Conferencia 6: Funciones en Python

Objetivos de la Sesión:

1. Comprender que las funciones son la herramienta principal para la abstracción y el encapsulamiento.
2. Aplicar el principio de diseño DRY (Don't Repeat Yourself) para escribir código más limpio y mantenible.
3. Identificar y describir los componentes de una función: signatura, parámetros y valor de retorno.
4. Asimilar el concepto de ámbito o scope, diferenciando variables locales y globales.
5. Implementar funciones flexibles mediante el uso de parámetros con valores por defecto y argumentos variables *args.
6. Analizar cómo la descomposición funcional en un algoritmo de ordenación demuestra los principios de diseño de software.

---

1. El Problema: Complejidad y Repetición

En el desarrollo de software, nos enfrentamos constantemente a dos enemigos: la complejidad y la repetición. Imagina que estamos construyendo un sistema de análisis de datos y necesitamos una lógica para validar y limpiar listas de números (eliminar valores no válidos y outliers).

Podríamos necesitar esta misma lógica en partes muy diferentes de nuestra aplicación:

- Al procesar los datos de sales diarias.
- Al analizar las lecturas de un sensor de temperatura.

Sin un mecanismo de reutilización, nuestro código podría verse así:

```
# Parte 1: Procesamiento de datos de sales
daily_sales = [150.5, 162.0, -1, 148.0, 5000.0, 155.3] # Datos con errores
valid_sales = []
for sale in daily_sales:
    # Lógica de validación: debe ser un número positivo y no un extremo
    if isinstance(sale, (int, float)) and 0 < sale < 1000:
        valid_sales.append(sale)
print(f"Ventas procesadas: {valid_sales}")


# Parte 2: En otra sección del programa, análisis de sensores
sensor_readings = [22.1, 23.5, 22.3, "error", 99.0, 21.9] # Datos con errores
valid_readings = []
for reading in sensor_readings:
    # Lógica de validación: debe ser un número y estar en un rango plausible
    if isinstance(reading, (int, float)) and -10 < reading < 50:
        valid_readings.append(reading)
print(f"Lecturas procesadas: {valid_readings}")
```

Este enfoque es problemático por varias razones:

- Violación del principio DRY (Don't Repeat Yourself): La estructura lógica de "recorrer una lista, aplicar una condición y añadir a una nueva lista" se repite. Si bien un ciclo for resuelve la repetición dentro de un solo bloque, no nos ayuda a reutilizar la lógica completa en diferentes contextos.
- Dificultad de mantenimiento: Si descubrimos un error en la lógica de validación o queremos mejorarla, debemos encontrar y corregir cada copia de ese código, lo cual es propenso a errores.
- Baja legibilidad: El código principal se ve abarrotado con detalles de implementación (el `for`, el `if`, el `append`), lo que dificulta entender el objetivo general a simple vista.

Para combatir esto, utilizamos las funciones, que son nuestra principal herramienta para la abstracción y el encapsulamiento.

- Abstracción: Es el arte de ocultar la complejidad. Nos permite usar una herramienta sin necesidad de entender sus mecanismos internos. Cuando manejas un carro, aprietas el acelerador (la interfaz) sin pensar en la inyección de combustible o la combustión interna (la implementación).
- Encapsulamiento: Es la práctica de agrupar el código (datos y operaciones) en una unidad autónoma, generalmente reutilizable, y proteger sus detalles internos del mundo exterior.

Una función encapsula una lógica y nos proporciona una abstracción simple para utilizarla.

---

2. Anatomía de una Función: La Solución Elegante

Vamos a refactorizar el ejemplo anterior encapsulando la lógica en una función.

```
# 1. Definición de la función (El "cómo" se implementa)
def clean_list(raw_data, min_val, max_val):
    clean_data = []
    for data in raw_data:
        if isinstance(data, (int, float)) and min_val < data < max_val:
            clean_data.append(data)

    # 3. Valor de retorno
    return clean_data

# 4. Llamada a la función (El "qué" queremos hacer)
daily_sales = [150.5, 162.0, -1, 148.0, 5000.0, 155.3]
valid_sales = clean_list(daily_sales, min_val=0, max_val=1000)
print(f"sales procesadas: {valid_sales}")

sensor_readings = [22.1, 23.5, 22.3, "error", 99.0, 21.9]
valid_readings = clean_list(sensor_readings, min_val=-10, max_val=50)
print(f"Lecturas procesadas: {valid_readings}")
```

Signatura de la Función

La primera línea es la signatura: `def clean_list(datos_brutos, min_val, max_val):`.

- `def`: Palabra clave que inicia la definición.
- `clean_list`: Un nombre descriptivo que sigue la convención snake_case.
- (`raw_data`, `min_val`, `max_val`): Los parámetros, que actúan como "puertas de entrada" para los datos que la función necesita para operar.

Cuerpo y Ámbito (Scope): El Principio de Encapsulamiento

El código indentado es el cuerpo de la función. Las variables definidas aquí, como datos_limpios y dato, son variables locales.

El ámbito (scope) es el mecanismo que refuerza el encapsulamiento. Estas variables locales solo existen dentro de la función mientras se está ejecutando. Una vez que la función termina, desaparecen. Esto es de interés porque:

- Evita efectos secundarios: El interior de la función es una "caja negra". No puede modificar accidentalmente variables del exterior (a menos que se lo indiquemos explícitamente).
- Previene colisiones de nombres: Podemos usar nombres de variables comunes como dato o i dentro de muchas funciones sin que interfieran entre sí.

Valor de Retorno (return)

La instrucción `return` es la "puerta de salida" de la función. Finaliza su ejecución y envía un valor de vuelta al código que la llamó. Si no hay `return`, la función devuelve implícitamente `None`.

---

3. Parametrización Avanzada

Python ofrece mecanismos para definir funciones con una mayor flexibilidad en sus argumentos.

Parámetros con Valores por Defecto

Es posible asignar un valor por defecto a uno o más parámetros en la signatura. Esto los convierte en opcionales durante la llamada a la función.

```
def format_name(name, last_names, second_name_first_letter=None):
    if second_name_first_letter:
        return f"{name} {second_name_first_letter}. {last_names}"
    else:
        return f"{name} {last_names}"

# Llamada sin el parámetro opcional
print(format_name("Yanesley", "Sánchez Rodríguez"))

# Llamada proveyendo el parámetro opcional
print(format_name("Jardiel", "Perez Rodríguez", "A"))
```

Nota importante: Los parámetros con valores por defecto deben declararse siempre después de los parámetros sin valor por defecto.

Argumentos Posicionales Variables (`*args`)

Para funciones que necesitan aceptar un número indeterminado de argumentos posicionales, se utiliza el prefijo `*`. Python agrupará todos los argumentos posicionales adicionales en una tupla.

```python
def calc_product(*factors):
    """Calcula el producto de una cantidad arbitraria de números."""
    if not factors:
        return 0
    product = 1
    for f in factors:
        product *= f
    return product

print(calc_product(2, 3))       # Salida: 6
print(calc_product(1, 5, 2, 3)) # Salida: 30
```

---

4. Aplicación: Algoritmo de Ordenación por Selección

La **ordenación por selección** (Selection Sort) es un algoritmo intuitivo que funciona de la siguiente manera:

Lógica del Algoritmo:

1. Se recorre la lista para encontrar el elemento de menor valor.
2. Se intercambia dicho elemento con el que ocupa la primera posición. La primera posición ahora contiene el valor correcto y queda "fija".
3. Se repite el proceso para el resto de la lista (desde la segunda posición en adelante), buscando el siguiente valor más bajo e intercambiándolo con el elemento en la segunda posición.
4. Este proceso continúa hasta que toda la lista esté ordenada.

Implementación:

```python
# Aunque siempre busquemos en el próximo código hasta el final de la lista
# damos la posibilidad de reutilizar el código de una forma más general.
def find_min_index(l,a,b):
    indice_min = a
    for i in range(a+1, b):
        if l[i] < l[indice_min]:
            indice_min = i
    return indice_min

def selection_sort(l):
    for i in range(len(l)):
        # 1. Encontrar el índice del mínimo en lo que queda de lista.
        p = find_min_index(l, i, len(l))

        # 2. Intercambiar este índice encontrado con la posición actual.
        l[i], l[p] = l[p], l[i]

    # Aquí se omite el return ya que no es necesario.


# --- Ejecución del código ---
my_list = [64, 25, 12, 22, 11, 90]
print(f"Lista sin ordenar: {my_list}")
```

```
selection_sort(my_list)
print(f"Lista ordenada: {my_list}")
```

Beneficios de esta Descomposición Funcional

- **Encapsulamiento y Responsabilidad Única**: La función `find_min_index` hace una sola cosa y la hace bien. Su lógica interna está completamente aislada. Si quisiéramos cambiar cómo busca el mínimo (por ejemplo, para manejar tipos de datos complejos), solo tendríamos que modificar esa función, y `selection_sort` seguiría funcionando sin cambios.
- **Abstracción y Legibilidad**: Al leer `selection_sort`, la lógica es clara: "para cada posición, encuentra el índice del mínimo y haz un intercambio". No nos perdemos en los detalles del bucle `for` de la búsqueda. Esto nos permite pensar sobre el problema a un nivel más alto.
- **Reusabilidad**: La función `find_min_index` es ahora una herramienta genérica. Podríamos usarla en cualquier otro lugar de nuestro código que necesite encontrar el elemento más pequeño en una porción de una lista.

Este ejemplo demuestra que el objetivo principal de las funciones no es solo evitar la repetición, sino gestionar la complejidad mediante la creación de bloques de construcción abstractos y encapsulados.

Resumen

Fundamentos del diseño de funciones en Python:

- **Propósito:** Las funciones son la principal herramienta para la abstracción, la reutilización de código y la reducción de la complejidad.
- **Estructura:** Hemos definido la signatura, los parámetros, el cuerpo y la sentencia `return`.
- **Ámbito (Scope):** Se introdujo el concepto de variables locales y su ciclo de vida dentro de la función.
- **Flexibilidad:** Se presentaron los parámetros por defecto y el mecanismo `*args` para manejar un número variable de argumentos.
- **Aplicación:** Se demostró cómo la descomposición de un problema en funciones más pequeñas mejora la calidad y legibilidad del código a través del ejemplo de ordenación por selección.

Ejercicio propuesto: Modificar la función `selection_sort` para que acepte un segundo parámetro booleano, `ascend=True`. Si este parámetro se establece en `False`, la función deberá ordenar la lista en orden descendente.