Módulo 3.1: Funciones

Prof. Carlos Cedeño

Las funciones son una parte esencial del lenguaje de programación Python: es posible que ya hayas encontrado y utilizado algunas de las muchas funciones integradas en el lenguaje Python o que vienen con su ecosistema de bibliotecas. Sin embargo, es necesario empezar a escribir tus propias funciones para poder resolver problemas mas complejos. Son fundamentales para escribir código modular, organizado y eficiente.

Crear una Función

Para crear una función en Python, usamos la palabra clave `def`, seguida del nombre de la función y paréntesis `()`. El cuerpo de la función se indenta.

```
def saludar():
  print("¡Hola, mundo!")

# Para usar la función, la llamamos por su nombre:
saludar()
```

Puntos Clave:

- Usa `def` para iniciar la definición de una función.
- El nombre de la función debe ser descriptivo y seguir las convenciones de nomenclatura (generalmente `snake_case` para funciones en Python).
- Los paréntesis `()` son obligatorios, incluso si la función no recibe ningún dato.
- El bloque de código dentro de la función debe estar indentado (generalmente 4 espacios).

Definir Parámetros

Los parámetros son variables que se utilizan para pasar información a una función. Se definen dentro de los paréntesis en la definición de la función.

```
def saludar_a(nombre):
  print(f"¡Hola, {nombre}!")

saludar_a("Ana")
saludar_a("Juan")
```

Parámetros por Defecto

Puedes asignar valores por defecto a los parámetros. Esto hace que el parámetro sea opcional cuando se llama a la función. Si no se proporciona un valor, se utilizará el valor por defecto.

```
def saludar_con_mensaje(nombre, mensaje="¿cómo estás?"):
  print(f"¡Hola, {nombre}! {mensaje}")

saludar_con_mensaje("Carlos")  # Usa el mensaje por defecto
saludar_con_mensaje("Elena", "espero que tengas un buen día.") # Proporciona un mensaje
```

Puntos Clave:

- Los parámetros se listan dentro de los paréntesis, separados por comas.
- Cuando llamas a la función, proporcionas **argumentos** para esos parámetros.
- Los parámetros por defecto se definen usando el signo igual (`=`) en la lista de parámetros.
- Los parámetros sin valor por defecto deben ir antes que los parámetros con valor por defecto.

Retornar Valores en Funciones

Una de las capacidades más importantes de las funciones es su habilidad para **devolver un valor** al lugar desde donde fueron llamadas. Esto permite que las funciones procesen datos y entreguen un resultado que puede ser utilizado en otras partes de tu programa. La sentencia `return` es la encargada de esta tarea.

La Sentencia `return`

Cuando Python encuentra una sentencia `return` dentro de una función, la ejecución de esa función finaliza inmediatamente y el valor especificado después de `return` es enviado de vuelta al código que llamó a la función.

```
def sumar(a, b):
  resultado = a + b
  return resultado # Devuelve el valor de la variable 'resultado'

# Llamamos a la función y almacenamos el valor devuelto
suma_total = sumar(5, 3)
print(f"El resultado de la suma es: {suma_total}") # Imprime: El resultado de la suma es: 8

# También podemos usar el valor devuelto directamente
print(f"10 + 20 es: {sumar(10, 20)}") # Imprime: 10 + 20 es: 30
```

Puntos Clave sobre `return`:

- Finaliza la ejecución de la función. Cualquier código dentro de la función después de una sentencia `return` que se ejecute no será alcanzado.
- Puede devolver cualquier tipo de dato: números, cadenas, listas, diccionarios, objetos, e incluso otras funciones.

Funciones que No Devuelven Nada Explícitamente (Devuelven `None`)

Si una función no tiene una sentencia `return`, o si tiene una sentencia `return` sin un valor especificado (por ejemplo, solo `return`), la función devuelve automáticamente un valor especial en Python llamado `None`. `None` se usa para representar la ausencia de un valor.

```python
def saludar(nombre):
  print(f"¡Hola, {nombre}!")
  # No hay sentencia 'return' explícita aquí

valor_devuelto = saludar("Ana")
print(f"La función 'saludar' devolvió: {valor_devuelto}")
# Imprime:
# ¡Hola, Ana!
# La función 'saludar' devolvió: None

def funcion_con_return_vacio():
  x = 10
  # Podría haber lógica aquí
  return # Sentencia return sin valor

resultado = funcion_con_return_vacio()
print(f"Función con return vacío devolvió: {resultado}")
# Imprime: Función con return vacío devolvió: None
```

Retornar Múltiples Valores

Python permite que una función retorne múltiples valores de una manera muy intuitiva. Simplemente listas los valores que quieres retornar después de la sentencia `return`, separados por comas. Python automáticamente empaqueta estos valores en una **tupla**.

```python
def obtener_informacion_usuario():
  nombre = "Carlos"
  edad = 30
  ciudad = "Madrid"
  return nombre, edad, ciudad # Devuelve tres valores

# Desempaquetado de la tupla devuelta
info_nombre, info_edad, info_ciudad = obtener_informacion_usuario()

print(f"Nombre: {info_nombre}")   # Imprime: Nombre: Carlos
print(f"Edad: {info_edad}")       # Imprime: Edad: 30
print(f"Ciudad: {info_ciudad}")   # Imprime: Ciudad: Madrid

# También puedes recibir la tupla completa
datos_completos = obtener_informacion_usuario()
print(f"Datos completos: {datos_completos}") # Imprime: Datos completos: ('Carlos', 30, 'Madrid')
print(type(datos_completos))                 # Imprime: <class 'tuple'>
```

Puntos Clave:

- Usa la palabra clave `return` seguida de los valores que deseas devolver, separados por comas.
- Python empaqueta los múltiples valores retornados en una tupla por defecto.
- Puedes desempaquetar los valores devueltos en variables individuales al llamar a la función.

¿Por qué es útil retornar valores?

- **Reutilización de Resultados:** Permite usar el resultado de una función en diferentes partes del código.
- **Composición de Funciones:** El resultado de una función puede ser la entrada de otra.
- **Pruebas:** Facilita la prueba de funciones, ya que puedes verificar si el valor devuelto es el esperado.
- **Claridad del Código:** Las funciones que devuelven valores explícitos suelen ser más fáciles de entender en cuanto a su propósito y salida.

No Permitir Usar Variables Externas en las Funciones (Buenas Prácticas y Ámbito)

Si bien Python técnicamente permite que las funciones accedan a variables definidas fuera de ellas (variables globales), **es una mala práctica** modificar variables globales directamente desde dentro de una función. Hace que el código sea más difícil de entender, depurar y mantener.

Las funciones deben ser, en la medida de lo posible, **autocontenidas y predecibles**. Esto significa que sus resultados deben depender idealmente solo de sus parámetros de entrada y no de variables externas "ocultas".

Ámbito de las Variables (Scope):

- **Variables Locales:** Son aquellas definidas *dentro* de una función. Solo son accesibles desde dentro de esa función.
- **Variables Globales:** Son aquellas definidas *fuera* de todas las funciones, en el ámbito principal del script.

```python
variable_global = 100

def mi_funcion_mal_diseno():
  # Esta función modifica una variable global, lo cual no es ideal.
  global variable_global # Necesario para modificarla, pero ¡cuidado!
  variable_global += 10
  print(f"Dentro de la función (mal diseño): {variable_global}")

def mi_funcion_buen_diseno(valor):
  # Esta función opera con un parámetro y devuelve un nuevo valor.
  nuevo_valor = valor + 10
  print(f"Dentro de la función (buen diseño): {nuevo_valor}")
  return nuevo_valor

print(f"Antes de llamar a las funciones: {variable_global}")

mi_funcion_mal_diseno()
print(f"Después de la función (mal diseño): {variable_global}")

variable_global = 100 # Reiniciamos para el buen diseño
variable_global = mi_funcion_buen_diseno(variable_global)
print(f"Después de la función (buen diseño): {variable_global}")
```

¿Cómo evitar el uso de variables externas?

1. **Pasar valores como parámetros:** Si una función necesita un valor, pásalo como un argumento.
2. **Retornar valores:** Si una función calcula algo que se necesita fuera, usa `return` para devolver ese valor.

Esto promueve la **pureza funcional** (aunque Python no la exige estrictamente) y hace que tus funciones sean más reutilizables y menos propensas a efectos secundarios inesperados.

Docstrings (Cadenas de Documentación)

Es una buena práctica incluir una cadena de documentación (docstring) justo después de la línea `def`. Describe lo que hace la función, sus parámetros y lo que retorna. Se encierran entre triples comillas (`"""Docstring aquí"""`).

```python
def calcular_area_rectangulo(ancho, alto):
  """
  Calcula el área de un rectángulo.

  Parámetros:
  ancho (int o float): El ancho del rectángulo.
  alto (int o float): El alto del rectángulo.

  Retorna:
  float: El área calculada del rectángulo.
  """
  if ancho < 0 or alto < 0:
    return "Las dimensiones deben ser positivas."
  return ancho * alto

print(calcular_area_rectangulo(5, 4))
help(calcular_area_rectangulo) # Muestra el docstring
```

Argumentos Posicionales y Nombrados (Keywords Arguments)

Cuando llamas a una función, puedes pasar argumentos por su posición o por su nombre.

```
def describir_persona(nombre, edad, ciudad="Desconocida"):
  print(f"{nombre} tiene {edad} años y vive en {ciudad}.")

# Argumentos posicionales
describir_persona("Laura", 30)

# Argumentos nombrados (keyword arguments)
# El orden no importa si usas los nombres
describir_persona(edad=25, nombre="Pedro")
describir_persona("Sofía", edad=22, ciudad="Madrid")
```