Capítulo 6: Funciones - Creando Bloques de Código Reutilizables

6.1 ¿Por Qué Necesitamos Funciones? El Principio DRY

A medida que nuestros programas crecen en tamaño y complejidad, comenzamos a notar patrones. A menudo, encontramos que necesitamos realizar la misma secuencia de operaciones en diferentes partes de nuestro código. Podríamos simplemente copiar y pegar esas líneas cada vez que las necesitemos, pero esta práctica tiene serios inconvenientes:

1. **Duplicación de Código:** Si hay un error en la lógica copiada o si necesitamos cambiarla, tendríamos que encontrar y modificar cada copia, lo cual es tedioso y propenso a errores.
2. **Menor Legibilidad:** El código se vuelve más largo y difícil de seguir.
3. **Mantenimiento Complicado:** Actualizar el código se convierte en una pesadilla.

Para combatir esto, existe un principio fundamental en la programación llamado **DRY: "Don't Repeat Yourself"** (No te Repitas). Las **funciones** son la herramienta principal de Python para adherirse a este principio.

Una **función** es un bloque de código con nombre, diseñado para realizar una tarea específica. Una vez definida, podemos "llamarla" (o invocarla) por su nombre tantas veces como sea necesario, desde cualquier parte de nuestro programa.

Beneficios de usar funciones:

- **Reutilización:** Escribes la lógica una vez y la usas múltiples veces.
- **Modularidad:** Permiten descomponer un programa complejo en piezas más pequeñas, lógicas y manejables. Cada función se encamina a una tarea concreta.
- **Legibilidad:** Un programa bien estructurado con funciones es más fácil de leer y entender. En lugar de un bloque extenso de código, ves una llamada a una función con un nombre descriptivo (ej: `calcular_impuesto(precio)`).
- **Mantenimiento:** Si necesitas corregir o mejorar una tarea específica, solo tienes que modificar el cuerpo de la función correspondiente.

Piensa en una función como una receta de cocina bien definida (ej: `preparar_salsa_bechamel`) o una herramienta especializada en una caja de herramientas (ej: `destornillador_estrella`).

6.2 Definiendo tu Primera Función

La sintaxis para definir una función en Python es la siguiente:

Código Python:

```
def nombre_de_la_funcion():
    # --- Inicio del bloque de la función ---
    # Código indentado que pertenece a la función
    instruccion_1
    instruccion_2
    # ...
    # --- Fin del bloque de la función ---
```

Componentes de la Definición:

1. `def`: Palabra clave que indica el inicio de la definición de una función.
2. `nombre_de_la_funcion`: El nombre que le das a tu función. Debe seguir las mismas reglas que los nombres de variables (letras, números, guion bajo; no empezar con número). La convención es usar `snake_case` (minúsculas y guiones bajos), por ejemplo: `calcular_total`, `imprimir_saludo`.
3. **Paréntesis** `()`: Obligatorios después del nombre. Más adelante veremos que aquí es donde se definen los parámetros que la función puede recibir.
4. **Dos Puntos** `:`: Indican el final de la "cabecera" de la función y el inicio de su cuerpo.
5. **Cuerpo de la Función**: El bloque de código **indentado** (generalmente 4 espacios) que contiene las instrucciones que la función ejecutará cuando sea llamada.

Si defines una función pero aún no sabes qué código irá dentro, puedes usar la palabra clave `pass` para evitar un error de sintaxis por un bloque vacío:

Código Python:

```
def funcion_pendiente():
    pass # No hace nada, pero es válido
```

6.3 Llamando (Invocando) una Función

Definir una función es como escribir la receta; no hace nada hasta que la "preparas" o, en términos de programación, hasta que la **llamas** (o invocas).

Para llamar a una función, simplemente escribes su nombre seguido de paréntesis:

Código Python:

```python
def saludar():
    print("¡Hola desde la función saludar!")
    print("Que tengas un buen día.")

# Llamamos a la función
print("Antes de llamar a saludar.")
saludar() # Se ejecuta el código dentro de saludar()
print("Después de llamar a saludar.")

# Podemos llamarla múltiples veces
saludar()
```

6.4 Pasando Información: Parámetros y Argumentos

La mayoría de las funciones necesitan recibir información del exterior para poder realizar su tarea de manera flexible. Por ejemplo, una función que suma dos números necesita saber *cuáles* son esos dos números.

6.4.1 Parámetros vs. Argumentos

- **Parámetros**: Son las variables que se listan dentro de los paréntesis en la **definición** de la función. Actúan como "espacios reservados" o variables locales dentro de la función, que tomarán los valores que se les pasen cuando la función sea llamada.
- **Argumentos**: Son los valores reales que se **pasan** a la función cuando se la **llama**. Estos valores se asignan a los parámetros correspondientes.

6.4.2 Argumentos Posicionales

Cuando llamas a una función, los argumentos se asignan a los parámetros según su **posición** (orden). El primer argumento se asigna al primer parámetro, el segundo al segundo, y así sucesivamente.

Código Python:

```python
def presentar_persona(nombre, edad): # 'nombre' y 'edad' son
parámetros
    print(f"Me llamo {nombre} y tengo {edad} años.")

# "Ana" es el argumento para 'nombre', 28 para 'edad'
presentar_persona("Ana", 28)

# "Luis" es el argumento para 'nombre', 35 para 'edad'
presentar_persona("Luis", 35)

# Si el número de argumentos no coincide con el de parámetros, Python
da un TypeError
# presentar_persona("Carlos") # TypeError: presentar_persona()
missing 1 required positional argument: 'edad'
```

6.4.3 Argumentos Nombrados (Keyword Arguments)

También puedes pasar argumentos especificando el nombre del parámetro al que quieres asignar el valor, usando la sintaxis `nombre_parametro=valor`.

- La principal ventaja es que **el orden de los argumentos nombrados no importa**.
- Hacen que la llamada a la función sea más explícita y legible, especialmente si la función tiene muchos parámetros.

Código Python:

```python
def configurar_opciones(color, tamaño, activado):
    print(f"Color: {color}, Tamaño: {tamaño}, Activado: {activado}")

# Usando argumentos nombrados
configurar_opciones(color="rojo", tamaño="grande", activado=True)
configurar_opciones(activado=False, color="azul", tamaño="pequeño") #
El orden no importa

# Puedes mezclar argumentos posicionales y nombrados,
# pero los posicionales DEBEN ir primero.
configurar_opciones("verde", activado=True, tamaño="mediano") #
'verde' es posicional para 'color'
```

```
# configurar_opciones(color="negro", True, tamaño="mini") #
SyntaxError: positional argument follows keyword argument
```

6.5 Devolviendo Valores: La Sentencia `return`

6.5.1 El Propósito de `return`

Las funciones no solo pueden ejecutar acciones (como imprimir en pantalla), sino que también pueden **calcular un resultado y devolverlo** al código que las llamó. Este valor devuelto puede ser asignado a una variable, usado en una expresión, o pasado como argumento a otra función. La sentencia `return` se encarga de esto.

6.5.2 Sintaxis y Comportamiento

Código Python:

```
def sumar(a, b):
    resultado = a + b
    return resultado # La función termina aquí y devuelve el valor de
'resultado'

# El valor devuelto por sumar(5, 3) se asigna a 'mi_suma'
mi_suma = sumar(5, 3)
print(f"La suma es: {mi_suma}") # Salida: 8

print(f"El doble de sumar(10, 2) es: {sumar(10, 2) * 2}") # Salida:
24
```

Cuando se ejecuta una sentencia `return`, la función **termina inmediatamente**, y cualquier código que siga después del `return` dentro de la función no se ejecutará.

6.5.3 Devolver `None`

Si una función llega al final de su bloque sin encontrar una sentencia `return`, o si tiene una sentencia `return` sin ningún valor a su lado (ej: `return` solo), la función devuelve automáticamente un valor especial llamado **`None`**. `None` es un tipo de dato en Python (`NoneType`) que representa la ausencia de un valor.

Código Python:

```python
def saludar_sin_retorno(nombre):
    print(f"Hola, {nombre}")
    # No hay 'return' explícito

valor_devuelto = saludar_sin_retorno("Mundo")
print(f"La función saludar_sin_retorno devolvió: {valor_devuelto}") #
Salida: None
```

6.5.4 Múltiples Sentencias `return`

Una función puede tener varias sentencias `return`, típicamente en diferentes ramas de una estructura condicional (`if/elif/else`). Sin embargo, tan pronto como se ejecuta una de ellas, la función termina.

Código Python:

```python
def obtener_valor_absoluto(numero):
    if numero < 0:
        return -numero
    else: # o simplemente 'return numero' sin el else, si es la
última línea
        return numero

print(f"Absoluto de -10: {obtener_valor_absoluto(-10)}") # Salida: 10
print(f"Absoluto de 7: {obtener_valor_absoluto(7)}")   # Salida: 7
```

6.5.5 Devolver Múltiples Valores

Python permite que una función devuelva múltiples valores separándolos por comas en la sentencia `return`. En realidad, lo que Python hace es empaquetar estos valores en una **tupla** y devolver esa tupla.

Código Python:

```python
def obtener_info_usuario():
    nombre = "Beatriz"
    edad = 42
    ciudad = "Madrid"
```

```
    return nombre, edad, ciudad # Devuelve la tupla ("Beatriz", 42,
"Madrid")

info_completa = obtener_info_usuario()
print(f"Información completa (tupla): {info_completa}")
print(f"Tipo de info_completa: {type(info_completa)}")

# Podemos desempaquetar la tupla directamente al recibir el retorno
nombre_u, edad_u, ciudad_u = obtener_info_usuario()
print(f"Nombre: {nombre_u}, Edad: {edad_u}, Ciudad: {ciudad_u}")
```

6.6 Documentando tus Funciones: Docstrings

Es una práctica fundamental en programación documentar tu código para que otros (¡y tu "yo" del futuro!) puedan entender qué hace, cómo usarlo y qué esperar de él. Para las funciones en Python, esto se hace mediante **docstrings** (cadenas de documentación).

Un docstring es un string literal (generalmente multilínea, usando triples comillas `"""..."""` o `'''...'''`) que se coloca como la **primera sentencia ejecutable** dentro de la definición de una función.

Código Python:

```
def calcular_imc(peso_kg, altura_m):
    """
    Calcula el Índice de Masa Corporal (IMC).

    El IMC se calcula como el peso en kilogramos dividido por
    la altura en metros al cuadrado.

    Args:
        peso_kg (float): El peso de la persona en kilogramos.
        altura_m (float): La altura de la persona en metros.

    Returns:
        float: El IMC calculado.
        str: Un mensaje de error si la altura es cero o negativa.
    """
    if altura_m <= 0:
```

```
        return "Error: La altura debe ser un valor positivo."
    imc = peso_kg / (altura_m ** 2)
    return imc
```

¿Qué incluir en un docstring?

- Una breve descripción de lo que hace la función (el propósito).
- Descripción de los argumentos (`Args`): nombre del parámetro, tipo esperado (opcional pero útil), y qué representa.
- Descripción de lo que retorna la función (`Returns`): tipo de dato devuelto y qué representa.

¿Cómo acceder a un docstring?

- Usando la función `help()`: `help(calcular_imc)`
- Accediendo al atributo especial `__doc__`: `print(calcular_imc.__doc__)`

6.7 El Alcance de las Variables (Scope)

El **alcance** (o *scope*) de una variable define la parte del programa donde esa variable es reconocida y puede ser utilizada.

6.7.1 Variables Locales

Las variables creadas **dentro** de una función se llaman **variables locales**. Solo "viven" y son accesibles desde el interior de esa función. Una vez que la función termina su ejecución, sus variables locales desaparecen (son destruidas).

Código Python:

```
def funcion_con_local():
    mensaje_local = "Soy una variable local."
    print(mensaje_local)

funcion_con_local()
# print(mensaje_local) # NameError: name 'mensaje_local' is not
defined (no existe fuera de la función)
```

6.7.2 Variables Globales

Las variables definidas **fuera** de todas las funciones, en el nivel principal del script, se llaman **variables globales**.

- **Lectura:** Por defecto, las funciones pueden *leer* los valores de las variables globales.
- **Modificación:** Si intentas *asignar un nuevo valor* a una variable global desde dentro de una función, Python, por defecto, creará una *nueva variable local* con el mismo nombre, "ocultando" (o *sombreando*) la global dentro de esa función. La variable global original no se modificará.

Código Python:

```
contador_global = 10 # Variable global

def mostrar_contador():
    print(f"Dentro de mostrar_contador (leyendo global):
{contador_global}")

def intentar_modificar_contador():
    # Esto crea una NUEVA variable local 'contador_global', no
modifica la global
    contador_global = 99
    print(f"Dentro de intentar_modificar_contador (variable local):
{contador_global}")

mostrar_contador()                # Salida: 10
intentar_modificar_contador()     # Salida: 99
print(f"Fuera, contador_global sigue siendo: {contador_global}") #
Salida: 10 (no cambió)
```

6.7.3 La Palabra Clave `global`

Si realmente necesitas modificar una variable global desde dentro de una función (lo cual se debe hacer con precaución, ya que puede hacer el código más difícil de seguir), debes declararla explícitamente usando la palabra clave `global` al inicio de la función.

Código Python:

```python
contador_global_real = 50

def modificar_global_de_verdad():
    global contador_global_real # Avisa a Python que nos referimos a la global
    contador_global_real += 10
    print(f"Dentro de modificar_global_de_verdad: {contador_global_real}")

print(f"Antes de modificar: {contador_global_real}") # 50
modificar_global_de_verdad()                        # 60
print(f"Después de modificar: {contador_global_real}") # 60 (¡sí cambió!)
```

Recomendación: En general, es mejor que las funciones operen con sus parámetros y devuelvan resultados, en lugar de depender o modificar directamente variables globales. Esto las hace más independientes y reutilizables.

6.8 Flexibilidad con Parámetros por Defecto

Puedes asignar un **valor por defecto** a uno o más parámetros en la definición de una función. Si al llamar la función no se proporciona un argumento para ese parámetro, Python usará automáticamente el valor por defecto.

Regla Importante: En la definición de la función, todos los parámetros con valores por defecto deben ir **después** de cualquier parámetro que no tenga valor por defecto.

Código Python:

```python
def saludar_formal(nombre, titulo="Estimado/a", mensaje="Bienvenido/a al sistema."):
    """Genera un saludo formal."""
    print(f"{titulo} {nombre}, {mensaje}")

saludar_formal("Dr. López") # 'mensaje' toma su valor por defecto
saludar_formal("Ana García", mensaje="Es un placer tenerte aquí.") # 'titulo' usa el defecto
saludar_formal("Pedro", "Querido", "¡Qué bueno verte!") # Se proveen todos
```

Esto hace que las funciones sean más flexibles y fáciles de usar en diferentes situaciones sin necesidad de pasar siempre todos los argumentos.

6.9 ¡A Practicar con Funciones!

Vamos a consolidar estos conceptos. Crea un archivo `clase6_practica.py`.

Ejercicio 1: Área de un Rectángulo

- Define una función llamada `calcular_area_rectangulo` que acepte dos parámetros: `base` y `altura`.
- La función debe calcular el área (`base * altura`) y **retornar** el resultado.
- Añade un docstring apropiado.
- En la parte principal del script, pide al usuario que ingrese la base y la altura (conviértelos a `float`).
- Llama a la función con los valores ingresados y guarda el resultado en una variable.
- Imprime el área calculada de forma clara.

`Código Python:`

```
# Solución Ejercicio 1
print("--- Ejercicio 1: Área Rectángulo ---")
def calcular_area_rectangulo(base, altura):
    """Calcula y retorna el área de un rectángulo.

    Args:
        base (float): La longitud de la base del rectángulo.
        altura (float): La altura del rectángulo.

    Returns:
        float: El área calculada del rectángulo.
    """
    if base < 0 or altura < 0:
        return "Error: La base y la altura deben ser positivas."
    return base * altura

try:
    base_usuario = float(input("Introduce la base del rectángulo: "))
    altura_usuario = float(input("Introduce la altura del rectángulo:
"))
    area_calculada = calcular_area_rectangulo(base_usuario,
altura_usuario)
```

```
    if isinstance(area_calculada, str): # Verificamos si es el
mensaje de error
        print(area_calculada)
    else:
        print(f"El área de un rectángulo con base {base_usuario} y
altura {altura_usuario} es: {area_calculada:.2f}")
except ValueError:
    print("Entrada inválida. Por favor, introduce números.")
print("-" * 20)
```

Ejercicio 2: Verificar si un Número es Par

- Define una función `es_par(numero)` que acepte un número entero.
- La función debe retornar `True` si el número es par, y `False` si es impar.
- Añade un docstring.
- Pide al usuario un número entero.
- Llama a la función e imprime un mensaje como: "El número X es par." o "El número X es impar." usando un `if/else`.

Código Python:

```
# Solución Ejercicio 2
print("--- Ejercicio 2: ¿Es Par? ---")
def es_par(numero):
    """Verifica si un número entero es par.

    Args:
        numero (int): El número a verificar.

    Returns:
        bool: True si el número es par, False en caso contrario.
    """
    return numero % 2 == 0

try:
    num_ingresado = int(input("Introduce un número entero: "))
    if es_par(num_ingresado):
        print(f"El número {num_ingresado} es PAR.")
    else:
        print(f"El número {num_ingresado} es IMPAR.")
```

```
except ValueError:
    print("Entrada inválida. Por favor, introduce un número entero.")
print("-" * 20)
```

Ejercicio 3: Sumar Elementos de una Lista

- Define una función `sumar_lista(numeros)` que acepte una lista de números.
- La función debe usar un bucle `for` para sumar todos los elementos de la lista y **retornar** la suma total.
- Añade un docstring.
- Crea una lista de ejemplo con algunos números.
- Llama a la función con tu lista e imprime el resultado.

Código Python:

```
# Solución Ejercicio 3
print("--- Ejercicio 3: Sumar Elementos de una Lista ---")
def sumar_lista(lista_numeros):
    """Calcula la suma de todos los números en una lista.

    Args:
        lista_numeros (list): Una lista que contiene números (int o
float).

    Returns:
        int o float: La suma de los elementos de la lista. Devuelve 0
si la lista está vacía.
    """
    total = 0
    for numero in lista_numeros:
        total += numero
    return total

mi_lista_de_numeros = [10, 20, 30, 5.5, -2.5]
print(f"La lista es: {mi_lista_de_numeros}")
suma_de_lista = sumar_lista(mi_lista_de_numeros)
print(f"La suma de sus elementos es: {suma_de_lista}")

print(f"La suma de una lista vacía es: {sumar_lista([])}")
```

```
print("-" * 20)
```

6.10 Resumen del Capítulo y Próximos Pasos

¡Felicidades! Has aprendido uno de los conceptos más poderosos y fundamentales de la programación: las **funciones**.

- Hemos visto cómo **definir** funciones con `def`, su nombre, parámetros y cuerpo indentado.
- Cómo **llamarlas** y pasarles **argumentos** (posicionales y nombrados/keyword).
- La importancia de la sentencia `return` para que las funciones devuelvan resultados (y `None` si no hay `return` explícito).
- Cómo **documentar** funciones usando **docstrings** `"""..."""`.
- Los conceptos básicos de **alcance de variables** (local vs. global) y el uso (cauteloso) de la palabra clave `global`.
- Cómo añadir flexibilidad con **parámetros con valores por defecto**.

El uso efectivo de funciones es clave para escribir código **DRY** (No te Repitas), modular, legible y fácil de mantener.

En el **Capítulo 7**, abordaremos temas cruciales para estructurar proyectos más grandes y manejar situaciones inesperadas: **Módulos y Paquetes** (para organizar código en múltiples archivos), **Entornos Virtuales** (para aislar dependencias de proyectos), **Manejo de Errores** con `try-except` (para que nuestros programas no "crasheen" fácilmente) y operaciones básicas con **Archivos de Texto**.

Sugerencias para practicar:

1. **Factorial de un Número:** Crea una función `factorial(n)` que reciba un entero no negativo `n` y retorne su factorial (`n! = n * (n-1) * ... * 1`, y `0! = 1`). Usa un bucle.
2. **Verificador de Palíndromos:** Crea una función `es_palindromo(palabra)` que reciba un string y retorne `True` si `palabra` es un palíndromo (se lee igual al derecho y al revés, ej: "ana", "reconocer"), y `False` si no. (Pista: `palabra.lower()` para ignorar mayúsculas/minúsculas y `palabra[::-1]` para invertir el string pueden ser útiles).
3. **Búsqueda en Lista (Función):** Crea una función `buscar_elemento(mi_lista, elemento_a_buscar)` que reciba una lista y un elemento. La función debe retornar `True` si `elemento_a_buscar` se encuentra en `mi_lista`, y `False` en caso contrario. *Intenta implementarla usando un bucle `for` y una comparación*, no el operador `in` directamente (¡solo para practicar la lógica de bucles y funciones!).

¡Sigue construyendo funciones para las tareas que se repiten! Nos vemos en el próximo capítulo.