ', 'Edad': 23}

imprimelo(**diccionario)  # ok
```

```
Estudiante Mary Edad 23
```

ℹ️ ¿Qué es `*kwargs`?

El `**kwargs` utilizado en la definición de `imprimelo()` es un **diccionario** que contiene nombres de variables como **claves** y sus valores como **valores**, mientras que `**` indica que contendrá **todos los argumentos pasados** a `imprimelo()`.

🧠 Reglas sobre `args` y `**kwargs`

Podemos usar cualquier otro nombre en lugar de `args` y `kwargs`.

No podemos usar **más de un** `*args` ni más de un `**kwargs` al definir una función.

🧩 Orden de los argumentos en funciones

Si una función va a recibir argumentos **requeridos** y **opcionales**, entonces deben ocurrir en el siguiente orden:

- Argumentos posicionales
- Argumentos posicionales de longitud variable (`args`)
- Argumentos de palabras clave (`keywords`)
- Argumentos de palabras clave de longitud variable (`*kwargs`)

✅ Ejemplo con todos los tipos de argumentos

```
def imprimelo(i, j, *args, x, y, **kwargs):
    print()
    print(i, j, end = '')
    for var in args:
        print(var, end = '')
    print(x, y, end = '')
    for name, value in kwargs.items():
        print(name, value, end = '')

# Ni args ni kwargs
imprimelo(10, 20, x = 30, y = 40)
```

```
10 2030 40
```

🧪 Combinando `args`, `*kwargs`, argumentos posicionales y de palabra clave

✅ `100`, `200` van a `args`, nada a `kwargs`

```
def imprimelo(i, j, *args, x, y, **kwargs):
    print()
    print(i, j, end = '')
    for var in args:
        print(var, end = '')
    print(x, y, end = '')
    for name, value in kwargs.items():
        print(name, value, end = '')

# 100, 200 van a args, nada va a kwargs
imprimelo(10, 20, 100, 200, x = 30, y = 40)
```

```
10 2010020030 40
```

✅ `a`, `b`, `c` van a `kwargs`

```
def imprimelo(i, j, *args, x, y, **kwargs):
    print()
    print(i, j, end = '')
    for var in args:
        print(var, end = '')
    print(x, y, end = '')
    for name, value in kwargs.items():
        print(name, value, end = '')

# 100, 200 van para args.
# 'a': 5, 'b': 6, 'c': 7 van para kwargs
imprimelo(10, 20, 100, 200, x = 30, y = 40, a = 5, b = 6, c = 7)
```

```
10 2010020030 40a 5b 6c 7
```

❌ Error: no se pasan `x` e `y`

```
def imprimelo(i, j, *args, x, y, **kwargs):
    print()
    print(i, j, end = '')
```

```
    for var in args:
        print(var, end = '')
    print(x, y, end = '')
    for name, value in kwargs.items():
        print(name, value, end = '')

# 30, 40 van para args y no queda nada para los argumentos requeridos x, y
# ERROR
imprimelo(10, 20, 30, 40)
```

```
TypeError: imprimelo() missing 2 required keyword-only arguments: 'x' and 'y'
```

Ejercicios de *args y **kwargs en Python

Ejercicio 1: Introducción a *args

Contexto:

- `args` permite pasar un número variable de argumentos posicionales a una función.

```
def sumar_numeros(*args):
    """
    Completa esta función para que sume todos los números
    que se le pasen como argumentos
    """
    pass

# Pruebas
print(sumar_numeros(1, 2, 3))          # Debería imprimir: 6
print(sumar_numeros(10, 20))           # Debería imprimir: 30
print(sumar_numeros(5))                # Debería imprimir: 5
print(sumar_numeros(1, 2, 3, 4, 5, 6))  # Debería imprimir: 21
```

Tarea:

1. Implementa la función `sumar_numeros()`
2. Imprime el tipo de dato de `args` dentro de la función (usa `print(type(args))`)
3. Explica por qué puedes pasar cualquier cantidad de números

💡 **Pista:** `args` es una tupla que contiene todos los argumentos posicionales

Ejercicio 2: Introducción a **kwargs

Contexto:

- `*kwargs` permite pasar un número variable de argumentos con nombre (keyword arguments) a una función.

```
def crear_perfil(**kwargs):
    """
    Completa esta función para que imprima todos los datos
    del perfil en formato: clave: valor
    """
    pass

# Pruebas
crear_perfil(nombre="Ana", edad=25, ciudad="Madrid")
# Salida esperada:
# nombre: Ana
```

```
# edad: 25
# ciudad: Madrid

crear_perfil(usuario="john_doe", email="john@example.com")
# Salida esperada:
# usuario: john_doe
# email: john@example.com
```

Tarea:

1. Implementa la función `crear_perfil()`
2. Imprime el tipo de dato de `kwargs` dentro de la función
3. Explica la diferencia entre `args` y `*kwargs`

💡 **Pista:** `kwargs` es un diccionario que contiene los argumentos con nombre

---

Ejercicio 3: Combinando *args y **kwargs

Objetivo: Usar ambos tipos de argumentos en la misma función

```
def hacer_pedido(cliente, *productos, **detalles):
    """
    Esta función debe:
    1. Mostrar el nombre del cliente
    2. Mostrar la lista de productos pedidos
    3. Mostrar los detalles adicionales del pedido (dirección, teléfono, etc.)
    """
    pass

# Pruebas
hacer_pedido(
    "Carlos",
    "Pizza",
    "Refresco",
    "Helado",
    direccion="Calle Mayor 10",
    telefono="123456789",
    pago="Tarjeta"
)

# Salida esperada:
# Cliente: Carlos
# Productos: Pizza, Refresco, Helado
# Dirección: Calle Mayor 10
# Teléfono: 123456789
# Pago: Tarjeta
```

Tarea:

1. Implementa la función `hacer_pedido()`
2. Asegúrate de manejar correctamente el parámetro regular, *args y **kwargs
3. Formatea la salida de manera clara y legible

⚠️ **Importante:** El orden SIEMPRE debe ser: parámetros normales, *args, **kwargs

---

Ejercicio 4: Calculadora Flexible

Objetivo: Crear una calculadora que acepte operaciones personalizadas

```
def calculadora(operacion, *numeros, **opciones):
    """
    Crea una calculadora que:
    - Reciba una operación: "suma", "resta", "multiplicacion", "promedio"
    - Reciba cualquier cantidad de números
    - Reciba opciones adicionales como:
      - redondear=True/False (redondea el resultado a 2 decimales)
      - mostrar_pasos=True/False (muestra los números usados)

    Retorna el resultado de la operación
    """
    pass

# Pruebas
print(calculadora("suma", 10, 20, 30))
# Salida: 60

print(calculadora("promedio", 10, 20, 30, 40))
# Salida: 25.0

print(calculadora("multiplicacion", 2, 3, 4, mostrar_pasos=True))
# Salida:
# Números: 2, 3, 4
# Resultado: 24

print(calculadora("suma", 10.5555, 20.7777, 30.1234, redondear=True))
# Salida: 61.46
```

Tarea:

1. Implementa las 4 operaciones: suma, resta, multiplicación y promedio
2. Implementa la opción `redondear` (usa `round()`)
3. Implementa la opción `mostrar_pasos`
4. Maneja el caso cuando se pase una operación inválida

⚙️ Argumentos con valores predeterminados

Al definir una función, se puede dar un valor predeterminado a los argumentos.

Se usará ese valor predeterminado **si no pasamos el valor** para ese argumento durante la llamada.

✅ Solo se pasa un argumento (los demás toman valores por defecto)

```
def func(a, b = 100, c = 3.14):
    return a + b + c

w = func(10)  # a toma el valor de 10, b = 100, c = 3.14
print(w)
```

```
113.14
```

✅ Se pasan dos argumentos

```
def func(a, b = 100, c = 3.14):
    return a + b + c
```

```
w = func(20, 50)  # a = 20, b = 50, c = 3.14
print(w)
```

```
73.14
```

✅ Se pasan los tres argumentos

```
def func(a, b = 100, c = 3.14):
    return a + b + c

w = func(30, 60, 6.28)  # a = 30, b = 60, c = 6.28
print(w)
```

```
96.28
```

✅ Se usan argumentos por palabra clave fuera de orden

```
def func(a, b = 100, c = 3.14):
    return a + b + c

w = func(1, c = 3, b = 5)  # a = 1, b = 5, c = 3
print(w)
```

```
9
```

> ⚠️ Tenga en cuenta que, al definir una función, los argumentos predeterminados deben ir después de los argumentos no predeterminados.

📦 Desempaquetando argumentos

Supongamos que una función contiene argumentos posicionales y los argumentos a pasar están en una lista, tupla o conjunto.

En tal caso, necesitamos **desempaquetar** la lista/tupla/conjunto usando el operador `*` antes de pasarlo a la función.

✅ Desempaquetando una lista

```
def imprimelo(a, b, c, d, e):
    print(a, b, c, d, e)

lista = [10, 20, 30, 40, 50]
imprimelo(*lista)
```

```
10 20 30 40 50
```

✅ Desempaquetando una tupla

```
def imprimelo(a, b, c, d, e):
    print(a, b, c, d, e)

tupla = ('A', 'B', 'C', 'D', 'E')
imprimelo(*tupla)
```

```
A B C D E
```

✅ Desempaquetando un conjunto

```
def imprimelo(a, b, c, d, e):
    print(a, b, c, d, e)

conjunto = {1, 2, 3, 4, 5}
imprimelo(*conjunto)
```

```
1 2 3 4 5
```

> ⚠️ Nota: En el caso del conjunto (set), los elementos pueden aparecer en un orden diferente porque los set en Python no tienen orden garantizado.

🧠 Desempaquetando diccionarios con argumentos por palabras clave

> Si una función espera argumentos por palabras clave y los valores están en un diccionario, podemos desempaquetarlos usando **.

```
def imprimelo(nombre='Sandra', edad=75, profesion='Data Analyst'):
    print(nombre, edad, profesion)

diccionario = {'nombre': 'Ana', 'edad': 50, 'profesion': 'Data Scientist'}

imprimelo(*diccionario)      # Pasa solo las claves (incorrecto)
imprimelo(**diccionario)     # Pasa las claves con sus valores
```

```
nombre edad profesion
Ana 50 Data Scientist
```

> La primera llamada pasa las claves, la segunda pasa los valores correctamente.

Ejercicios sobre Desempaquetado en Python

Ejercicio 1: Desempaquetado con el Operador * en Listas

Contexto: El operador `*` permite capturar múltiples elementos en una sola variable, creando una lista con los elementos restantes.

```
# Ejemplo básico
numeros = [1, 2, 3, 4, 5]
primero, *resto = numeros
print(primero)  # 1
print(resto)    # [2, 3, 4, 5]
```

Parte A: Extraer el primer elemento y el resto

```
temperaturas = [18, 20, 22, 19, 21, 23, 20]

# TODO: Usa * para capturar la primera temperatura y todas las demás

print(f"Temperatura inicial: {primera}°C")
```

```
print(f"Temperaturas siguientes: {siguientes}")
# Salida esperada:
# Temperatura inicial: 18°C
# Temperaturas siguientes: [20, 22, 19, 21, 23, 20]
```

Parte B: Extraer el último elemento

```
palabras = ["Python", "es", "un", "lenguaje", "genial"]

# TODO: Usa * para capturar todas las palabras excepto la última

print(f"Palabras principales: {principales}")
print(f"Última palabra: {ultima}")
# Salida esperada:
# Palabras principales: ['Python', 'es', 'un', 'lenguaje']
# Última palabra: genial
```

Parte C: Capturar elementos del medio

```
datos_sensor = [100, 98, 102, 99, 101, 103, 97, 105]

# TODO: Captura el primer valor, el último valor, y todos los valores intermedios usando *

print(f"Lectura inicial: {inicial}")
print(f"Lectura final: {final}")
print(f"Lecturas intermedias: {intermedias}")
print(f"Promedio intermedio: {sum(intermedias)/len(intermedias):.2f}")
# Salida esperada:
# Lectura inicial: 100
# Lectura final: 105
# Lecturas intermedias: [98, 102, 99, 101, 103, 97]
# Promedio intermedio: 100.00
```

Tarea:

1. Completa las tres partes del ejercicio usando el operador *
2. ¿Puedes usar más de un operador * en el mismo desempaquetado? Pruébalo
3. ¿Qué tipo de dato es la variable capturada por *?

Ejercicio 2: Desempaquetado Avanzado con * en Tuplas

Contexto: El operador `*` también funciona con tuplas y permite ignorar elementos que no necesitamos usando `_`.

```
# Ejemplo con tuplas
datos = (1, 2, 3, 4, 5)
primero, *_, ultimo = datos
print(primero, ultimo)  # 1 5 (ignora los del medio)
```

Parte A: Procesar información de productos

```
# Formato: (código, nombre, precio, stock, categoría, proveedor, fecha)
producto = ("P001", "Laptop", 999.99, 50, "Electrónica", "TechCorp", "2024-01-15")

# TODO: Extrae solo el código, nombre y precio, ignora el resto con *_

print(f"Código: {codigo}")
print(f"Producto: {nombre}")
print(f"Precio: ${precio}")
# Salida esperada:
```

```
# Código: P001
# Producto: Laptop
# Precio: $999.99
```

Parte B: Separar cabecera de datos

```
registro = ("ID", "Nombre", "Email", 101, "Ana García", "ana@example.com", 102, "Luis Pérez", "luis@example.
com")

# TODO: Separa las primeras 3 etiquetas (cabecera) del resto de datos usando *

print(f"Cabecera: {cabecera}")
print(f"Datos: {datos}")
print(f"Total de campos de datos: {len(datos)}")
# Salida esperada:
# Cabecera: ['ID', 'Nombre', 'Email']
# Datos: [101, 'Ana García', 'ana@example.com', 102, 'Luis Pérez', 'luis@example.com']
# Total de campos de datos: 6
```

Parte C: Desempaquetar argumentos de función

```
def crear_reporte(titulo, autor, *secciones, **metadatos):
    """
    Función que crea un reporte con:
    - titulo y autor (argumentos requeridos)
    - secciones variables (usando *)
    - metadatos adicionales (usando **)
    """
    print(f"=== {titulo} ===")
    print(f"Autor: {autor}\n")
    print("Secciones:")
    for i, seccion in enumerate(secciones, 1):
        print(f"  {i}. {seccion}")
    print("\nMetadatos:")
    for clave, valor in metadatos.items():
        print(f"  {clave}: {valor}")

# TODO: Llama a la función con diferentes cantidades de argumentos

# Test 1: Con 2 secciones y 2 metadatos
crear_reporte(
    "Informe Anual",
    "María López",
    "Introducción",
    "Resultados",
    fecha="2024-12-01",
    version="1.0"
)

print("\n" + "="*40 + "\n")

# Test 2: Con 4 secciones y 3 metadatos
crear_reporte(
    "Análisis Trimestral",
    "Carlos Ruiz",
    "Resumen Ejecutivo",
    "Análisis Financiero",
    "Proyecciones",
    "Conclusiones",
    fecha="2024-10-15",
```

```
    departamento="Finanzas",
    confidencial=True
)
```

Tarea:

1. Completa las tres partes
2. En la Parte C, ¿qué pasa si no pasas ninguna sección?
3. ¿Puedes llamar a la función solo con título y autor? Pruébalo

---

Ejercicio 3: Desempaquetado de Diccionarios con **

Contexto: El operador `**` desempaqueta diccionarios, útil para fusionar diccionarios o pasar argumentos a funciones.

```
# Ejemplo: Fusionar diccionarios
dict1 = {"a": 1, "b": 2}
dict2 = {"c": 3, "d": 4}
fusionado = {**dict1, **dict2}
print(fusionado)  # {'a': 1, 'b': 2, 'c': 3, 'd': 4}
```

Parte A: Fusionar configuraciones

```
config_default = {
    "idioma": "es",
    "tema": "claro",
    "notificaciones": True,
    "volumen": 50
}

config_usuario = {
    "tema": "oscuro",
    "volumen": 80
}

# TODO: Crea config_final fusionando ambos diccionarios con **
# La configuración del usuario debe sobrescribir la default

config_final = ...

print(config_final)
# Salida esperada:
# {'idioma': 'es', 'tema': 'oscuro', 'notificaciones': True, 'volumen': 80}
```

Parte B: Construir un perfil completo

```
info_personal = {"nombre": "Laura", "edad": 28}
info_contacto = {"email": "laura@example.com", "telefono": "555-0123"}
info_profesional = {"puesto": "Desarrolladora", "empresa": "TechCorp"}

# TODO: Crea un diccionario 'perfil_completo' que combine los tres usando **

perfil_completo = ...

print("Perfil Completo:")
for clave, valor in perfil_completo.items():
    print(f"  {clave}: {valor}")
# Salida esperada:
# Perfil Completo:
```

```
#   nombre: Laura
#   edad: 28
#   email: laura@example.com
#   telefono: 555-0123
#   puesto: Desarrolladora
#   empresa: TechCorp
```

Parte C: Pasar diccionarios como argumentos de función

```
def registrar_venta(producto, cantidad, precio, descuento=0, cliente="Anónimo"):
    total = cantidad * precio
    total_descuento = total * (1 - descuento)
    print(f"=== REGISTRO DE VENTA ===")
    print(f"Cliente: {cliente}")
    print(f"Producto: {producto}")
    print(f"Cantidad: {cantidad}")
    print(f"Precio unitario: ${precio}")
    print(f"Descuento: {descuento*100}%")
    print(f"Total: ${total_descuento:.2f}")

# Datos de venta almacenados en diccionarios
venta1 = {
    "producto": "Laptop",
    "cantidad": 2,
    "precio": 999.99,
    "descuento": 0.10,
    "cliente": "Empresa XYZ"
}

venta2 = {
    "producto": "Mouse",
    "cantidad": 5,
    "precio": 25.50
}

# TODO: Llama a la función registrar_venta usando ** para desempaquetar los diccionarios

print("VENTA 1:")
# Desempaqueta venta1 aquí

print("\nVENTA 2:")
# Desempaqueta venta2 aquí
```

Tarea:

1. Completa las tres partes usando el operador **
2. ¿Qué sucede si fusionas diccionarios con claves repetidas? ¿Cuál valor prevalece?
3. ¿Qué error obtienes si el diccionario tiene claves que no coinciden con los parámetros de la función?

Ejercicio 4: Desempaquetado de Sets y Casos Combinados

Parte A: Desempaquetado de Sets

```
# Los sets también se pueden desempaquetar (pero el orden no está garantizado)
numeros_unicos = {5, 2, 8, 1, 9}

# TODO: Desempaqueta los 5 números en variables individuales
a, b, c, d, e = ...
```

```
print(f"Números: {a}, {b}, {c}, {d}, {e}")
# Nota: El orden puede variar porque los sets no tienen orden definido
```

Parte B: Desempaquetado Anidado

```
# Estructuras anidadas pueden desempaquetarse en un solo paso
estudiantes = [
    ("Ana", [9, 8, 10]),
    ("Luis", [7, 9, 8]),
    ("María", [10, 10, 9])
]

# TODO: Desempaqueta el primer estudiante y sus tres calificaciones

nombre, (nota1, nota2, nota3) = estudiantes[0]

print(f"{nombre}: {nota1}, {nota2}, {nota3}")
# Salida esperada: Ana: 9, 8, 10
```

Parte C: Ejercicio Integrador

```
# Datos de una venta
venta = {
    "id": 1001,
    "productos": ["Laptop", "Mouse", "Teclado"],
    "precios": (999, 25, 75),
    "cliente": {"nombre": "Pedro", "tipo": "Premium"}
}

# TODO: Extrae la siguiente información:
# - ID de la venta
# - Primer producto y el resto
# - Todos los precios en variables separadas
# - Nombre del cliente

id_venta = ...
primer_producto, *otros_productos = ...
precio1, precio2, precio3 = ...
nombre_cliente = ...

print(f"Venta #{id_venta}")
print(f"Cliente: {nombre_cliente}")
print(f"Producto principal: {primer_producto} - ${precio1}")
print(f"Otros productos: {otros_productos}")
print(f"Total: ${precio1 + precio2 + precio3}")

# Salida esperada:
# Venta #1001
# Cliente: Pedro
# Producto principal: Laptop - $999
# Otros productos: ['Mouse', 'Teclado']
# Total: $1099
```

Tarea:

1. Completa las tres partes
2. Intenta resolver la Parte C de la forma más eficiente posible
3. ¿Puedes pensar en un caso real donde usarías desempaquetado anidado?

🧠 Alcance y vida útil de parámetros

> Los parámetros y variables **definidos dentro de una función** tienen un **alcance local**, por lo tanto:

- No son visibles desde fuera de la función.
- Solo existen mientras se ejecuta la función.
- Se destruyen una vez que salimos de la función.

> En cambio, las variables externas a la función sí son visibles dentro de ella, pero:

- Podemos **leer** sus valores,
- No podemos **modificarlas** directamente,
- Para modificarlas, debemos declararlas como **globales** usando la palabra clave `global`.

> ⚠️ Cuando se llama a una función, el intérprete primero busca en el ámbito local y luego en el ámbito global.
>
> Si no encuentra el nombre, se lanza un error del tipo `NameError`.

🔷 Uso de una variable externa a la función

```
def f():
    print(s)

s = 'Python'
f()
```

```
Python
```

> ✔️ La variable s se define fuera, pero se puede leer dentro.

🔷 Variables interna y externa con el mismo nombre

```
def f():
    s = 'Mundo'
    print(s)

s = 'Python'
f()
print(s)
```

```
Mundo
Python
```

> ✔️ La variable `s` dentro de la función es distinta de la `s` global. No se sobrescribe.

🧠 Alcance y vida útil de variables en Python

En Python, **todas las variables definidas dentro de una función son locales por defecto**, a menos que se indique lo contrario. Esto significa que:

- Solo existen **mientras la función se está ejecutando**.
- **No pueden ser accedidas desde fuera** de esa función.
- Al salir de la función, **se destruyen automáticamente**.

> 💡 Usar variables locales es una **buena práctica de programación**, ya que ayuda a evitar errores y conflictos. En general, **es preferible pasar información a una función usando parámetros** o **devolver un valor** con `return`, en lugar de usar o modificar variables externas.

🧠¿Qué pasa con las variables externas (globales)?

Las variables definidas **fuera de una función** se conocen como **variables globales**. Estas variables:

- Son visibles **dentro de las funciones** (pero solo para lectura).
- **No pueden ser modificadas directamente** dentro de una función, a menos que se declaren como `global`.

Si una función necesita **modificar** una variable global, se debe declarar explícitamente con la palabra clave `global`.

```
s = "Hola"

def cambiar():
    global s
    s = "Adiós"

cambiar()
print(s)  # Salida: Adiós
```

🧠 Errores comunes: usar variables globales sin declararlas

Si intentas **usar y modificar** una variable global dentro de una función **sin declararla como global**, Python lanzará un error porque considera que estás trabajando con una variable local no inicializada:

```
s = "Hola"

def fallo():
    print(s)   # Error: variable local 's' usada antes de asignar valor
    s = "Adiós"

fallo()
```

Salida de error:

```
UnboundLocalError: local variable 's' referenced before assignment
```

> Si definimos la variable externa como global dentro de la función, ya no aparecerá ningún error. Y podremos cambiar el valor de la variable global desde dentro de la función.

✅ Con `global`

```
s = 'Alfa'

def f():
    global s
    print(f'El valor 1 de s es: {s}')
    s = 'Mundo'
    print(f'El valor 2 de s es: {s}')

s = 'Python'
f()
print()
print(f'El valor 3 de s es: {s}')
```

Salida:

```
El valor 1 de s es: Python
El valor 2 de s es: Mundo

El valor 3 de s es: Mundo
```

❌ Sin `global`

```
s = 'Alfa'

def f():
    # global s
    s = 'Mundillo'
    print(f'El valor 1 de s es: {s}')
    s = 'Mundo'
    print(f'El valor 2 de s es: {s}')

s = 'Python'
f()
print()
print(f'El valor 3 de s es: {s}')
```

Salida:

```
El valor 1 de s es: Mundillo
El valor 2 de s es: Mundo

El valor 3 de s es: Python
```

Ejercicios de Variables Locales y Globales en Python

Ejercicio 1: Identificar el Scope

Objetivo: Comprender qué es una variable local vs global

```
nombre = "Ana"  # ¿Esta variable es local o global?

def saludar():
    edad = 25  # ¿Esta variable es local o global?
    print(f"Hola {nombre}, tienes {edad} años")

saludar()
print(edad)  # ¿Qué pasará aquí?
```

Tarea: Ejecuta el código, identifica el error y explica por qué ocurre.

Ejercicio 2: Modificar una Variable Global

Objetivo: Usar la palabra clave `global`

```
contador = 0

def incrementar():
    # Modifica esta función para que incremente el contador global
    contador = contador + 1

incrementar()
```

```
incrementar()
print(contador)  # Debería imprimir 2
```

Tarea: Corrige la función para que modifique correctamente la variable global.

Ejercicio 3: Variable Local con el Mismo Nombre

Objetivo: Entender la prioridad de variables locales

```
mensaje = "Global"

def mostrar_mensaje():
    mensaje = "Local"
    print(mensaje)

mostrar_mensaje()
print(mensaje)
```

Tarea: Predice qué se imprimirá en cada `print()` antes de ejecutar el código. Luego ejecuta y verifica.

Ejercicio 4: Calculadora de Descuentos

Objetivo: Aplicar conceptos en un caso práctico

```
descuento_global = 0.10  # 10% de descuento

def calcular_precio(precio_original):
    # Crea una variable local 'precio_final' que aplique el descuento
    # Usa la variable global descuento_global
    pass

# Prueba tu función
print(calcular_precio(100))  # Debería imprimir 90.0
```

Tarea: Completa la función para calcular el precio con descuento.

Ejercicio 5: Sistema de Puntos de un Juego

Objetivo: Combinar variables locales y globales

```
puntos_totales = 0

def ganar_puntos(puntos):
    # Incrementa puntos_totales con la cantidad recibida
    pass

def perder_puntos(puntos):
    # Reduce puntos_totales con la cantidad recibida
    # No permitas que los puntos sean negativos
    pass

def mostrar_puntos():
    # Muestra los puntos actuales
    pass

# Prueba el sistema
ganar_puntos(50)
ganar_puntos(30)
```

```
perder_puntos(20)
mostrar_puntos()  # Debería mostrar: "Puntos totales: 60"
```

Tarea: Implementa las tres funciones usando correctamente variables globales y locales.

Ejercicios Propuestos: Variables Locales y Globales en Python

Ejercicio 6: Control de Temperatura de un Termostato

Descripción: Crea un programa que simule el control de temperatura de un termostato inteligente. El programa debe cumplir con los siguientes requisitos:

Requisitos:

1. Debe existir una variable global llamada `temperatura_actual` que comience en 20 grados
2. Debe existir una variable global llamada `temperatura_deseada` que comience en 22 grados
3. Implementa las siguientes funciones:
   - `aumentar_temperatura(grados)`: aumenta la temperatura actual
   - `disminuir_temperatura(grados)`: disminuye la temperatura actual
   - `establecer_temperatura_deseada(nueva_temperatura)`: cambia la temperatura deseada
   - `mostrar_estado()`: muestra tanto la temperatura actual como la deseada
   - `necesita_calefaccion()`: retorna `True` si la temperatura actual es menor a la deseada, `False` en caso contrario

Ejemplo de uso esperado:

```
mostrar_estado()
# Salida: Temperatura actual: 20°C | Temperatura deseada: 22°C

aumentar_temperatura(3)
print(necesita_calefaccion())  # Salida: False

establecer_temperatura_deseada(25)
print(necesita_calefaccion())  # Salida: True
```

Ejercicio 7: Sistema de Inventario de una Tienda

Descripción: Desarrolla un sistema simple de inventario para una tienda que gestione la cantidad de productos en stock y el total de ventas realizadas.

Requisitos:

1. Crea dos variables globales:
   - `stock_productos`: debe comenzar en 100 unidades
   - `ventas_totales`: debe comenzar en 0
2. Implementa las siguientes funciones:
   - `vender_producto(cantidad)`: reduce el stock y suma al total de ventas (cada producto cuesta $50)
   - `reponer_stock(cantidad)`: añade productos al inventario
   - `consultar_inventario()`: muestra el stock actual y las ventas totales en dinero
   - `verificar_stock_bajo()`: retorna `True` si el stock es menor a 20 unidades (para alertar que se necesita reposición)

Validaciones importantes:

- No se puede vender más productos de los que hay en stock
- No se pueden vender cantidades negativas

- El stock no puede ser negativo

Ejemplo de uso esperado:

```
consultar_inventario()
# Salida: Stock disponible: 100 unidades | Ventas totales: $0

vender_producto(15)
consultar_inventario()
# Salida: Stock disponible: 85 unidades | Ventas totales: $750

vender_producto(100)
# Salida: Error: No hay suficiente stock disponible

reponer_stock(50)
print(verificar_stock_bajo())  # Salida: False
```

Practica 09