

## GUÍA TEÓRICA

# FUNCIONES EN PYTHON

---

### Módulo 3 – Laboratorio de lenguajes de programación

<b>¿Qué es una función?</b>	<b>2</b>
<b>¿Por qué usar funciones?</b>	<b>2</b>
<b>Conceptos clave</b>	<b>2</b>
Definición y uso:	3
Funciones:	3
Funciones con parámetros	4
Funciones que devuelven valores	4
Uso de return en funciones	5
¿Qué es return?	5
¿Qué pasa si no usamos return?	5
<b>Alcance de las variables (Scope)</b>	<b>6</b>
Alcance local	6
Alcance global	6
¿Qué pasa si uso el mismo nombre dentro de la función?	7
<b>Funciones anidadas</b>	<b>7</b>
<b>Llamadas a Funciones con Argumento</b>	<b>8</b>
Orden de los argumentos	8
Cantidad de argumentos	9
Parámetros opcionales (con valores por defecto)	9
Llamado con argumentos nombrados	10
Recomendación	10
<b>Tipos de funciones en Python</b>	<b>11</b>
Funciones definidas por el usuario	11
Funciones integradas (built-in)	11

# Funciones

## ¿Qué es una función?

Una **función** es un bloque de código (*lo que Python llama **statements** o **sentencias***) reutilizable que agrupa instrucciones bajo un **nombre**. Permite ejecutar ese conjunto de instrucciones cada vez que sea necesario, simplemente "*llamando*" a la función por su nombre y datos de entrada (**parámetros**).

Más allá de su uso básico, las funciones son una alternativa a copiar y pegar código, en lugar de repetir múltiples veces el mismo bloque de operaciones, podemos escribirla una sola vez dentro de una función y re-utilizarla cuando sea necesario.

Esto **reduce la duplicación de código** y **facilita el mantenimiento**: si en el futuro hay que modificar algo en ese bloque de operaciones, sólo será necesario hacerlo en un único lugar, en la definición de la función.

Además, las funciones permiten trabajar de forma **ordenada y modular**. La **modularidad**, considerada una **buena práctica de programación**, consiste en **dividir un problema grande en partes más pequeñas, independientes y manejables**. Cada función representa una de esas partes o subtareas.

## ¿Por qué usar funciones?

- **Evitan la repetición** de código.
- **Dividen un problema en tareas pequeñas**, más fáciles de entender, programar y mantener.
- Facilitan el proceso de pruebas (*testing*).
- Permiten reutilizar soluciones en otros programas.
- Ayudan a que el programa sea más claro y legible.
- Facilitan el trabajo en equipo y la organización del código.

## Conceptos clave

Las funciones se definen usando la palabra clave **def**, seguido de un **nombre** para la función. Como cuando en pseudocódigo ponemos **FUNCIÓN** o **PROCEDIMIENTO** seguido de un **nombre**. Elementos de una función:

- La instrucción **def** crea un objeto función y lo asigna a un nombre.
- **Parámetro**: variable que se declara en la definición de una función, sirve para recibir valores, son los datos de **entrada**.
- **Argumento**: valor literal o variable que se pasa al llamar la función.
- **return**: instrucción que establece que devuelve la función como resultado.

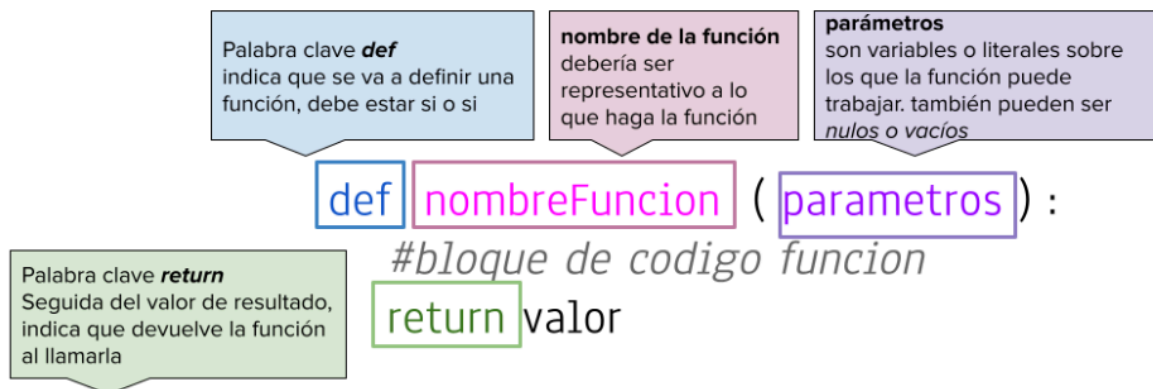
⚠ En Python **no** se diferencian funciones y procedimientos. Se usan con **def** en ambos casos. La diferencia está en si se utiliza o no **return**.

## Definición y uso:

**def** crea lo que Python refiere como un **objeto** (para python las distintas estructuras de datos, e incluso los archivos, se denominan **objetos**) que tiene un **nombre**, y permite ejecutar el bloque de sentencias incluido en **def** al llamar a la función por su nombre.

```
def <nombre>(par1, par2,... parN):  
    ...  
    return <valor>
```

- La línea de encabezado especifica el nombre de la función y una lista de argumentos entre paréntesis. El nombre no pueden ser palabras reservadas como **while** o de funciones ya implementadas como **print()**
- Los parámetros definen los datos de entrada, no se debe definir el tipo de dato, pero normalmente para hacer las operaciones se requerirá un tipo de dato. Esto se puede indicar en un comentario en la definición de la función.
- Luego se definen las sentencias que incluye la función. Que se incluye en la misma se delimita por la **indentación**. Donde este la primera sentencia a la misma altura que **def** termina el bloque.
- Los cuerpos de función pueden contener una instrucción **return**, pero no es obligatorio. Si no se incluye un return solo ejecutara las acciones, siendo lo análogo al **procedimiento** en pseudocódigo





⚠ Presta atención que cuando hacemos referencia a una función el **nombre** va seguido de **parentesis ()**, aunque no contenga parámetros. **Esto es igual en pseudocódigo.**

Recuerda que la llamada de la función debe estar después de su definición

## Ejemplos de funciones:

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

 En pseudocódigo esto sería un procedimiento, no tiene sentencia return, solo ejecuta acciones, como mostrar un mensaje por pantalla

 En pseudocódigo:

```
PROCEDIMIENTO saludar() Es  
    Escribir("Hola, mundo")  
FinAccion
```

## Funciones con parámetros

```
def saludar(nombre):  
    print("Hola", nombre)  
  
saludar("Ana")
```


Recibe un parámetro y personaliza el saludo. No es necesario definir el tipo de dato. No devuelve valor.

 En pseudocódigo:


```
PROCEDIMIENTO saludar(nombre:alfanumerico) Es  
    Escribir("Hola ", nombre)  
FinAccion
```

## Funciones que devuelven valores

```
def sumar(a, b):  
    return a + b  
  
resultado = sumar(3, 5) # Pasa 2 argumentos y guarda en resultado  
print(resultado) # 8
```

 En pseudocódigo esto sería una **función**, devuelve un valor porque tiene sentencia **return**. Al igual que en pseudocódigo deberíamos resguardar el valor

## asignando la llamada a una variable

 En pseudocódigo:

```
Función sumar(a, b: entero) : entero
    sumar := a + b
FinFunción

resultado:= sumar(3, 5)
Escribir(resultado)
```



En realidad, aunque no se incluya una sentencia **return**, la función va a devolver un valor y es *None* de valor nulo.

[Python None Keyword - GeeksforGeeks](https://www.geeksforgeeks.org/python-none-keyword/)

Otro elemento que se suele agregar a la definición de funciones es un comentario especificando: **que hace, que datos de entrada precisa, y en qué orden, y que retorna la función**. Se denominan **docstring** y se definen dentro de **triples comillas** `'''` y se accede a ellas con la función ya implementada utilizando: **help()**

### Estructura de un docstring:

1. Primera línea: Debe contener una breve descripción de la función.
2. Segunda línea (opcional): Debe ser una línea en blanco para separar visualmente la descripción de la documentación más detallada.
3. Líneas siguientes: Contienen una descripción más detallada, incluyendo la documentación de los argumentos, el valor de retorno, los efectos secundarios, etc.

```
def sumar(x, y):
    """
    Esta función suma dos números.

    Args:
        x: El primer número.
        y: El segundo número.

    Returns:
        Un entero → La suma de x e y.
    """
    return x + y
```

```
1 def sumar(x, y):
2     """
3     Esta función suma dos números.
4
5     Args:
6     x: El primer número.
7     y: El segundo número.
8
9     Returns:
10    Un entero → La suma de x e y.
11    """
12    return x + y
13
14 help(sumar)
```

Help on function sumar in module \_\_main\_\_:

```
sumar(x, y)
    Esta función suma dos números.

    Args:
        x: El primer número.
        y: El segundo número.

    Returns:
        Un entero → La suma de x e y.
```

Process finished with exit code 0


Los docstrings son una herramienta esencial para documentar el código Python. Ayudan a mejorar la comprensión del código, facilitan la creación de documentación formal y promueven la buena práctica de programación.

 [Python Docstrings \(With Examples\)](#)

## Uso de return en funciones

### ¿Qué es **return**?

- La palabra clave **return** se usa dentro de una función para especificar que valor o valores **devuelve** la función al **llamarla**, es su **resultado**.
- Cuando se ejecuta una sentencia **return**, la función **termina inmediatamente** y envía ese valor como resultado.

 En pseudocódigo esto sería asignar (**:=**) el valor al nombre de la función.

### ¿Qué pasa si no usamos **return**?

- La función sólo hace lo que está dentro de su cuerpo, pero **no devuelve ningún valor útil**.
- Por eso, no podemos usar su “resultado” para hacer cálculos o asignaciones, porque será **None**.
- Se suele usar funciones sin **return** cuando queremos realizar solo ejecutar **acciones**, como mostrar mensajes, modificar datos, etc.

```
def sin_retorno():  
  
    print("Solo imprime")  
  
x = sin_retorno()  
  
print(x)  # Muestra: None
```

Resultado:

```
n_Lab/prueba_break.py  
Solo imprime  
None  
PS C:\Users\natas\OneDrive\Documentos\Python_Lab>
```

## Alcance de las variables (Scope)

El **Alcance** o "**scope**" determina desde qué partes del código se puede acceder a una variable. En Python, el alcance puede ser:

### Alcance local

Una variable tiene **alcance local** cuando es definida dentro de una función. Solo se puede usar dentro de esa función.

```
def mostrar():  
    mensaje = "Hola" # variable local  
    print(mensaje)  
  
mostrar()  
print(mensaje) # ❌ Error: mensaje no existe fuera de la función
```

Resultado:

```
Hola  
Traceback (most recent call last):  
  File "c:\Users\natas\OneDrive\Documentos\Python_Lab\prueba_break.py", line 6, in <module>  
    print(mensaje) # ❌ Error: mensaje no existe fuera de la función  
          ^^^^^^^  
NameError: name 'mensaje' is not defined  
PS C:\Users\natas\OneDrive\Documentos\Python_Lab> 
```

### Alcance global

Una variable tiene **alcance global** cuando se define por fuera de todas las funciones. Puede ser usada en todo el programa.

```
nombre = "Ana" # variable global  
  
def saludar():  
    print("Hola", nombre)  
  
saludar() # ✅ Hola Ana
```

Resultado:

```
Hola Ana  
PS C:\Users\natas\OneDrive\Documentos\Python_Lab> 
```

## ¿Qué pasa si uso el mismo nombre dentro de la función?

Python **crea una nueva variable local**, sin afectar la variable global

```
x = 10 # variable global
```

```
def cambiar():  
    x = 5 # nueva variable local  
    print("Dentro:", x)  
  
cambiar()  
print("Fuera:", x) # sigue siendo 10
```

Resultado:

```
Dentro: 5  
Fuera: 10  
PS C:\Users\natas\OneDrive\Documentos\Python_Lab>
```

Si realmente querés que una variable definida dentro del scope de una función se pueda **modificar a variable global**, tenés que decirle explícitamente a Python usando **global**

```
contador = 0  
  
def incrementar():  
    global contador  
    contador += 1  
  
incrementar()  
print(contador) # 1
```

Resultado:

```
1  
PS C:\Users\natas\OneDrive\Documentos\Python_Lab>
```

## Funciones anidadas

Una **función anidada** es una función **definida dentro de otra función**. Se usan para:

- Encapsular comportamiento que sólo tiene sentido dentro de la función principal.
- Evitar repetir código dentro de la función exterior.
- Organizar mejor el código.

```
def procesar():  
    def mensaje():  
        print("Inicio del proceso")  
  
    mensaje()  
    print("Procesando datos...")
```



```
procesar()
```

Resultado:

```
Inicio del proceso
Procesando datos...
PS C:\Users\natas\OneDrive\Documentos\Python_Lab> █
```

## Llamadas a Funciones con Argumento

Cuando definimos una función, podemos hacer que reciba **argumentos** (también llamados *parámetros*). Estos son **valores que le damos a la función** para que los use dentro de su cuerpo.

Cuando **llamamos a una función** en Python, debemos respetar el número y el orden de los **argumentos que definimos**. Si no lo hacemos, el programa **lanzará un error** y se detendrá la ejecución.

### Orden de los argumentos

El orden sí importa

Los argumentos se asignan en el mismo orden que se definieron.

```
def mostrar_datos(nombre, edad):

    print("Nombre:", nombre)

    print("Edad:", edad)

mostrar_datos("Lucía", 12)  # nombre = "Lucía", edad = 12
```

Resultado:

```
Nombre: Lucía
Edad: 12
PS C:\Users\natas\OneDrive\Documentos\Python_Lab> █
```

⚠ Si cambiamos el orden:

```
def mostrar_datos(nombre, edad):

    print("Nombre:", nombre)

    print("Edad:", edad)
```

```
mostrar_datos(12, "Lucía") # nombre = 12,  
edad= "Lucía"
```

Resultado:

```
Nombre: 12  
Edad: Lucía  
PS C:\Users\natas\OneDrive\Documentos\Python_Lab>
```

## Cantidad de argumentos

Debe coincidir el número de argumentos al llamar la función con los parámetros definidos.

```
def saludar(nombre, mensaje):  
  
    print(mensaje, nombre)  
  
saludar("Ana", "Hola") # ✓ correcto  
  
saludar("Ana") # ✗ falta un argumento
```

Resultado:

```
Hola Ana  
Traceback (most recent call last):  
  File "c:\Users\natas\OneDrive\Documentos\Python_Lab\prueba_break.py", line 6, in <module>  
    saludar("Ana") # ✗ falta un argumento  
    ~~~~~^~~~~~  
TypeError: saludar() missing 1 required positional argument: 'mensaje'  
PS C:\Users\natas\OneDrive\Documentos\Python_Lab>
```

## Parámetros opcionales (con valores por defecto)

Podemos asignar un valor por defecto a un parámetro. Si no se da ese argumento, usará ese valor por defecto.


```
def saludar(nombre, mensaje="Hola"):  
  
    print(mensaje, nombre)  
  
saludar("Marta") # Usa el valor por defecto: "Hola"  
  
saludar("Marta", "Buen día") # Usa "Buen día"
```

Resultado:

```
Hola Marta  
Buen día Marta  
PS C:\Users\natas\OneDrive\Documentos\Python_Lab>
```

## Llamado con argumentos nombrados

Se puede especificar el nombre del parámetro al llamar la función, y así **no importa el orden**.

```
def mostrar_datos(nombre, edad):  
  
    print("Nombre:", nombre)  
  
    print("Edad:", edad)  
  
mostrar_datos(edad=14, nombre="Juan") #  válido
```

Resultado:

```
Python_Lab/prueba_01_cad.py  
Nombre: Juan  
Edad: 14  
PS C:\Users\natas\OneDrive\Documentos\Python_Lab>
```

## Recomendación

 Siempre verificar:

- Que el número de argumentos coincida con la definición.
- Que estén en el orden correcto (si son posicionales).
- O usar argumentos con nombre para mayor claridad.

## Tipos de funciones en Python

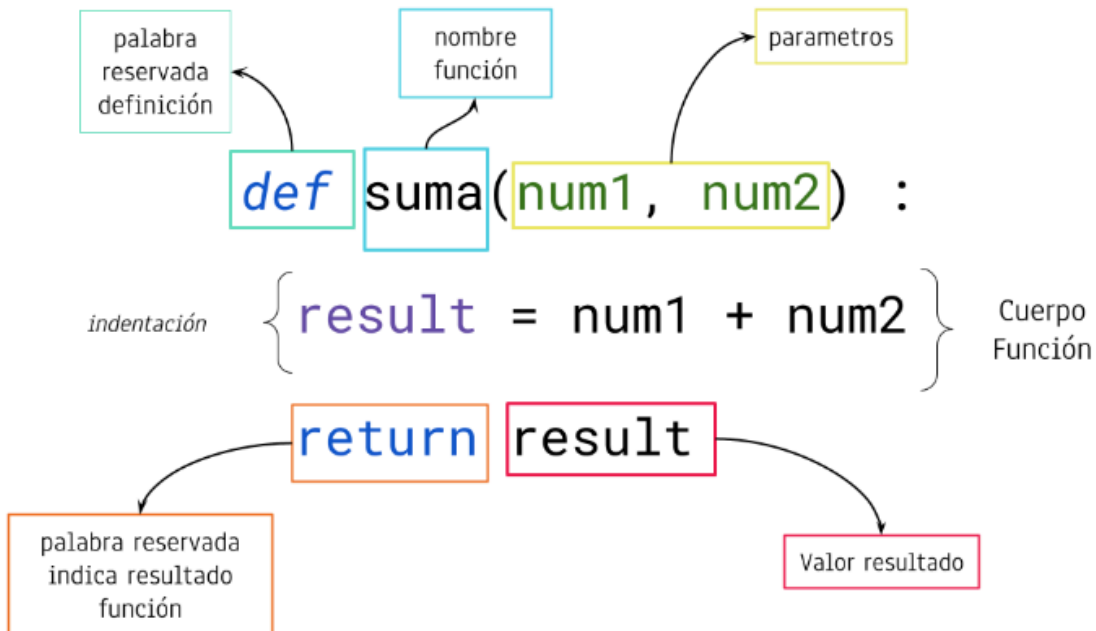
En Python, podemos usar dos tipos principales de funciones:

### Funciones definidas por el usuario

Las **funciones definidas por el usuario** son funciones que **vos creás** para resolver una tarea específica dentro de tu programa. Se escriben usando la palabra clave

**def** y tienen un **nombre**, **parámetros (opcionales)** y un **bloque de instrucciones** que se ejecuta cada vez que se llama la función.

*Repasamos nuevamente la sintaxis:*



## Funciones integradas (*built-in*)

Las **funciones integradas** son funciones que ya vienen definidas en Python. Están listas para usarse al contar con un **intérprete de python** (lo que se descargó en el módulo 1). Son muy útiles para tareas básicas como mostrar mensajes, convertir tipos de datos, hacer cálculos simples, y más. Estas funciones **se pueden usar directamente** en cualquier programa sin necesidad de importarlas ni definir las. Algunas de las más comunes son:

Función	Descripción	Ejemplo
<code>print()</code>	Muestra un mensaje en pantalla  Análogo al ESCRIBIR() de pseudocódigo	<code>print("Hola")</code>
<code>len()</code>	Devuelve la longitud de una secuencia (lista, string, etc.)	<code>len("hola") → 4</code>

<code>type()</code>	Muestra el tipo de un dato	<code>type(3) → &lt;class 'int'&gt;</code>
<code>int()</code>	Convierte cualquier variable o literal de otro tipo de dato a entero	<code>int("5") → 5</code>
<code>float()</code>	Convierte a decimal	<code>float("3.14") → 3.14</code>
<code>str()</code>	Convierte a texto - string	<code>str(10) → "10"</code>
<code>bin()</code>	Convierte un entero a binario. Arranca en el prefijo 0b	<code>bin(3) → '0b11'</code>
<code>input()</code>	Permite ingresar datos desde teclado  Análogo al LEER() de pseudocódigo	<code>nombre = input("Tu nombre: ")</code>
<code>abs()</code>	Devuelve el valor absoluto de un numero	<code>abs(-5) → 5</code>

## Otros recursos:

### Ejercicios con solución:

<https://pynative.com/python-functions-exercise-with-solutions/> (en ingles - puedes ayudarte con un chat o el traductor deepl)

Ejemplos de funciones (en español):

<https://github.com/jvadillo/aprende-python-desde-cero-a-experto/blob/master/manuscript/07-Funciones.md>

**github es un sitio para repositorios de código, en tercer año aprenderás sobre como usar git y github y son una habilidad esencial para insertarse en la industria IT.**

[Libro Learning Python O'Reilly](#) (en ingles)

[Libro Data Structures and Algorithms in Python](#) (en ingles)