Clase 9: Funciones II

Argumentos Avanzados

Resumen de Estudio

Curso de Python - Fundamentos

Este documento cubre los argumentos avanzados de funciones en Python: argumentos por defecto, *args, **kwargs, orden de parametros y unpacking. Dominar estos conceptos te permitira crear funciones mucho mas flexibles y reutilizables.

1. Argumentos por Defecto

Que son?

Los argumentos por defecto son valores predeterminados que se asignan a un parametro cuando no se proporciona un valor al llamar la funcion. Esto hace las funciones mas flexibles ya que algunos argumentos se vuelven opcionales.

Sintaxis:

```
def funcion(param_obligatorio, param_opcional=valor_default):
    # El param_opcional usara valor_default si no se pasa
    pass

# Ejemplos de uso:
funcion("valor")           # param_opcional = valor_default
funcion("valor", "otro")   # param_opcional = "otro"
```

Ejemplo Practico:

```
def saludar(nombre, idioma="espanol"):
    if idioma == "espanol":
        return f"Hola, {nombre}!"
    elif idioma == "ingles":
        return f"Hello, {nombre}!"
    else:
        return f"Hi, {nombre}!"

# Llamadas
print(saludar("Ana"))              # Usa default: "Hola, Ana!"
print(saludar("Ana", "ingles"))    # "Hello, Ana!"

# Multiples defaults
def crear_config(host, puerto=8080, timeout=30, ssl=False):
    return {
        "host": host,
        "puerto": puerto,
        "timeout": timeout,
        "ssl": ssl
    }

config1 = crear_config("localhost")  # Usa todos los defaults
config2 = crear_config("api.com", 443, ssl=True)  # Cambia algunos
```

Regla Importante:

REGLA: Los parametros con default SIEMPRE van DESPUES de los parametros sin default.
def bien(a, b, x=10): # Correcto
def mal(a, x=10, b): # SyntaxError!

2. *args - Argumentos Variables Posicionales

Que es *args?

*args permite que una funcion reciba un numero variable de argumentos posicionales (sin nombre). Dentro de la funcion, args es una TUPLA que contiene todos los argumentos extra que se pasaron.

```
def sumar_todos(*numeros):
    print(f"Recibido: {numeros}")     # Es una tupla
    print(f"Tipo: {type(numeros)}")   # <class 'tuple'>

    total = 0
    for num in numeros:
        total += num
    return total

# Llamadas con diferente cantidad de argumentos
print(sumar_todos(1, 2))           # 3
print(sumar_todos(1, 2, 3, 4, 5))  # 15
print(sumar_todos(10))             # 10
print(sumar_todos())               # 0 (tupla vacia)
```

Combinando con parametros normales:

```
def calcular(operacion, *valores):
    '''El primer argumento va a 'operacion', el resto a *valores'''
    if operacion == "suma":
        return sum(valores)
    elif operacion == "promedio":
        return sum(valores) / len(valores) if valores else 0
    elif operacion == "maximo":
        return max(valores) if valores else None

# Uso
print(calcular("suma", 1, 2, 3, 4, 5))      # 15
print(calcular("promedio", 10, 20, 30))     # 20.0
print(calcular("maximo", 5, 2, 8, 1, 9))    # 9
```

NOTA: El nombre 'args' es solo CONVENCION. Lo importante es el asterisco (*).
Puedes usar *numeros, *valores, *items, etc.

3. **kwargs - Argumentos Variables con Nombre

Que es **kwargs?

**kwargs permite que una funcion reciba un numero variable de argumentos con nombre (keyword arguments). Dentro de la funcion, kwargs es un DICCIONARIO donde las claves son los nombres de los argumentos y los valores son los valores pasados.

```
def crear_usuario(nombre, **datos_extra):
    print(f"Datos extra: {datos_extra}")      # Es un diccionario
    print(f"Tipo: {type(datos_extra)}")       # <class 'dict'>

    usuario = {"nombre": nombre}

    # Agregamos todos los datos extra
    for clave, valor in datos_extra.items():
        usuario[clave] = valor

    return usuario

# Llamadas con diferentes argumentos con nombre
u1 = crear_usuario("Ana", edad=25, ciudad="Madrid")
# {'nombre': 'Ana', 'edad': 25, 'ciudad': 'Madrid'}

u2 = crear_usuario("Carlos", profesion="Developer", experiencia=5, remoto=True)
# {'nombre': 'Carlos', 'profesion': 'Developer', 'experiencia': 5, 'remoto': True}

u3 = crear_usuario("Luis")  # Sin datos extra
# {'nombre': 'Luis'}
```

NOTA: El nombre 'kwargs' viene de 'keyword arguments'. Es convencion.
Puedes usar **opciones, **config, **datos, etc.
Las claves del diccionario son siempre strings (nombres de los argumentos).

4. Orden de Parametros

El orden de los tipos de parametros en Python esta FIJO y no se puede cambiar. Si usas el orden incorrecto, obtendras un SyntaxError.

Posicion	Tipo	Ejemplo	Descripcion
1	Normales	a, b	Parametros obligatorios
2	*args	*args	Argumentos posicionales extra
3	Default	x=10	Parametros con valor por defecto
4	**kwargs	**kwargs	Argumentos con nombre extra

```
# ORDEN CORRECTO
def funcion_completa(a, b, *args, x=10, y=20, **kwargs):
    print(f"Normales: a={a}, b={b}")
    print(f"*args: {args}")
    print(f"Default: x={x}, y={y}")
    print(f"**kwargs: {kwargs}")

# Llamada de ejemplo
funcion_completa(1, 2, 3, 4, 5, x=100, extra="hola", debug=True)
# Normales: a=1, b=2
# *args: (3, 4, 5)
# Default: x=100, y=20
# **kwargs: {'extra': 'hola', 'debug': True}

# ORDENES INCORRECTOS (dan SyntaxError)
# def mal1(a, x=10, *args):     # default antes de *args
# def mal2(a, **kwargs, b):     # normal despues de **kwargs
# def mal3(*args, a):           # normal despues de *args
```

MNEMOTECNICO para recordar el orden:
N-A-D-K: Normal, *Args, Default, **Kwargs
O: 'Normales primero, Asteriscos y Defaults despues, Kwargs al final'

5. Unpacking (Desempaquetado)

Unpacking permite pasar elementos de una lista/tupla o diccionario como argumentos individuales a una funcion, en lugar de pasar la estructura completa.

Unpacking de listas/tuplas con *

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

numeros = [10, 20, 30]

# Sin unpacking - ERROR
# sumar(numeros)  # TypeError: espera 3 argumentos, recibe 1

# Con unpacking - CORRECTO
resultado = sumar(*numeros)   # Equivale a sumar(10, 20, 30)
print(resultado)              # 60

# Tambien funciona con tuplas
datos = (1, 2, 3)
print(sumar(*datos))          # 6
```

Unpacking de diccionarios con **

```
def crear_perfil(nombre, edad, ciudad):
    return f"{nombre}, {edad} anios de {ciudad}"

datos = {
    "nombre": "Ana",
    "edad": 25,
    "ciudad": "Madrid"
}

# Con unpacking
perfil = crear_perfil(**datos)
# Equivale a: crear_perfil(nombre="Ana", edad=25, ciudad="Madrid")
print(perfil)  # "Ana, 25 anios de Madrid"

# IMPORTANTE: Las claves del dict deben coincidir con los nombres de parametros
```

Combinando unpacking:

```
def configurar(host, puerto, *extras, timeout=30, **opciones):
    print(f"Servidor: {host}:{puerto}, timeout={timeout}")
    print(f"Extras: {extras}")
    print(f"Opciones: {opciones}")

# Datos en diferentes estructuras
servidor = ["localhost", 8080]
config = {"debug": True, "log": "verbose"}

# Combinamos todo en una llamada
configurar(*servidor, "extra1", "extra2", timeout=60, **config)
# Servidor: localhost:8080, timeout=60
# Extras: ('extra1', 'extra2')
# Opciones: {'debug': True, 'log': 'verbose'}
```

6. Tabla Comparativa

Aspecto	Normales	*args	Default	**kwargs
Sintaxis def	def f(a, b):	def f(*args):	def f(x=10):	def f(**kw):
Sintaxis llamada	f(1, 2)	f(1, 2, 3, ...)	f() o f(x=5)	f(a=1, b=2)
Obligatorio?	Si	No	No	No
Tipo recibido	Variable	Tupla	Variable	Diccionario
Cantidad	Fija	Variable (0+)	Fija (0-1)	Variable (0+)
Posicion en def	1ro	2do	3ro	4to (ultimo)

7. Cuando Usar Cada Tipo

Argumentos por Default:

- Cuando un parametro tiene un valor tipico que se usa la mayoria de las veces - Para configuraciones opcionales (timeout=30, debug=False) - Cuando quieres hacer argumentos opcionales sin usar None - Ejemplos: open(file, mode='r'), print(end='\n'), range(start=0)

*args:

- Cuando no sabes CUANTOS argumentos recibiras - Para funciones matematicas (suma, promedio de N numeros) - Para wrappers que pasan argumentos a otras funciones - Ejemplos: print(*values), max(*args), min(*args)

**kwargs:

- Cuando no sabes QUE argumentos (con nombre) recibiras - Para funciones de configuracion flexible - Para crear objetos con atributos dinamicos - Ejemplos: dict(**kwargs), configuraciones de API

EJERCICIOS ADICIONALES

NIVEL PRINCIPIANTE

Ejercicio P1: Saludo Personalizado

Crea una funcion saludar_usuario(nombre, saludo='Hola', despedida=False) que: - Si despedida es False, retorne '{saludo}, {nombre}!' - Si despedida es True, retorne 'Adios, {nombre}!' Prueba con: saludar_usuario('Ana'), saludar_usuario('Luis', 'Buenos dias'), saludar_usuario('Maria', despedida=True)

Ejercicio P2: Calculadora Basica

Crea una funcion calcular(*numeros) que reciba cualquier cantidad de numeros y retorne un diccionario con: suma, cantidad, minimo y maximo. Si no recibe numeros, retornar un dict con valores None.

Ejercicio P3: Formateador de Texto

Crea una funcion formatear(texto, mayusculas=False, titulo=False, invertir=False) que: - Si mayusculas=True, convierta el texto a mayusculas - Si titulo=True, convierta a formato titulo (cada palabra con mayuscula inicial) - Si invertir=True, invierta el texto Los parametros se pueden combinar.

Ejercicio P4: Constructor de Lista

Crea una funcion construir_lista(*elementos, ordenar=False, unicos=False) que: - Reciba cualquier cantidad de elementos - Si ordenar=True, retorne la lista ordenada - Si unicos=True, elimine duplicados Prueba con: construir_lista(3, 1, 2, 1, 3, ordenar=True, unicos=True)

Ejercicio P5: Diccionario Dinamico

Crea una funcion crear_dict(**pares) que reciba cualquier cantidad de argumentos con nombre y retorne un diccionario. Agrega una clave 'total_claves' con el numero de claves recibidas. Prueba: crear_dict(nombre='Ana', edad=25, ciudad='Madrid')

NIVEL INTERMEDIO

Ejercicio I1: Sistema de Pedidos

Crea un sistema con dos funciones: 1. crear_pedido(cliente, *productos, envio='estandar', **extras) - productos son tuplas (nombre, precio, cantidad) - envio puede ser 'estandar' (gratis), 'express' (+15), 'premium' (+30) - extras puede incluir: descuento, regalo, notas, etc. - Retorna dict con toda la info y total calculado 2. resumen_pedidos(*pedidos) - Recibe varios pedidos - Retorna: total_pedidos, suma_totales, pedido_mayor, clientes_unicos

Ejercicio I2: Validador de Datos

Crea una funcion validar_datos(**campos) que: - Reciba cualquier cantidad de campos como clave=valor - Valide: nombre (string no vacio), edad (int positivo), email (contiene @) - Retorne dict con: validos (dict de campos validos), errores (dict campo: mensaje) - Campos no reconocidos se agregan a 'extras' sin validar

Ejercicio I3: Logger Flexible

Crea una funcion log(mensaje, *extras, nivel='INFO', timestamp=True, **metadata): - mensaje: texto principal del log - *extras: datos adicionales a incluir en el log - nivel: 'DEBUG', 'INFO', 'WARNING', 'ERROR' (default 'INFO') - timestamp: si True, incluir fecha/hora actual - **metadata: cualquier dato extra (usuario, modulo, version, etc.) Retorna string formateado: [NIVEL][TIMESTAMP] mensaje extras | metadata

Ejercicio I4: Filtrador de Coleccion

Crea una funcion filtrar(coleccion, **criterios) que: - coleccion: lista de diccionarios - **criterios: campos y valores a filtrar - Retorne solo los elementos que cumplan TODOS los criterios Ejemplo: filtrar(usuarios, edad=25, activo=True) retorna usuarios de 25 anios activos

Ejercicio I5: Constructor de Funciones

Crea una funcion crear_multiplicador(factor=2) que RETORNE otra funcion. La funcion retornada debe multiplicar su argumento por el factor. Ejemplo: doble = crear_multiplicador(2) triple = crear_multiplicador(3) print(doble(5)) # 10 print(triple(5)) # 15

NIVEL AVANZADO

Ejercicio A1: Decorador Simple

Un decorador es una funcion que modifica el comportamiento de otra funcion. Crea un decorador llamado 'contador' que cuente cuantas veces se llama una funcion. def contador(funcion): # Tu codigo aqui # Debe retornar una funcion wrapper que: # 1. Incremente un contador # 2. Llame a la funcion original # 3. Imprima 'Llamada #N' pass @contador def saludar(nombre): return f'Hola, {nombre}' saludar('Ana') # Llamada #1 saludar('Luis') # Llamada #2

Ejercicio A2: Decorador con Argumentos

Crea un decorador 'repetir(veces=2)' que ejecute la funcion decorada N veces. def repetir(veces=2): # Tu codigo aqui pass @repetir(veces=3) def saludar(nombre): print(f'Hola, {nombre}') saludar('Ana') # Imprime 'Hola, Ana' 3 veces

Ejercicio A3: Sistema de Plugins

Crea un sistema de plugins que permita registrar y ejecutar funciones dinamicamente: 1. crear_sistema() - Retorna un dict con las funciones del sistema 2. registrar_plugin(sistema, nombre, funcion) - Agrega un plugin al sistema 3. ejecutar_plugin(sistema, nombre, *args, **kwargs) - Ejecuta un plugin 4. listar_plugins(sistema) - Lista todos los plugins registrados 5. ejecutar_todos(sistema, *args, **kwargs) - Ejecuta todos los plugins Cada plugin debe poder recibir cualquier combinacion de argumentos.

SOLUCIONES

Soluciones Nivel Principiante

Solucion P1:

```
def saludar_usuario(nombre, saludo="Hola", despedida=False):
    if despedida:
        return f"Adios, {nombre}!"
    return f"{saludo}, {nombre}!"

# Pruebas
print(saludar_usuario("Ana"))                    # Hola, Ana!
print(saludar_usuario("Luis", "Buenos dias"))    # Buenos dias, Luis!
print(saludar_usuario("Maria", despedida=True))  # Adios, Maria!
```

Solucion P2:

```
def calcular(*numeros):
    if not numeros:
        return {"suma": None, "cantidad": 0, "minimo": None, "maximo": None}

    return {
        "suma": sum(numeros),
        "cantidad": len(numeros),
        "minimo": min(numeros),
        "maximo": max(numeros)
    }

# Pruebas
print(calcular(1, 2, 3, 4, 5))  # {'suma': 15, 'cantidad': 5, 'minimo': 1, 'maximo': 5}
print(calcular())               # {'suma': None, 'cantidad': 0, 'minimo': None, 'maximo': None}
```

Solucion P3:

```
def formatear(texto, mayusculas=False, titulo=False, invertir=False):
    resultado = texto

    if mayusculas:
        resultado = resultado.upper()
    if titulo:
        resultado = resultado.title()
    if invertir:
        resultado = resultado[::-1]

    return resultado

# Pruebas
print(formatear("hola mundo"))                          # hola mundo
print(formatear("hola mundo", mayusculas=True))         # HOLA MUNDO
print(formatear("hola mundo", titulo=True))             # Hola Mundo
print(formatear("hola mundo", invertir=True))           # odnum aloh
print(formatear("hola", mayusculas=True, invertir=True))  # ALOH
```

Solucion P4:

```
def construir_lista(*elementos, ordenar=False, unicos=False):
    resultado = list(elementos)

    if unicos:
        resultado = list(set(resultado))

    if ordenar:
        resultado.sort()

    return resultado

# Pruebas
print(construir_lista(3, 1, 2, 1, 3))                          # [3, 1, 2, 1, 3]
print(construir_lista(3, 1, 2, 1, 3, ordenar=True))            # [1, 1, 2, 3, 3]
print(construir_lista(3, 1, 2, 1, 3, unicos=True))             # [1, 2, 3] (orden puede variar)
print(construir_lista(3, 1, 2, 1, 3, ordenar=True, unicos=True))  # [1, 2, 3]
```

Solucion P5:

```
def crear_dict(**pares):
    resultado = dict(pares)
    resultado["total_claves"] = len(pares)
    return resultado

# Pruebas
print(crear_dict(nombre="Ana", edad=25, ciudad="Madrid"))
# {'nombre': 'Ana', 'edad': 25, 'ciudad': 'Madrid', 'total_claves': 3}

print(crear_dict(a=1, b=2))
# {'a': 1, 'b': 2, 'total_claves': 2}
```

Soluciones Nivel Intermedio

Solucion I1 - Parte 1 (crear_pedido):

```
def crear_pedido(cliente, *productos, envio="estandar", **extras):
    costos_envio = {"estandar": 0, "express": 15, "premium": 30}
    subtotal = 0
    items = []
    for producto in productos:
        nombre, precio, cantidad = producto
        total_item = precio * cantidad
        subtotal += total_item
        items.append({"nombre": nombre, "precio": precio,
                      "cantidad": cantidad, "total": total_item})
    costo_envio = costos_envio.get(envio, 0)
    descuento = extras.get("descuento", 0)
    total = subtotal + costo_envio - descuento
    pedido = {"cliente": cliente, "items": items, "subtotal": subtotal,
              "envio": envio, "costo_envio": costo_envio, "total": total}
    for k, v in extras.items():
        if k != "descuento": pedido[k] = v
    return pedido
```

Solucion I1 - Parte 2 (resumen_pedidos):

```
def resumen_pedidos(*pedidos):
    if not pedidos:
        return {"total_pedidos": 0, "suma_totales": 0}
    suma = sum(p["total"] for p in pedidos)
    mayor = max(pedidos, key=lambda p: p["total"])
    clientes = list(set(p["cliente"] for p in pedidos))
    return {"total_pedidos": len(pedidos), "suma_totales": suma,
            "pedido_mayor": mayor["total"], "clientes_unicos": clientes}

# Pruebas
p1 = crear_pedido("Ana", ("Laptop", 1000, 1), envio="express", descuento=50)
p2 = crear_pedido("Carlos", ("Teclado", 80, 1), regalo=True)
print(resumen_pedidos(p1, p2))
```

Solucion I2:

```
def validar_datos(**campos):
    validos = {}
    errores = {}
    extras = {}

    validaciones = {
        "nombre": lambda v: isinstance(v, str) and len(v.strip()) > 0,
        "edad": lambda v: isinstance(v, int) and v > 0,
        "email": lambda v: isinstance(v, str) and "@" in v
    }

    mensajes_error = {
        "nombre": "Debe ser string no vacio",
        "edad": "Debe ser entero positivo",
        "email": "Debe contener @"
    }

    for campo, valor in campos.items():
        if campo in validaciones:
            if validaciones[campo](valor):
                validos[campo] = valor
            else:
                errores[campo] = mensajes_error[campo]
        else:
            extras[campo] = valor

    return {"validos": validos, "errores": errores, "extras": extras}

# Pruebas
resultado = validar_datos(nombre="Ana", edad=-5, email="test",
                          telefono="123456", activo=True)
print(resultado)
# {'validos': {'nombre': 'Ana'}, 'errores': {'edad': '...', 'email': '...'},
#  'extras': {'telefono': '123456', 'activo': True}}
```

Solucion I3:

```
from datetime import datetime

def log(mensaje, *extras, nivel="INFO", timestamp=True, **metadata):
    partes = []

    # Nivel
    partes.append(f"[{nivel}]")

    # Timestamp
    if timestamp:
        partes.append(f"[{datetime.now().strftime('%Y-%m-%d %H:%M:%S')}]")

    # Mensaje principal
    partes.append(mensaje)

    # Extras
    if extras:
        partes.append(f"- Extras: {extras}")

    # Metadata
    if metadata:
        meta_str = ", ".join(f"{k}={v}" for k, v in metadata.items())
        partes.append(f"| {meta_str}")

    return " ".join(partes)

# Pruebas
print(log("Usuario conectado"))
print(log("Error critico", "codigo: 500", nivel="ERROR", usuario="admin"))
print(log("Debug info", nivel="DEBUG", timestamp=False, modulo="auth"))
```

Solucion I4:

```
def filtrar(coleccion, **criterios):
    resultados = []

    for item in coleccion:
        cumple = True
        for campo, valor in criterios.items():
            if campo not in item or item[campo] != valor:
                cumple = False
                break
        if cumple:
            resultados.append(item)

    return resultados

# Pruebas
usuarios = [
    {"nombre": "Ana", "edad": 25, "activo": True},
    {"nombre": "Luis", "edad": 30, "activo": True},
    {"nombre": "Maria", "edad": 25, "activo": False},
    {"nombre": "Carlos", "edad": 25, "activo": True}
]

print(filtrar(usuarios, edad=25))  # Ana, Maria, Carlos
print(filtrar(usuarios, edad=25, activo=True))  # Ana, Carlos
```

Solucion I5:

```
def crear_multiplicador(factor=2):
    def multiplicar(numero):
        return numero * factor
    return multiplicar

# Pruebas
doble = crear_multiplicador(2)
triple = crear_multiplicador(3)
por_diez = crear_multiplicador(10)

print(doble(5))     # 10
print(triple(5))    # 15
print(por_diez(5))  # 50

# Usando el default
duplicar = crear_multiplicador()  # factor=2
print(duplicar(7))  # 14
```

Soluciones Nivel Avanzado

Solucion A1:

```
def contador(funcion):
    llamadas = [0]  # Usamos lista para poder modificar en closure

    def wrapper(*args, **kwargs):
        llamadas[0] += 1
        print(f"Llamada #{llamadas[0]}")
        return funcion(*args, **kwargs)

    return wrapper

@contador
def saludar(nombre):
    return f"Hola, {nombre}"

@contador
def sumar(a, b):
    return a + b

# Pruebas
print(saludar("Ana"))    # Llamada #1, Hola, Ana
print(saludar("Luis"))   # Llamada #2, Hola, Luis
print(sumar(3, 5))       # Llamada #1, 8 (contador separado)
```

Solucion A2:

```
def repetir(veces=2):
    def decorador(funcion):
        def wrapper(*args, **kwargs):
            resultado = None
            for _ in range(veces):
                resultado = funcion(*args, **kwargs)
            return resultado
        return wrapper
    return decorador

@repetir(veces=3)
def saludar(nombre):
    print(f"Hola, {nombre}")
    return f"Saludo a {nombre}"

@repetir()  # Usa default veces=2
def despedir(nombre):
    print(f"Adios, {nombre}")

# Pruebas
saludar("Ana")     # Imprime "Hola, Ana" 3 veces
despedir("Luis")   # Imprime "Adios, Luis" 2 veces
```

Solucion A3:

```
def crear_sistema():
    return {"plugins": {}}

def registrar_plugin(sistema, nombre, funcion):
    sistema["plugins"][nombre] = funcion
    print(f"Plugin '{nombre}' registrado")

def ejecutar_plugin(sistema, nombre, *args, **kwargs):
    if nombre not in sistema["plugins"]:
        return f"Error: Plugin '{nombre}' no encontrado"
    return sistema["plugins"][nombre](*args, **kwargs)

def listar_plugins(sistema):
    return list(sistema["plugins"].keys())

def ejecutar_todos(sistema, *args, **kwargs):
    resultados = {}
    for nombre, funcion in sistema["plugins"].items():
        try:
            resultados[nombre] = funcion(*args, **kwargs)
        except Exception as e:
            resultados[nombre] = f"Error: {e}"
    return resultados

# Pruebas
sistema = crear_sistema()

# Registrar plugins
registrar_plugin(sistema, "saludar",
                 lambda nombre: f"Hola, {nombre}")
registrar_plugin(sistema, "despedir",
                 lambda nombre: f"Adios, {nombre}")
registrar_plugin(sistema, "sumar",
                 lambda *nums: sum(nums))

# Usar plugins
print(ejecutar_plugin(sistema, "saludar", "Ana"))  # Hola, Ana
print(ejecutar_plugin(sistema, "sumar", 1, 2, 3))  # 6
print(listar_plugins(sistema))  # ['saludar', 'despedir', 'sumar']
```