📝 Apuntes de Programación: Las Funciones en Python

Las funciones son bloques de código nombrados que realizan una tarea específica. Son esenciales para la modularidad, la reutilización del código y la legibilidad.

1. Declaración y Ejecución Básica

A. Definición Simple

Se definen con la palabra clave `def`. No requieren parámetros, pero deben ser invocadas con su nombre y paréntesis.

```
def miFuncion(): #* Se define la función sin parámetros
    print("Hola soy una función")

miFuncion() #* Se llama a la función
```

B. Funciones con Parámetros y Ámbito

Una función puede recibir datos de entrada, llamados parámetros.

```
def funcionConParametro(mensaje): # 'mensaje' es el parámetro
    valor = 5 # Esta es una variable local, solo existe dentro de la función
    print(mensaje, valor)

funcionConParametro("Hola Mundo")
```

- **Ámbito (Scope):** Las variables creadas dentro de la función (como `valor = 5`) son **locales** y no afectan a las variables fuera de ella.

2. Retorno de Valores (`return`)

La sentencia `return` finaliza la ejecución de la función y devuelve un resultado al punto donde fue llamada.

A. Retorno de un Valor Único

```
def valorRetorno(nombre, despedida):
    # Concatena y devuelve la cadena resultante
    return "Hola " + nombre + despedida

nombre = "José María"
print(valorRetorno(nombre, " Que te vaya bien"))
```

B. Retorno de Múltiples Valores

Python permite que una función devuelva múltiples valores. En realidad, devuelve una **tupla**, la cual puede ser desempaquetada inmediatamente.

```
def devolveNumeros():
    return 1, 2, 3 # Esto devuelve una tupla: (1, 2, 3)

n1, n2, n3 = devolveNumeros() # Desempaquetado de la tupla en tres variables
print(n1, n2, n3, sep="-") # Salida: 1-2-3
```

3. Parámetros y Tipos de Datos (Paso por Valor vs. Referencia)

Cuando se pasa una variable a una función, Python utiliza diferentes mecanismos según el tipo de dato.

A. Paso por Valor (Tipos Inmutables)

Para tipos inmutables (como números enteros, cadenas, tuplas), la función recibe una **copia** del valor. Cualquier cambio dentro de la función no afecta a la variable original.

```
def funcion(valor):
    valor *= 5 # Modifica la copia local
    print(valor)

n = 2
funcion(n) # n se pasa por valor
print(n) # n sigue siendo 2 (el original)
```

B. Paso por Referencia (Tipos Mutables)

Para tipos mutables (como **listas** o **diccionarios**), la función recibe una **referencia** al objeto. Los cambios hechos dentro de la función SÍ afectan al objeto original.

```
def funcion2(valor):
    valor[0] *= 5 # Modifica el objeto de la lista original
    print(valor)

n = [2]
funcion2(n) # n se pasa por referencia
print(n) # n ahora es [10] (modificado)
```

4. Argumentos Flexibles

A. Parámetros por Defecto

Puedes asignar un valor por defecto a un parámetro. Si se omite al llamar a la función, se usa el valor predeterminado.

```
def valorRetorno2(nombre, despedida = " te veo pronto!"): # Valor por defecto
    return "Hola " + nombre + despedida

print(valorRetorno2("Antonio")) # Usa el valor por defecto
```

B. Argumentos Arbitrarios (`*args`)

El asterisco `*` permite que una función acepte un número variable de argumentos (de cero a muchos), los cuales son recogidos en una **tupla** llamada `nombres` (o como la llames).

```
def muestraProfe(*nombres): # *nombres es una tupla con todos los argumentos
    print(nombres)

muestraProfe("Natalia", "Agustín") # Recoge: ('Natalia', 'Agustín')
```

C. Desempaquetado de Argumentos

El asterisco `*` también se usa para desempaquetar una lista o tupla, pasando sus elementos como argumentos individuales a una función.

```
def repitreNombre(veces, nombre):
    for _ in range(veces):
        print(nombre, end=" *** ")

datos = [2, "pepe"]
repitreNombre(*datos) # Se traduce a: repitreNombre(2, "pepe")
```

5. Consejos para Mejorar la Programación con Funciones (Examen Pro-Tip)

🎯 **Principio 1: Unicidad de Tareas (Single Responsibility)**

- **Regla de oro:** Una función debe hacer **una sola cosa** y hacerla bien.
- **Beneficio:** Facilita la prueba (testing), la depuración (debugging) y la reutilización del código.

🎯 **Principio 2: Anotaciones de Tipo (Type Hinting)**

- Aunque Python no obliga a usarlas (es un lenguaje de tipado dinámico), las anotaciones de tipo (`:int` , `-> str`) mejoran la claridad y son una señal de código profesional.
- **Sintaxis:** `def funcion(argumento: tipo_esperado) -> tipo_de_retorno:`

```
def prueba(arg1: int, arg2: float) -> str:
    # Aquí se indica que espera un int y un float, y devuelve un str
    return "Hola"
```

- **Consejo de Examen:** Si te piden código legible y robusto, usa anotaciones de tipo.

🎯 **Principio 3: Usar `**kwargs` para Diccionarios**

- De manera similar a `*args` , el doble asterisco `**kwargs` se utiliza para recibir un número variable de **argumentos de palabra clave** (keyword arguments). Los recoge en un **diccionario**.

```
def configurar_perfil(**opciones):
    print(opciones)
```

```
configurar_perfil(tema='oscuro', notificaciones=True, idioma='es')
# Recoge: {'tema': 'oscuro', 'notificaciones': True, 'idioma': 'es'}
```