



Funciones en Python

Índice:

1. Definir una función

Parámetros por defecto

Retorno de valores con Return

2. Ampliación: Argumentos en funciones

Argumentos posicionales

Argumentos nominales

3. Paso por valor vs paso por referencia

4. Alcance de las variables en funciones (local y global)

5. Introducción a módulos en Python

Crear un módulo

Importar un módulo

Importar elementos específicos

Importar todo

Módulos estándar en Python

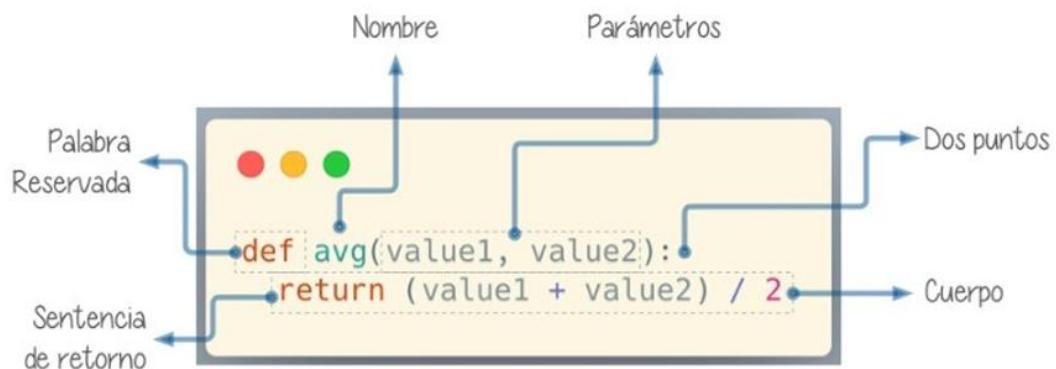
6. Creando un paquete de módulos

Anexo 1. Empaquetar y desempaquetar

Anexo 2. Funciones Lambda



Las **funciones** en Python nos permiten organizar el código en bloques reutilizables, lo que facilita la legibilidad y el mantenimiento.



1. Definir una función

Para definir una función usamos la palabra clave **def**, seguida del nombre de la función y un conjunto de parámetros entre paréntesis. Luego, el cuerpo de la función se escribe con sangría.

Ejemplo básico de función [sin parámetros](#):

```
def saludar():
    print("¡Hola! Bienvenido a Python.")

saludar() # Llamada a la función
```

Ejemplo con parámetros:

```
def sumar(a, b):
    return a + b # Devuelve la suma de a y b

resultado = sumar(5, 3)
print(resultado) # Output: 8
```



REGISTRO NACIONAL DE ASOCIACIONES N°611922
DECLARADA ENTIDAD DE UTILIDAD PÚBLICA ESTATAL
AGENCIA DE COLOCACIÓN: ID 0100000017

CENTRO EN MÁLAGA
C/DOS ACERAS 23, 29012
MÁLAGA | (+34) 952 300 500
ARRABAL@ARRABALEMPLEO.ORG

CENTRO EN CÁDIZ
TR.ª ALAMEDA DE SOLANO, 32, 11130
CHICLANA DE LA FRONTERA | (+34) 956 900 312
CHICLANA@ARRABALEMPLEO.ORG



Parámetros por defecto

Podemos asignar valores predeterminados a los parámetros para que la función pueda ejecutarse sin necesidad de proporcionarlos siempre.

Ejemplo con valores por defecto:

```
def saludar(nombre="Invitado"):
    print(f"¡Hola, {nombre}!")

saludar()          # Output: ¡Hola, Invitado!
saludar("Carlos") # Output: ¡Hola, Carlos!
```

Retorno de valores con return

Las funciones pueden devolver valores usando return. Esto permite almacenar el resultado en una variable o usarlo en otras partes del código.

Ejemplo con return:

```
def cuadrado(numero):
    return numero ** 2 # Retorna el número elevado al cuadrado

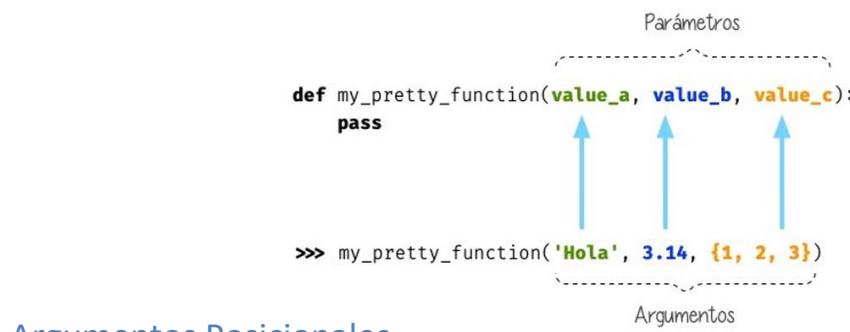
resultado = cuadrado(4)
print(resultado) # Output: 16
```





Ampliación: Argumentos en funciones

Cuando llamamos a una función en Python, podemos pasarle argumentos de diferentes formas. Las más comunes son:



Argumentos Posicionales

Los argumentos **posicionales** se asignan a los parámetros en el orden en el que aparecen en la llamada a la función.

 **Ventaja:** Es simple y fácil de entender.

 **Desventaja:** Debemos recordar el orden correcto de los parámetros.

Ejemplo de argumentos posicionales:

```
def mostrar_info(nombre, edad):
    print(f"Nombre: {nombre}")
    print(f"Edad: {edad}")

# Llamada con argumentos posicionales
mostrar_info("Carlos", 30)
```

```
Nombre: Carlos
Edad: 30
```

Si cambiamos el orden de los argumentos, obtenemos un resultado incorrecto:

```
mostrar_info(30, "Carlos") # Error Lógico
```

Salida incorrecta:

```
Nombre: 30
Edad: Carlos
```

Esto ocurre porque el primer argumento se asigna a nombre y el segundo a edad, sin verificar si tienen sentido.



REGISTRO NACIONAL DE ASOCIACIONES N°611922
DECLARADA ENTIDAD DE UTILIDAD PÚBLICA ESTATAL
AGENCIA DE COLOCACIÓN: ID 0100000017

CENTRO EN MÁLAGA
C/DOS ACERAS 23, 29012
MÁLAGA | (+34) 952 300 500
ARRABAL@ARRABALEMPLEO.ORG

CENTRO EN CÁDIZ
TR.^a ALAMEDA DE SOLANO, 32, 11130
CHICLANA DE LA FRONTERA | (+34) 956 900 312
CHICLANA@ARRABALEMPLEO.ORG



Argumentos Nominales (Keyword Arguments)

En este caso, asignamos los valores a los parámetros **por su nombre** en la llamada a la función. Esto evita depender del orden de los argumentos.

-  **Ventaja:** Mejora la legibilidad y evita errores de orden.
-  **Desventaja:** Puede ser más verboso al escribir.

Ejemplo de argumentos nominales:

```
mostrar_info(edad=30, nombre="Carlos") # No importa el orden
```

Salida:

```
Nombre: Carlos
Edad: 30
```

Combinación de argumentos posicionales y nominales

Podemos mezclar ambos tipos, **pero los posicionales deben ir antes de los nominales**.

Ejemplo válido:

```
def datos_personales(nombre, edad, ciudad):
    print(f"Nombre: {nombre}, Edad: {edad}, Ciudad: {ciudad}")

datos_personales("Ana", 25, ciudad="Barcelona") # Correcto
```

Ejemplo incorrecto (nominal antes de posicional)

```
datos_personales(nombre="Ana", 25, "Barcelona") # ✗ ERROR
```

Resumen

Tipo de argumento	Descripción	Ejemplo
Posicionales	Se asignan en el orden de la definición.	funcion(1, 2, 3)
Nominales	Se asignan por nombre, sin importar el orden.	funcion(a=1, b=2)
Mezcla	Posicionales primero, luego nominales.	funcion(1, b=2)





Paso por valor vs. Paso por referencia en Python

❖ Paso por valor (para tipos inmutables)

Los **tipos inmutables** como enteros, flotantes, cadenas de texto y tuplas **se pasan como si fuera por valor**. Es decir, si los modificamos dentro de una función, no afectan a la variable original.

Ejemplo con tipos inmutables (entero)

```
def duplicar(numero):
    numero = numero * 2 # Se crea una nueva variable en memoria
    print("Dentro de la función:", numero)

x = 10
duplicar(x)
print("Fuera de la función:", x)
```

Dentro de la función: 20
Fuera de la función: 10

¿Por qué x sigue siendo 10 fuera de la función?

Porque los enteros son **inmutables**, así que cuando se pasa x a la función, lo que realmente se pasa es **una copia de su valor**, no el objeto original.

❖ Paso por referencia (para tipos mutables)

Los **tipos mutables** como listas, diccionarios y conjuntos **se pasan como si fuera por referencia**. Es decir, si los modificamos dentro de una función, los cambios afectan a la variable original.

Ejemplo con tipos mutables (lista)

```
def agregar_elemento(lista):
    lista.append(4) # Modifica la lista original
    print("Dentro de la función:", lista)

mi_lista = [1, 2, 3]
agregar_elemento(mi_lista)
print("Fuera de la función:", mi_lista)
```

Dentro de la función: [1, 2, 3, 4]
Fuera de la función: [1, 2, 3, 4]

¿Por qué mi_lista sí cambia fuera de la función?

Porque las listas son **mutables**, y lo que se pasa a la función es la **referencia en memoria** de la lista original, no una copia.





Resumen

Tipo de dato	¿Cómo se pasa en Python?	Ejemplo
Inmutables (int, float, str, tuple)	Como si fuera por valor (no se modifican fuera de la función)	x = 10 (No cambia fuera de la función)
Mutables (list, dict, set)	Como si fuera por referencia (los cambios afectan fuera de la función)	mi_lista.append(4) (Cambia fuera de la función)

💡 Evitar modificaciones no deseadas en mutables

Si no queremos que un objeto mutable se modifique dentro de la función, podemos hacer una **copia** usando `copy()`.

Ejemplo con copia de lista

```
def modificar_lista(lista):
    lista = lista.copy() # Creamos una copia para evitar modificar la original
    lista.append(99)
    print("Dentro de la función:", lista)

mi_lista = [1, 2, 3]
modificar_lista(mi_lista)
print("Fuera de la función:", mi_lista) # La lista original NO cambia
```

Salida:

```
Dentro de la función: [1, 2, 3, 99]
Fuera de la función: [1, 2, 3]
```

Aquí, `lista.copy()` crea una nueva lista, por lo que la original no se modifica.

Conclusión

- ✓ En Python, los **inmutables** se comportan como si se pasaran por valor.
- ✓ Los **mutables** se comportan como si se pasaran por referencia.
- ✓ Para evitar efectos no deseados con objetos mutables, podemos usar `.copy()`.





Alcance de variables en funciones (local y global)

Las variables dentro de una función tienen un **ámbito local**, lo que significa que solo existen dentro de esa función.

Ejemplo de variable local:

```
def mi_funcion():
    x = 10 # Variable Local
    print(x)

mi_funcion()
# print(x) # Esto daría un error porque 'x' no existe fuera de la función.
```

Para modificar una variable global dentro de una función, usamos **global**:

Ejemplo con global:

```
contador = 0

def incrementar():
    global contador
    contador += 1

incrementar()
print(contador) # Output: 1
```

Resumen de funciones en Python

Concepto	Descripción
Definición (def)	Se usa <code>def nombre_funcion():</code> para definir una función.
Parámetros	Se pueden pasar valores dentro de () para personalizar la función.
Valores por defecto	Se pueden asignar valores predeterminados a los parámetros.
return	Permite que una función devuelva un valor.
Variables locales	Solo existen dentro de la función.
global	Se usa para modificar variables globales dentro de una función.





Introducción a los Módulos en Python

Un **módulo** en Python es un archivo con extensión .py que contiene **código reutilizable** como funciones, clases y variables. Los módulos nos permiten **organizar mejor nuestro código y evitar repetir código innecesariamente**.

💡 ¿Por qué usar módulos?

- Reutilización:** Podemos escribir funciones en un módulo y usarlas en varios programas.
 - Organización:** Dividimos un programa grande en archivos más pequeños y manejables.
 - Mantenibilidad:** Si necesitamos hacer cambios, solo modificamos el módulo en lugar de cada script.
-

Creando un Módulo

Un módulo es simplemente un archivo .py con código dentro.

Ejemplo: mimodulo.py

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

PI = 3.1416
```

Este módulo contiene una función saludar() y una variable PI.

Importando un Módulo

Para usar un módulo en otro archivo, usamos la palabra clave **import**.

Ejemplo: Usando el módulo en otro script

```
import mimodulo # Importamos el módulo

print(mimodulo.saludar("Ana")) # Llamamos a la función del módulo
print(mimodulo.PI) # Usamos la variable del módulo
```

¡Hola, Ana!
3.1416



REGISTRO NACIONAL DE ASOCIACIONES N°611922
DECLARADA ENTIDAD DE UTILIDAD PÚBLICA ESTATAL
AGENCIA DE COLOCACIÓN: ID 0100000017

CENTRO EN MÁLAGA
C/DOS ACERAS 23, 29012
MÁLAGA | (+34) 952 300 500
ARRABAL@ARRABALEMPLEO.ORG

CENTRO EN CÁDIZ
TR.ª ALAMEDA DE SOLANO, 32, 11130
CHICLANA DE LA FRONTERA | (+34) 956 900 312
CHICLANA@ARRABALEMPLEO.ORG



Importando Elementos Específicos

Podemos importar solo ciertas funciones o variables usando `from ... import ...`

```
from mimodulo import saludar, PI

print(saludar("Carlos")) # No necesitamos escribir mimodulo.saludar()
print(PI)
```

También podemos usar `as` para renombrar módulos o funciones:

```
import mimodulo as mm

print(mm.saludar("Luis")) # Usamos el alias 'mm'
```

Importando Todo con *

Podemos importar **todo** el contenido de un módulo con `from ... import *`, pero **no es recomendable** porque puede generar conflictos de nombres.

```
from mimodulo import *

print(saludar("Lucía")) # Se usa sin prefijo
print(PI)
```

Módulos Estándar en Python

Python tiene muchos módulos incorporados que podemos usar sin necesidad de instalarlos.

Ejemplo: Módulo math

```
import math

print(math.sqrt(16)) # Raíz cuadrada
print(math.pi) # Número Pi
```

Ejemplo: Módulo random

```
import random

print(random.randint(1, 10)) # Número aleatorio entre 1 y 10
```





Creando un Paquete de Módulos

Si queremos organizar varios módulos, podemos agruparlos en una **carpeta** llamada **paquete**.

Estructura:

```
mi_paquete/
|__ __init__.py    # Indica que es un paquete
|__ modulo1.py
|__ modulo2.py
```

Luego, podemos importar un módulo desde el paquete:

```
from mi_paquete import modulo1
```

Resumen

Concepto	Ejemplo
Crear un módulo	Archivo .py con funciones y variables.
Importar un módulo	import mimodulo
Importar algo específico	from mimodulo import funcion
Renombrar un módulo	import mimodulo as mm
Importar todo	from mimodulo import * (no recomendado)
Módulos estándar	import math, import random
Crear un paquete	Carpeta con __init__.py y varios módulos



Anexo 1: Empaquetar y desempaquetar.

Argumentos posicionales con * (Empaquetado y Desempaquetado)

- ❖ **Empaquetar argumentos con * en una función**

Cuando usamos `*nombre_parametro` en la definición de una función, **todos los argumentos posicionales que se pasen se empaquetarán en una tupla**.

Útil cuando no sabemos cuántos argumentos se pasarán a la función.

Ejemplo de empaquetado con `*args`:

```
def sumar_todos(*numeros):
    print(numeros) # Muestra la tupla con los argumentos
    return sum(numeros)

resultado = sumar_todos(1, 2, 3, 4, 5)
print(resultado)
```

(1, 2, 3, 4, 5)
15

Aquí, `números` es una **tupla** que almacena todos los valores pasados a la función.

- ❖ **Desempaquetar una lista o tupla con * en la llamada a la función**

También podemos usar `*` para **desempaquetar** una lista o tupla cuando llamamos a la función.

Ejemplo de desempaquetado en una llamada a la función:

```
def mostrar_info(nombre, edad):
    print(f"Nombre: {nombre}")
    print(f"Edad: {edad}")

datos = ("Ana", 25)
mostrar_info(*datos) # Desempaquetá la tupla y pasa los valores
```

Nombre: Ana
Edad: 25

Aquí, `*datos` descompone la tupla en sus elementos individuales, pasándolos como argumentos posicionales.



REGISTRO NACIONAL DE ASOCIACIONES N°611922
DECLARADA ENTIDAD DE UTILIDAD PÚBLICA ESTATAL
AGENCIA DE COLOCACIÓN: ID 0100000017

CENTRO EN MÁLAGA
C/DOS ACERAS 23, 29012
MÁLAGA | (+34) 952 300 500
ARRABAL@ARRABALEMPLEO.ORG

CENTRO EN CÁDIZ
TR.ª ALAMEDA DE SOLANO, 32, 11130
CHICLANA DE LA FRONTERA | (+34) 956 900 312
CHICLANA@ARRABALEMPLEO.ORG



Resumen

Uso de *	Descripción	Ejemplo
Empaquetar (*args)	Convierte múltiples argumentos en una tupla dentro de la función.	def funcion(*args):
Desempaquetar (*)	Expande una lista/tupla en varios argumentos posicionales.	funcion(*mi_tupla)

Argumentos Nominales con ** (Empaquetado y Desempaquetado)

◊ Empaquetar argumentos con ** en una función

Cuando usamos `**nombre_parametro` en la definición de una función, **todos los argumentos nominales (keyword arguments) que se pasen se almacenarán en un diccionario**.

Útil cuando no sabemos cuántos argumentos nominales se pasarán a la función.

Ejemplo de empaquetado con `**kwargs`:

```
def mostrar_info(**kwargs):
    print(kwargs) # Muestra el diccionario con los argumentos
    for clave, valor in kwargs.items():
        print(f'{clave}: {valor}')

# Llamada con argumentos nominales
mostrar_info(nombre="Ana", edad=25, ciudad="Barcelona")
```

Salida:

```
{'nombre': 'Ana', 'edad': 25, 'ciudad': 'Barcelona'}
nombre: Ana
edad: 25
ciudad: Barcelona
```

Aquí, `kwargs` es un **diccionario** que almacena todos los valores pasados como argumentos nominales.



REGISTRO NACIONAL DE ASOCIACIONES N°611922
DECLARADA ENTIDAD DE UTILIDAD PÚBLICA ESTATAL
AGENCIA DE COLOCACIÓN: ID 0100000017

CENTRO EN MÁLAGA
C/DOS ACERAS 23, 29012
MÁLAGA | (+34) 952 300 500
ARRABAL@ARRABALEMPLEO.ORG

CENTRO EN CÁDIZ
TR.^a ALAMEDA DE SOLANO, 32, 11130
CHICLANA DE LA FRONTERA | (+34) 956 900 312
CHICLANA@ARRABALEMPLEO.ORG



◇ Desempaquetar un diccionario con ** en la llamada a la función

También podemos usar ** para **desempaquetar un diccionario** cuando llamamos a una función.

Ejemplo de desempaquetado en una llamada a la función:

```
def mostrar_info(nombre, edad, ciudad):
    print(f"Nombre: {nombre}")
    print(f"Edad: {edad}")
    print(f"Ciudad: {ciudad}")

datos = {"nombre": "Carlos", "edad": 30, "ciudad": "Madrid"}
mostrar_info(**datos) # Desempaquetar el diccionario y pasa los valores
```

Salida:

```
Nombre: Carlos
Edad: 30
Ciudad: Madrid
```

Aquí, **datos descompone el diccionario y pasa cada clave como un argumento nominal.

Resumen

Uso de **	Descripción	Ejemplo
Empaquetar (**kwargs)	Convierte múltiples argumentos nominales en un diccionario.	def funcion(**kwargs):
Desempaquetar (**)	Expande un diccionario en varios argumentos nominales.	funcion(**mi_diccionario)

Combinación de *args y **kwargs

Podemos usar **ambos tipos de argumentos en una misma función**, pero ***args debe ir antes de **kwargs**.



REGISTRO NACIONAL DE ASOCIACIONES N°611922
DECLARADA ENTIDAD DE UTILIDAD PÚBLICA ESTATAL
AGENCIA DE COLOCACIÓN: ID 0100000017

CENTRO EN MÁLAGA
C/DOS ACERAS 23, 29012
MÁLAGA | (+34) 952 300 500
ARRABAL@ARRABALEMPLEO.ORG

CENTRO EN CÁDIZ
TR.^a ALAMEDA DE SOLANO, 32, 11130
CHICLANA DE LA FRONTERA | (+34) 956 900 312
CHICLANA@ARRABALEMPLEO.ORG



Anexo 2:

Funciones anónimas (lambda)

Las funciones lambda son funciones pequeñas y rápidas que se escriben en una sola línea.

Ejemplo de función lambda:

```
doble = lambda x: x * 2

print(doble(4)) # Output: 8
```

Otro ejemplo con dos parámetros:

```
multiplicar = lambda x, y: x * y
print(multiplicar(3, 4)) # Output: 12
```

