1. Funciones en Python – Introducción

1.1. Introducción

Hasta ahora hemos escrito programas que se ejecutan **línea por línea**. Pero a medida que los programas crecen, necesitamos una forma de **organizar el código** para que no sea repetitivo y sea fácil de mantener.

Ahí entran las **funciones**: bloques de código que podemos reutilizar cuando los necesitemos.

1.2. ¿Qué es una función?

Una **función** es un bloque de código que:

- Tiene un **nombre**.
- Puede recibir **valores de entrada** (parámetros).
- Puede devolver un **resultado** (con return).
- Se puede **usar tantas veces como queramos** con solo llamarla.

Ejemplo de función muy simple:

```
def saludar():
    print("Hola, bienvenido a Python")

saludar()  # Llamada a la función
```

1.3. ¿Por qué usar funciones?

Las funciones hacen que el código sea:

- **Reutilizable** → no repetimos las mismas instrucciones muchas veces.
- **Claro** → cada función tiene un propósito definido.
- **Modular** → el programa se divide en piezas pequeñas fáciles de entender.
- **Mantenible** → si algo falla, arreglamos solo en la función y se corrige en todas partes donde se use.

Ejemplo: sin función vs. con función

```
# Sin función
print("Hola, Ana")
print("Hola, Luis")
print("Hola, Marta")

# Con función
def saludar(nombre):
  print("Hola,", nombre)

saludar("Ana")
saludar("Luis")
saludar("Marta")
```

1.4. Funciones ya existentes vs. funciones definidas por el usuario

En Python ya tenemos muchas funciones listas para usar:

- len() → devuelve la longitud de una lista, string, etc.
- print() → muestra algo en pantalla.
- range() → genera secuencias de números.

Pero también podemos crear **nuestras propias funciones** (definidas por el usuario) con la palabra clave def.

Ejemplo:

```
# Función existente
print(len("Python"))  # 6

# Función creada por el usuario
def cuadrado(n):
  return n * n

print(cuadrado(5))    # 25
```

Resumen:

- Una función es un bloque de código con nombre.
- Se usan para organizar, simplificar y reutilizar.
- Python trae funciones listas, pero también podemos crear las nuestras.

2. Funciones en Python – Definir y llamar funciones

2.1. Definir una función

En Python usamos la palabra clave **def** para crear una función.

Sintaxis básica:

```
def nombre_funcion(parámetros):
    # bloque de código
    return resultado
```

- def → palabra reservada que indica que estamos creando una función.
- nombre_funcion → identificador que usamos para llamarla después.
- parámetros → datos de entrada (opcionales).
- return → valor que devuelve la función (opcional).

2.2. Llamar a una función

Una vez definida, la **ejecutamos** escribiendo su nombre y (si corresponde) los parámetros entre paréntesis.

```
def saludar():
    print("Hola, ¿cómo estás?")

# Llamada a la función
saludar()
```

2.3. print vs. return

Esto suele confundir a los principiantes:

- print() → **muestra** algo en pantalla, pero no lo guarda para usarlo después.
- return → **devuelve un valor** que podemos guardar en una variable y reutilizar.

Ejemplo:

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

def sumar_print(a, b):
  print(a + b)

# Usando return
resultado = sumar(3, 4)
print("El resultado es:", resultado)  # Puedo reutilizar el valor

# Usando print
sumar_print(3, 4)   # Solo lo muestra en pantalla
```

2.4. Funciones sin parámetros y con parámetros

Función sin parámetros

```
def mensaje_bienvenida():
  return "¡Bienvenido al curso de Python!"

print(mensaje_bienvenida())
```

Función con parámetros

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

print(saludar("Ana"))
```

```
print(saludar("Luis"))
```

2.5. Funciones sin return

Si no usamos return, la función devuelve None por defecto.

```
def despedir():
   print("Adiós")

resultado = despedir()
print("Resultado:", resultado)  # Muestra: None
```

Resumen de esta parte

- Las funciones se definen con def y se llaman con su nombre.
- print muestra información; return devuelve valores para usarlos.
- Puede haber funciones con o sin parámetros, con o sin return.

3. Funciones en Python – Parámetros

3.1. Parámetros simples

Los **parámetros** permiten que una función reciba **información desde fuera** para usarla dentro del bloque de código.

Ejemplo:

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

saludar("Ana")
saludar("Luis")
```

- Aquí, nombre es un **parámetro**.

- Podemos llamar la función con distintos valores, y la función usará ese valor cada vez.

3.2. Múltiples parámetros

Una función puede recibir **varios parámetros**, separados por comas.

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

print(sumar(3, 4))   # 7
print(sumar(10, 5))  # 15
```

- a y b son parámetros.
- Los valores que pasamos al llamar a la función se llaman **argumentos**.

3.3. Parámetros con valores por defecto

Podemos dar un **valor predeterminado** a un parámetro. Si al llamar la función no pasamos un valor, se usa el valor por defecto.

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

saludar("Ana")    # Hola, Ana!
saludar()         # Hola, amigo!
```

- Útil para que la función sea flexible y no requiera siempre todos los argumentos.

3.4. Parámetros opcionales

- Son parámetros con **valor por defecto** que podemos omitir al llamar la función.
- Se pueden combinar varios parámetros con o sin valor por defecto.

```
def presentarse(nombre, edad=18, ciudad="Madrid"):
    print(f"Soy {nombre}, tengo {edad} años y vivo en {ciudad}.")
```

```
presentarse("Luis")
presentarse("Ana", 25)
presentarse("Marta", ciudad="Barcelona")
```

- Observa que podemos usar **keyword arguments** (nombre=valor) para indicar a qué parámetro nos referimos, incluso si no seguimos el orden.

3.5. Buenas prácticas

- Siempre poner los parámetros con valores por defecto **al final** de la lista.
- Usar nombres de parámetros claros y descriptivos.

```
# Correcto
def ejemplo(a, b=10):
  pass

# Incorrecto (da error)
def ejemplo(a=10, b):
  pass
```

3.6 Tipos de argumentos

3.6.1 Argumentos posicionales

- Se pasan **en el mismo orden** que los parámetros de la función.
- El valor de cada argumento se asigna al parámetro correspondiente según su posición.

```
def presentar(nombre, edad):
  print(f"Hola, soy {nombre} y tengo {edad} años.")

presentar("Ana", 25)   # Ana → nombre, 25 → edad
```

3.6.2 Argumentos nombrados (keyword arguments)

- Se pasan **indicando explícitamente el nombre del parámetro**.
- Esto permite **cambiar el orden** y mejorar la legibilidad.

```
presentar(edad=30, nombre="Luis")  # Luis → nombre, 30 → edad
```

3.6.3 Orden de argumentos

- **Primero** los posicionales, **después** los nombrados.
- Mezclar mal puede dar error.

```
def ejemplo(a, b, c=10):
    print(a, b, c)

ejemplo(1, 2)          # a=1, b=2, c=10 (por defecto)
ejemplo(1, 2, c=5)     # a=1, b=2, c=5
# ejemplo(a=1, 2)      # ❌ ERROR: los posicionales deben ir antes de los keyword
```

3.7 Número variable de argumentos

3.7.1 *args → múltiples argumentos posicionales

- Permite pasar **cualquier cantidad de argumentos posicionales** a la función.
- Dentro de la función se reciben como una **tupla**.

```
def sumar_todos(*numeros):
    total = 0
    for n in numeros:
        total += n
    return total

print(sumar_todos(1, 2, 3))     # 6
print(sumar_todos(5, 10, 15, 20)) # 50
```

3.7.2 **kwargs → múltiples argumentos con nombre

- Permite pasar **cualquier cantidad de argumentos con nombre**.
- Dentro de la función se reciben como un **diccionario**.

```
def mostrar_info(**info):
    for clave, valor in info.items():
        print(f"{clave}: {valor}")
```

```
mostrar_info(nombre="Ana", edad=25, ciudad="Madrid")
# Salida:
# nombre: Ana
# edad: 25
# ciudad: Madrid
```

- Útil cuando no sabemos de antemano cuántos argumentos vamos a recibir.

Formas de usar **kwargs dentro de la función

A). Recorrerlo con .items() (la más común)

Te da las **claves y valores** juntos.

```
def mostrar_info(**kwargs):
  for clave, valor in kwargs.items():
    print(f"{clave}: {valor}")

mostrar_info(nombre="Ana", edad=25, ciudad="Madrid")
```

Salida:

```
nombre: Ana
edad: 25
ciudad: Madrid
```

B) Acceder por clave (como un diccionario normal)

Como kwargs es un diccionario, puedes acceder a un valor con kwargs['clave'].

```
def saludar(**kwargs):
  print("Hola", kwargs["nombre"])

saludar(nombre="Carlos")
```

Salida:

Hola Carlos

Si la clave no existe, da error. Mejor usar .get() (devuelve None si no existe):

```
def saludar(**kwargs):
   print("Hola", kwargs.get("nombre", "invitado"))

saludar()
```

Salida:

Hola invitado

C) Usar solo las claves (.keys()) o solo los valores (.values())

```
def mostrar_claves(**kwargs):
   print("Claves:", list(kwargs.keys()))
   print("Valores:", list(kwargs.values()))

mostrar_claves(a=1, b=2, c=3)
```

Salida:

```
Claves: ['a', 'b', 'c']
Valores: [1, 2, 3]
```

D) Tratar kwargs como un diccionario completo {}

Puedes modificarlo, añadir o borrar claves.

```
def modificar(**kwargs):
   datos = dict(kwargs)  # copiamos el diccionario
   datos["extra"] = "nuevo valor"
   print(datos)
```

```
modificar(x=10, y=20)
```

Salida:

```
{'x': 10, 'y': 20, 'extra': 'nuevo valor'}
```

E) Pasarlo a otra función

Como es un diccionario, puedes “reenviarlo” a otra función usando **.

```
def detalles(**kwargs):
  print("Detalles:", kwargs)

def usuario(**kwargs):
  detalles(**kwargs)  # reenvía los mismos kwargs

usuario(nombre="Lucía", edad=30)
```

Salida:

```
Detalles: {'nombre': 'Lucía', 'edad': 30}
```

Resumen **kwargs

Dentro de la función, **kwargs es un diccionario, y puedes:

1. Recorrerlo con .items() → (clave, valor).
2. Acceder por clave: kwargs['clave'] o kwargs.get('clave').
3. Obtener claves o valores: kwargs.keys(), kwargs.values().
4. Manipularlo como un diccionario {} normal.
5. Pasarlo a otra función con **kwargs.

Resumen general de parámetros

- Los parámetros permiten pasar información a la función.
- Pueden ser simples, múltiples, opcionales o con valor por defecto.

- Los argumentos se pasan en el mismo orden de definición o usando keyword arguments.
- **Argumentos posicionales:** asignación por posición.
- **Keyword arguments:** asignación por nombre, más flexibles.
- **Orden:** posicionales primero, luego nombrados.
- *args → tupla de múltiples argumentos posicionales.
- **kwargs → diccionario de múltiples argumentos con nombre.

4. Ámbito de variables y buenas prácticas

4.1. Ámbito de variables

El **ámbito** (scope) indica **dónde se puede usar una variable** en el código. En Python, las variables pueden ser **locales** o **globales**.

4.1.1. Variables locales

- Son variables **definidas dentro de una función**.
- Solo existen **dentro de esa función**.
- No afectan ni son afectadas por variables con el mismo nombre fuera de la función.

```
def sumar():
   x = 5  # variable local
   y = 10
   return x + y

print(sumar())  # 15
# print(x)  # ERROR: x no existe fuera de la función
```

4.1.2. Variables globales

- Son variables **definidas fuera de cualquier función**.
- Pueden ser usadas dentro de funciones, pero **modificarlas dentro de la función requiere global**.

```
z = 100  # variable global

def mostrar_global():
    print(z)  # puede leer la variable global

mostrar_global()  # 100
```

4.1.3. Modificar variables globales

Para cambiar una variable global dentro de una función, se usa global. **Advertencia:** No es recomendable abusar de esto, puede hacer el código confuso.

```
contador = 0

def incrementar():
    global contador
    contador += 1

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

4.2. Buenas prácticas en funciones

1. **Una función = una tarea**
    a. Cada función debe tener un propósito claro y no mezclar responsabilidades.
2. **Nombres claros**
    a. Usa nombres descriptivos para la función y los parámetros.
    def calcular_area_circulo(radio):
        pass

3. **Evitar modificar variables globales**
    a. Mejor devolver valores con return y asignarlos fuera de la función.
4. **Documentar la función**
    a. Agrega un docstring para explicar qué hace la función y sus parámetros.

```
def sumar(a, b):
    """Devuelve la suma de a y b"""
```

```
    return a + b
```

5. **Evitar funciones demasiado largas**
    a. Si una función tiene muchas líneas o hace varias cosas, es mejor dividirla en varias funciones pequeñas.

Resumen de esta parte

- Variables locales existen solo dentro de la función.
- Variables globales existen en todo el programa, pero modificarlas dentro de la función puede ser peligroso.
- Buenas prácticas: funciones claras, con un solo propósito, con nombres descriptivos y documentadas.

5. Funciones como objetos (introducción)

En Python, **las funciones son objetos**. Esto significa que se pueden:

- **Asignar a variables**
- **Pasar como argumentos** a otras funciones
- **Devolver funciones** desde otras funciones

5.1. Asignar funciones a variables

Podemos guardar una función en otra variable y llamarla usando esa variable.

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

# Asignar a otra variable
mi_funcion = saludar

print(mi_funcion("Ana"))  # Hola, Ana!
```

- Esto muestra que **el nombre de la función es un objeto** que se puede almacenar y usar.

5.2. Pasar una función como argumento

Algunas funciones de Python aceptan **otras funciones como parámetros**, lo que permite realizar operaciones más flexibles.

5.2.1 Ejemplo con map

map(func, iterable) aplica la función func a cada elemento del iterable.

```
def cuadrado(x):
  return x * x

numeros = [1, 2, 3, 4, 5]
resultado = list(map(cuadrado, numeros))
print(resultado)  # [1, 4, 9, 16, 25]
```

- cuadrado se pasa como **función**, no como cuadrado().

5.2.2 Ejemplo con sorted y key

Podemos usar una función para **definir el criterio de ordenación**.

```
def longitud(texto):
  return len(texto)

palabras = ["manzana", "pera", "uva", "banana"]
ordenadas = sorted(palabras, key=longitud)
print(ordenadas)  # ['uva', 'pera', 'manzana', 'banana']
```

- key=longitud le dice a sorted que use la longitud de cada palabra para ordenar.

Resumen

- En Python, **las funciones son objetos**.
- Podemos **asignarlas a variables**, **pasarlas como argumentos** y usar funciones integradas (map, sorted) que aceptan otras funciones.

- Esto es muy útil para escribir **código más modular y reutilizable**.

6. Funciones en Python – Funciones lambda, map, reduce y sorted

6.1. map()

Qué hace

- Aplica una **función a cada elemento** de un iterable (lista, tupla, etc.)
- Devuelve un **map object**, que podemos convertir en lista con list().

Sintaxis

map(función, iterable)

Ejemplo

```
numeros = [1, 2, 3, 4, 5]

# Función que multiplica por 2
def multiplicar_por_dos(x):
   return x * 2

resultado = list(map(multiplicar_por_dos, numeros))
print(resultado)  # [2, 4, 6, 8, 10]

# Con lambda
resultado2 = list(map(lambda x: x*3, numeros))
print(resultado2)  # [3, 6, 9, 12, 15]
```

6.2. filter()

Qué hace

- Filtra los elementos de un iterable según una **condición**.
- Devuelve solo los elementos que **cumplen la condición**.

Sintaxis

filter(función, iterable)

- La función debe devolver True o False para cada elemento.

Ejemplo

```
numeros = [1, 2, 3, 4, 5, 6]

# Filtrar números pares
def es_par(x):
   return x % 2 == 0

pares = list(filter(es_par, numeros))
print(pares)  # [2, 4, 6]

# Con lambda
pares2 = list(filter(lambda x: x%2==0, numeros))
print(pares2)  # [2, 4, 6]
```

6.3. sorted()

Qué hace

- Ordena los elementos de un iterable y **devuelve una lista nueva**.
- Se puede ordenar de menor a mayor por defecto, o con un criterio personalizado usando key.
- Se puede ordenar de forma descendente con reverse=True.

Sintaxis

sorted(iterable, key=None, reverse=False)

Ejemplo básico

```
numeros = [5, 2, 9, 1, 7]
print(sorted(numeros))         # [1, 2, 5, 7, 9]
print(sorted(numeros, reverse=True))  # [9, 7, 5, 2, 1]
```

Ejemplo con key

```
palabras = ["manzana", "uva", "pera", "banana"]

# Ordenar por longitud de la palabra
ordenadas = sorted(palabras, key=len)
print(ordenadas)  # ['uva', 'pera', 'manzana', 'banana']

# Ordenar por última letra
ordenadas2 = sorted(palabras, key=lambda palabra: palabra[-1])
print(ordenadas2)  # ['pera', 'banana', 'manzana', 'uva']
```

Resumen rápido de map, filter y sorted

Función	Qué hace	Devuelve
map	Aplica una función a cada elemento	Un map object (iterable)
filter	Filtra elementos según condición	Un filter object (iterable)
sorted	Ordena elementos según criterio	Una lista nueva

6.4. ¿Qué es una función lambda?

- Es una **función anónima**, es decir, **no tiene nombre**.
- Se usa para crear funciones **muy cortas y simples**, generalmente de una sola línea.
- Sintaxis:

lambda argumentos: expresión

- El resultado de la expresión se devuelve automáticamente.
- Ideal para usar en funciones que aceptan otras funciones (map, filter, sorted).

6.5. Ejemplo básico

```
# Función normal
def cuadrado(x):
   return x * x

# Función lambda equivalente
cuadrado_lambda = lambda x: x * x

print(cuadrado_lambda(5))  # 25
```

- Se puede asignar a una variable o usar directamente sin nombre.

6.6. Uso con map

```
numeros = [1, 2, 3, 4, 5]
resultado = list(map(lambda x: x*2, numeros))
print(resultado)  # [2, 4, 6, 8, 10]
```

- Aquí usamos la lambda directamente dentro de map sin crear una función por separado.

6.7. Uso con sorted y key

```
palabras = ["manzana", "pera", "uva", "banana"]
# Ordenar por longitud usando lambda
ordenadas = sorted(palabras, key=lambda palabra: len(palabra))
print(ordenadas)  # ['uva', 'pera', 'manzana', 'banana']
```

- lambda palabra: len(palabra) define **sobre la marcha** cómo se ordena la lista.

Resumen de lambda

- **Lambda** = función pequeña, anónima y de una sola línea.
- Útil para operaciones rápidas o para pasar funciones como argumentos.
- Ideal con map, filter, sorted y otras funciones que aceptan funciones como parámetros.