8. Funciones

Una **función** es un bloque de código reutilizable que realiza una tarea específica. En lugar de escribir el mismo código varias veces, podemos **definir una función** y llamarla cuando sea necesario.

8.1. Utilidades

- Para organizar el código en partes más pequeñas y manejables.
- Para evitar la repetición de código (principio **DRY**: *Don't Repeat Yourself*).
- Para hacer el código más modular, claro y fácil de entender.
- Para reutilizar lógica sin necesidad de copiar y pegar código.

8.2. Beneficios

1. **Reutilización:** Podemos llamar la misma función en diferentes partes del código.
2. **Modularidad**: Se pueden dividir programas complejos en funciones más pequeñas y manejables.
3. **Claridad**: Hace que el código sea más fácil de entender y mantener.
4. **Mantenimiento:** Si hay un error en una función, solo se corrige en un solo lugar.
5. **Evita redundancia:** Se evita escribir el mismo código varias veces.

8.3. Definición y llamado de una función

Para crear una función, usamos la palabra clave **def**, seguida del **nombre** de la función y paréntesis **()**.

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

Para ejecutar la función, simplemente la llamamos por su nombre con **()**.

```
print("Este es mi programa")
saludar()
```

Resultado por pantalla

```
>>> %Run -c $EDITOR_CONTENT
 Este es mi programa
 ¡Hola usuario, bienvenido!
>>>
```

8.4. Parametrización y retorno

Hasta ahora, nuestras funciones han sido bastante básicas, pero Python nos permite hacerlas más dinámicas mediante **parámetros** y **valores de retorno**.

8.4.1. Parámetros en funciones

Las funciones pueden recibir valores como **parámetros**. Estos valores permiten que la función realice cálculos o procesos basados en datos externos.

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

En este caso, la función saludar() recibe una variable llamada nombre como parámetro.

```
print("Bienvenidos a mi programa")
saludar("Ana")
saludar("Carlos")
```

Al llamarla, le envíamos un parámetro para que haga uso de él dentro de la función. Y cuando ejecutamos el programa tenemos este resultado:

```
>>> %Run -c $EDITOR_CONTENT
 Bienvenidos a mi programa
 Hola, Ana!
 Hola, Carlos!

>>>
```

En este ejemplo le estamos enviando un nombre ya establecido pero también podemos pedir al usuario un nombre y usarlo como parámetro, veamos como:

```
print("Bienvenidos a mi programa")
nombre = input("Ingrese su nombre: ")
saludar(nombre)
```

```
>>> %Run -c $EDITOR_CONTENT
 Bienvenidos a mi programa
 Ingrese su nombre:
```

```
>>> %Run -c $EDITOR_CONTENT
 Bienvenidos a mi programa
 Ingrese su nombre: Leopoldo
 Hola, Leopoldo!
>>>
```

Múltiples parámetros

Podemos definir funciones que reciben más de un parámetro separándolos con comas, estos pueden ser de cualquier tipo.

Declaramos una función que se llamará sumar y recibirá dos números para ser sumados.

```
def sumar(a, b):
    print(f"La suma es: {a + b}")
```

```
print("Bienvenidos a mi programa")
nombre = input("Ingrese su nombre: ")
saludar(nombre)
numero1 = int(input("Ingrese un numero: "))
numero2 = int(input("Ingrese otro numero: "))
sumar(numero1, numero2)
```

```
>>> %Run -c $EDITOR_CONTENT
 Bienvenidos a mi programa
 Ingrese su nombre: Leopoldo
 Hola, Leopoldo!
 Ingrese un numero: 5
 Ingrese otro numero: 10
 La suma es: 15
>>>
```

Las funciones pueden recibir valores como parámetros. En este caso, la función sumar tiene dos parámetros **(a y b)**. Cuando llamamos a **sumar(numero1, numero2)**, los valores almacenados en **numero1** y **numero2** se copian en **a** y **b**.
No importa que los nombres sean diferentes, ya que los parámetros de la función son solo nombres temporales usados dentro de su propio contexto.

8.4.2. Retorno de las funciones

A veces necesitamos que una función **devuelva** un resultado en lugar de solo imprimirlo. Para esto usamos la palabra clave **return**.

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

Declaramos la función y en lugar de imprimir el resultado se devuelve, el cual es capurado por otra variable:

```python
def configurar_modo(modo=None):
    if modo is None:
        modo = "Normal"
    print(f"Modo seleccionado: {modo}")

configurar_modo("Avanzado")
configurar_modo()
```

```python
resultado = sumar(4, 7)
print(f"El resultado es {resultado}")
```

Lo mismo podría realizarse si se pidiera números al usuario como el ejemplo anterior.

Múltiples retornos

En Python, una función puede devolver más de un valor separándolos por comas. Cuando llamamos a la función, podemos capturar cada valor en una variable diferente.

```python
def operaciones(a, b):
    suma = a + b
    resta = a - b
    return suma, resta
```

Esta función retorna dos resultados, el de suma y el de resta que son capturados por ambas variables en nuestro programa. Es fundamental el orden de lo que retorna y las variables que reciben. Si primero retorno la suma y luego la resta, cuando capturo resultados debo especificar ese mismo orden

```python
resultado_suma, resultado_resta = operaciones(10, 5)

print("La suma es:", resultado_suma)
print("La resta es:", resultado_resta)
```

8.4.3. Parámetros opcionales en python

En Python, una función puede tener parámetros **opcionales**, lo que significa que pueden omitirse al llamar a la función. Esto se logra asignando un **valor por defecto** en la definición de la función.

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

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

```
Hola, Ana!
Hola, Usuario!

>>>
```

Se pueden combinar parámetros obligatorios y opcionales:

```python
def presentar(nombre, edad=18):
    print(f"Nombre: {nombre}, Edad: {edad}")

presentar("Carlos", 25)
presentar("Lucia")
```

```
Nombre: Carlos, Edad: 25
Nombre: Lucia, Edad: 18

>>>
```

Uso de None como Valor por Defecto

A veces, se usa **None** como valor por defecto y luego se verifica dentro de la función:

```
Modo seleccionado: Avanzado
Modo seleccionado: Normal

>>>
```

Los parámetros opcionales hacen que las funciones sean más flexibles y fáciles de usar, permitiendo que el código sea más limpio y adaptable a diferentes casos sin necesidad de sobrecargar la definición de funciones con variantes innecesarias.

8.4.4. Parámetros y tipos de retorno especificando tipo de dato

En Python, al definir una función, los **parámetros formales** son los nombres de las variables que la función espera recibir como argumentos. Se pueden **anotar** con un tipo de dato para indicar qué tipo de valor deberían recibir, aunque Python no lo impone estrictamente.

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

En este caso, el parámetro nombre está anotado con str, indicando que se espera una **cadena de texto**.

Especificar tipo de retorno

```python
def duplicar(numero: int) -> int:
    return numero * 2

print(duplicar(5))
```

Aquí numero: int sugiere que el argumento debe ser un número entero, y **-> int** indica que la función debería retornar un entero.

Importante: Estas anotaciones no son obligatorias y Python no las usa para restringir valores en tiempo de ejecución. Es decir que es opcional, son útiles para mejorar la legibilidad del código.

8.5. Variables locales y globales en Python

En Python, las variables pueden tener **alcance local o global**, dependiendo de dónde y cómo se declaren.

8.5.1. Variables Locales

Son aquellas que se **declaran dentro de una función** y **solo pueden usarse dentro de esa función**. No existen fuera de ella.

```
def saludo():
    mensaje = "Hola, bienvenido"
    print(mensaje)
```

Aquí, **mensaje** solo existe dentro de **saludo()**. Si intentamos acceder a **mensaje** fuera de la función, obtenemos un error.

8.5.2. Variables Globales

Son aquellas que se **declaran fuera de cualquier función** y pueden ser **accedidas desde cualquier parte del programa**.

```
nombre = "Ana"
saludar()
```

En el programa declaramos la variable nombre y le asignamos un valor, despues llamamos a la función saludar()

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

nombre es una variable global, por lo que la función tiene acceso a ella.

```
>>> %Run -c $EDITOR_CONTENT
 Hola, Ana
>>>
```

8.5.3. Modificar Variables Globales dentro de una Función (global)

Por defecto, si intentamos modificar una variable global dentro de una función, **Python crea una nueva variable local con el mismo nombre**, en lugar de modificar la global.

```
contador = 0

def incrementar():
    contador = contador + 1
    print(contador)

incrementar()
```

Este código **generará un error**, ya que Python considera que **contador** dentro de la función es una nueva variable local y no sabe que queremos modificar la global.

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

```
contador = 0

def incrementar():
    global contador
    contador += 1


incrementar()
incrementar()
incrementar()
print(contador)
```

```
>>> %Run -c $EDITOR_CONTENT
 3
>>>
```

Buenas Prácticas y Riesgos del Uso de global

- **Evitar modificar variables globales dentro de funciones** porque puede hacer que el código sea difícil de entender y depurar.
- **Usar return en lugar de global** cuando sea posible.

```python
def incrementar(valor):
    return valor + 1

contador = 0
contador = incrementar(contador)
print(contador)
```

Este enfoque es más limpio y seguro, porque mantiene el control sobre las variables sin afectar otras partes del programa.

8.6. Variables mutables y variables inmutables

En Python, los tipos de datos pueden clasificarse en **mutables e inmutables**, dependiendo de si su contenido puede cambiar después de su creación.
Mutable: Se puede modificar después de haber sido creado.
Inmutable: No se puede modificar después de haber sido creado. Si queremos cambiarlo, debemos crear una nueva variable con el nuevo valor.

8.6.1 Pasajes de parámetros por valor y por referencia

Pasaje por valor

En el **pasaje por valor**, la función recibe una **copia** del argumento. Cualquier modificación dentro de la función **no afecta** la variable original.

Pasaje por referencia

En el **pasaje por referencia**, la función recibe una **referencia** a la variable original en memoria. Cualquier modificación dentro de la función **afecta directamente** a la variable original.

En Python, el comportamiento del pasaje de parámetros es un poco más complejo. Aunque parece que las **variables mutables** se pasan por referencia y las **inmutables** por valor, en realidad **Python siempre pasa los argumentos por asignación de referencia**.

Regla general en Python

- **Los objetos inmutables (int, float, str, tuple, etc.)** se comportan **como si fueran pasados por valor**, ya que al modificarlos dentro de la función, se crea un **nuevo objeto en memoria** y la referencia original no cambia.
- **Los objetos mutables (list, dict, set, etc.)** se comportan **como si fueran pasados por referencia**, porque cualquier modificación dentro de la función afectará al objeto original en memoria.