ChatGPT

Funciones en Python

Una **función** en Python es un bloque de código independiente diseñado para realizar una tarea específica. Al definirla, agrupamos instrucciones que podemos **invocar** ("llamar") varias veces sin reescribirlas [1] [2]. De hecho, definir funciones ayuda a organizar programas complejos, evitar la repetición de código y facilitar el mantenimiento. Por ejemplo, pensar en una función como una "receta": define los ingredientes (parámetros) y los pasos a seguir (cuerpo de la función) para obtener un resultado (valor de retorno) [2] [1]. Entre sus ventajas se cuentan:

- **Reutilización de código:** escribirlo una vez y usarlo muchas veces [3].
- **Legibilidad y organización:** dividir un problema grande en partes más pequeñas y descriptivas [3].
- **Facilidad de prueba (testing):** probar cada función aisladamente es más sencillo [3].
- **Mantenibilidad:** modificar código en un solo lugar actualiza todos los usos.

En resumen, las funciones son fragmentos de código reutilizables que mejoran la claridad y eficiencia de un programa [2] [1].

Definición e invocación de funciones

Para definir una función usamos la palabra clave `def` seguida del nombre de la función y paréntesis con sus parámetros, y luego un bloque indentado con el código a ejecutar. Por ejemplo:

```
def saludar():            # Definición de la función
    return "¡Hola!"       # Cuerpo y valor de retorno
```

La **invocación** o llamada a la función se hace usando su nombre seguido de paréntesis (con o sin argumentos según su definición). Por ejemplo, al ejecutar `print(saludar())` se llama a la función y se imprime su resultado. Es importante no olvidar los paréntesis al llamar a una función; sin ellos, estamos obteniendo el objeto función en sí, no su ejecución [4].

En resumen, definir una función es declararla con `def nombre_función(...)`, mientras que invocarla es ejecutarla mediante `nombre_función(...)` [4] [1].

Pasos básicos para definir una función en Python (resumidos de [25]):
1. Escribe `def nombre_función(parámetros):` [5].
2. Dentro del bloque indentado, agrega las instrucciones que realice la función [5].
3. (Opcional) Usa `return` para devolver un valor; si no usas `return`, la función devolverá `None` por defecto [6] [5].

Por ejemplo, estos tres pasos en conjunto crearían una función sencilla que saluda:

```
def saludo(nombre):            # 1. Definición con parámetro
    mensaje = "¡Hola, " + nombre + "!"
```

```
    return mensaje            # 2-3. Instrucciones y retorno de valor

print(saludo("Ana"))          # Invocación de la función -> ¡Hola, Ana!
```

Parámetros y argumentos en funciones

Una función puede recibir datos de entrada ("parámetros") para operar con ellos. Cuando llamamos a la función le pasamos **argumentos** que se asignan a esos parámetros [7]. Existen varios tipos de parámetros en Python:

- **Posicionales:** se pasan en el orden en que están definidos. Por ejemplo:

```
def area_rectangulo(ancho, alto):
    return ancho * alto

print(area_rectangulo(3, 4))  # 12
```

Aquí `ancho=3` y `alto=4` por posición [8]. El orden de los argumentos importa.

- **Por nombre (keyword):** se especifica explícitamente `parametro=valor` en la llamada, sin importar el orden. Ejemplo:

```
def crear_usuario(nombre, edad):
    return f"{nombre}, {edad} años"

print(crear_usuario(edad=30, nombre="Luis"))
```

Esto es equivalente a pasar `(nombre="Luis", edad=30)` [9].

- **Con valores por defecto:** algunos parámetros pueden tener valores predeterminados si no se proporcionan argumentos. Por ejemplo:

```
def saludar(nombre, saludo="Hola"):
    return f"{saludo}, {nombre}!"

print(saludar("María"))         # "Hola, María!"
print(saludar("Pedro", "Buenos días"))  # "Buenos días, Pedro!"
```

En este caso, `saludo="Hola"` se usa si no se especifica otro valor [10]. Nota: los parámetros con valores por defecto deben ir al final de la lista [11].

- **Colecciones (listas/diccionarios):** podemos pasar objetos mutables como listas o diccionarios. Por ejemplo, una lista entera:

```
def imprimir_lista(lista):
    for e in lista:
        print(e)

imprimir_lista([1, 2, 3])  # Imprime 1, 2, 3
```

O un diccionario completo como **argumento con nombre** usando `**`:

```
def imprimir_dicc(**datos):
    for clave, valor in datos.items():
        print(clave, ":", valor)

imprimir_dicc(nombre="Ana", edad=25)  # Imprime clave:valor del
diccionario
```

Python también ofrece `*args` para un número variable de parámetros posicionales y `**kwargs` para nombre variables; éstos agrupan los valores en una tupla o diccionario respectivamente [12] [13].

En todos los casos, al llamar a la función se entregan los **argumentos** que corresponden a los parámetros definidos, y el cuerpo de la función usa esos valores para su tarea [7] [9].

Valor de retorno

Una función puede devolver un valor mediante la sentencia `return`. Al ejecutarse `return`, la función termina y el valor proporcionado se devuelve al contexto llamador [14]. Por ejemplo:

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

resultado = sumar(5, 7)
print(resultado)  # 12
```

Si una función no incluye `return`, Python devuelve implícitamente `None` [6]. También pueden retornarse múltiples valores separándolos por comas; Python los empaqueta en una tupla, como en:

```
def stats(nums):
    return sum(nums), max(nums)

total, mayor = stats([10,20,30])
print(total, mayor)  # 60 30
```

En resumen, `return` se usa para producir un resultado desde la función; sin ella, el retorno será `None` [6]. Además, los resultados pueden ser cualquier tipo de objeto (número, cadena, lista, tupla, etc.) [15] [6].

Variables locales y globales

El **ámbito** (scope) de una variable indica dónde es accesible. En Python existen principalmente **variables locales** y **globales**.

- **Variables locales:** las definidas dentro de una función solo existen durante la ejecución de esa función [16] [17]. Por ejemplo, en:

```
def mi_funcion():
    x = 10        # x es local a mi_funcion
    print("Dentro:", x)

mi_funcion()
print("Fuera:", x)  # Error: x no existe aquí
```

  La `x` dentro de la función desaparece al terminar, y fuera de ella no existe [16].

- **Variables globales:** las definidas fuera de toda función son accesibles desde cualquier parte del módulo, incluso dentro de funciones (aunque si hay una local con el mismo nombre, **ésta oculta a la global** dentro de la función) [18] [17]. Por ejemplo:

```
y = 5           # Variable global
def test():
    y = 10      # Variable local con mismo nombre
    print(y)    # 10 (la local)
test()
print(y)        # 5 (la global no cambió)
```

Si es necesario modificar una variable global desde dentro de una función, hay que declararla con `global` [19], pero **se desaconseja** hacerlo cuando sea posible [20]. Lo más claro suele ser pasar valores como parámetros y devolver resultados con `return`.

En resumen, los parámetros y variables internas a una función tienen **alcance local** y no existen fuera de ella, mientras las definidas en el nivel global están disponibles para todo el programa [16] [17]. Evitar abusar de variables globales mejora la claridad y previene errores en programas grandes [20] [17].

Funciones integradas y definidas por el usuario

Python incluye muchas **funciones integradas** ("built-in") listas para usar, como `print()`, `len()`, `min()`, `max()`, etc. Estas vienen con el lenguaje y su documentación está disponible en la ayuda oficial [21]. Además, los programadores pueden crear **funciones personalizadas** (funciones definidas por el usuario) usando `def` [21]. (También existen las funciones anónimas o **lambda**, que se definen sin usar la palabra clave `def` [21].)

Al final, invocar una función (sea integrada o nuestra propia función) se hace de igual modo: usando `nombre_función(parámetros)`. La diferencia es que unas ya están predefinidas por Python y otras las definimos nosotros para tareas específicas. Por ejemplo, `print(…)` es una función integrada para

mostrar texto, mientras que `area_rectangulo(ancho, alto)` puede ser una función definida por el usuario para calcular áreas.

En conclusión, las funciones son piezas clave en Python para modularizar y reutilizar código, ya sean preconstruidas o creadas por nosotros. Usar funciones con parámetros y retorno ordena la lógica del programa y permite soluciones más limpias y flexibles [3] [1].

Ejemplos sencillos:

```
# Ejemplo 1: función con parámetro y retorno
def saludar(nombre):
    return f"¡Hola, {nombre}!"
print(saludar("Laura"))  # ¡Hola, Laura!

# Ejemplo 2: alcance local vs. global
y = 5
def cambio():
    y = 10
    print("Dentro:", y)  # 10 (variable local)
cambio()
print("Fuera:", y)       # 5 (variable global)
```

En el primer ejemplo se ve cómo la función `saludar` recibe un parámetro y devuelve un saludo personalizado. En el segundo, se muestra que la variable `y` definida dentro de `cambio()` es local y no modifica la `y` global [16] [18]. Estos ejemplos ilustran los conceptos básicos de definición, llamada, parámetros, retorno y alcance de variables en funciones de Python.

Fuentes: Conceptos extraídos y adaptados de documentación y tutoriales de Python [2] [1] [4] [16] [18] [21].

---

[1] [5] [7] [21] Funciones de Python: Cómo invocar y escribir funciones | DataCamp
https://www.datacamp.com/es/tutorial/functions-python-tutorial

[2] [3] [4] [6] [8] [9] [10] [11] [12] [13] [14] [15] [16] [18] [19] [20] ▷ Funciones en Python: Guía Completa [Sintaxis, Parámetros y Ejemplos] 2025
https://elpythonista.com/funciones-en-python-guia-completa-2025-sintaxis-parametros-y-ejemplos

[17] Funciones | Aprende con Alf
https://aprendeconalf.es/docencia/python/manual/funciones/