# **Programación de funciones**

Las funciones son fragmentos de código que se pueden ejecutar múltiples veces, además pueden recibir y devolver información para comunicarse con el proceso principal.

# **Definición y llamada**

Se definen con la palabra reservada def, el nombre de la función y unos paréntesis, que también se utilizan para hacer la llamada:

def saludar():

print("Hola! Este print se llama desde la función saludar()")

saludar()

Hola! Este print se llama desde la función saludar()

Dentro de una función podemos utilizar variables y sentencias de control:

def dibujar\_tabla\_del\_5():

for i in range(10):

print("5 \* {} = {}".format(i,i\*5))

dibujar\_tabla\_del\_5()

5 \* 0 = 0

5 \* 1 = 5

5 \* 2 = 10

5 \* 3 = 15

5 \* 4 = 20

5 \* 5 = 25

5 \* 6 = 30

5 \* 7 = 35

5 \* 8 = 40

5 \* 9 = 45

## **Ámbito de las variables**

Una variable declarada en una función no existe en la función principal:

def test():

n = 10

test()

print(n)

---------------------------------------------------------------------------

NameError Traceback (most recent call last)

<ipython-input-4-667d7c7a2c02> in <module>()

----> 1 print(n)

NameError: name 'n' is not defined

Sin embargo, una variable declarada fuera de la función (al mismo nivel), sí que es accesible desde la función:

m = 10

def test():

print(m)

test()

10

Siempre que declaremos la variable antes de la ejecución, podemos acceder a ella desde dentro:

def test():

print(l)

l = 10

test()

10

En el caso que declaremos de nuevo una variable en la función, se creará un copia de la misma que sólo funcionará dentro de la función.

**Por tanto no podemos modificar una variable externa dentro de una función**:

def test():

o = 5 # variable que sólo existe dentro de la función

print(o)

o = 10 # variable externa, no modificable

test()

print(o)

5

10

## **La instrucción global**

Para poder modificar una variable externa en la función, debemos indicar que es global de la siguiente forma:

def test():

global o # variable que hace referencia a la o externa

o = 5

print(o)

o = 10

test()

print(o)

5

5

# **Retorno de valores**

Para comunicarse con el exterior, las funciones pueden devolver valores al proceso principal gracias a la instrucción **return**.

En el momento de devolver un valor, la ejecución de la función finalizará:

def test():

return "Una cadena retornada"

test()

'Una cadena retornada'

Los valores devueltos se tratan como valores literales directos del tipo de dato retornado:

print(test())

Una cadena retornada

Por ejemplo no podemos sumar una cadena con un número:

c = test() + 10

---------------------------------------------------------------------------

TypeError Traceback (most recent call last)

<ipython-input-6-0ac9d7015445> in <module>()

----> 1 c = test() + 10

TypeError: Can't convert 'int' object to str implicitly

También podemos devolver cualquier tipo de colección y manejarla directamente:

def test():

return [1,2,3,4,5]

print(test())

print(test()[-1])

print(test()[1:4])

[1, 2, 3, 4, 5]

5

[2, 3, 4]

Evidentemente es posible asignar el valor retornado a una variable:

lista = test()

print(lista[-1])

5

## **Retorno múltiple**

Una característica interesante, es la posibilidad de devolver múltiples valores separados por comas:

def test():

return "Una cadena", 20, [1,2,3]

test()

('Una cadena', 20, [1, 2, 3])

Estos valores se tratan en conjunto como una tupla inmutable y se pueden reasignar a distintas variables:

cadena, numero, lista = test()

print(cadena)

print(numero)

print(lista)

Una cadena

20

[1, 2, 3]

# **Envío de valores**

Para comunicarse con el exterior las funciones no sólo pueden devolver valores, también pueden recibir información:

def suma(a, b): # valores que se reciben

return a + b

Una cadena

20

[1, 2, 3]

Ahora podemos enviar dos valores a la función:

resultado = suma(2, 5) # valores que se envían

print(resultado)

7

# **Argumentos y parámetros**

En la definición de una función los valores que se reciben se denominan **parámetros**, pero durante la llamada los valores que se envían se denominan **argumentos**.

## **Argumentos por posición**

Cuando enviamos argumentos a una función, estos se reciben por orden en los parámetros definidos. Se dice por tanto que son argumentos por posición:

def resta(a, b):

return a - b

resta(30, 10) # argumento 30 => posición 0 => parámetro a

# argumento 10 => posición 1 => parámetro b

20

## **Argumentos por nombre**

Sin embargo es posible evadir el orden de los parámetros si indicamos durante la llamada que valor tiene cada parámetro a partir de su nombre:

resta(b=30, a=10)

-20

## **Llamada sin argumentos**

Al llamar una función que tiene definidos unos parámetros, si no pasamos los argumentos correctamente provocará un error:

resta()

---------------------------------------------------------------------------

TypeError Traceback (most recent call last)

<ipython-input-4-78c8f433960e> in <module>()

----> 1 resta()

TypeError: resta() missing 2 required positional arguments: 'a' and 'b'

## **Parámetros por defecto**

Para solucionarlo podemos asignar unos valores por defecto nulos a los parámetros, de esa forma podríamos hacer una comprobación antes de ejecutar el código de la función:

def resta(a=None, b=None):

if a == None or b == None:

print("Error, debes enviar dos números a la función")

return # indicamos el final de la función aunque no devuelva nada

return a-b

resta()

Error, debes enviar dos números a la función

# **Paso por valor y referencia**

Dependiendo del tipo de dato que enviemos a la función, podemos diferenciar dos comportamientos:

* **Paso por valor**: Se crea una copia local de la variable dentro de la función.
* **Paso por referencia**: Se maneja directamente la variable, los cambios realizados dentro de la función le afectarán también fuera.

Tradicionalmente:

* **Los tipos simples se pasan por valor**: Enteros, flotantes, cadenas, lógicos...
* **Los tipos compuestos se pasan por referencia**: Listas, diccionarios, conjuntos...

## **Ejemplo de paso por valor**

Como ya sabemos los números se pasan por valor y crean una copia dentro de la función, por eso no les afecta externamente lo que hagamos con ellos:

def doblar\_valor(numero):

numero \*= 2

n = 10

doblar\_valor(n)

print(n)

10

## **Ejemplo de paso por referencia**

Sin embargo las listas u otras colecciones, al ser tipos compuestos se pasan por referencia, y si las modificamos dentro de la función estaremos modificándolas también fuera:

def doblar\_valores(numeros):

for i,n in enumerate(numeros):

numeros[i] \*= 2

ns = [10,50,100]

doblar\_valores(ns)

print(ns)

[20, 100, 200]

Para modificar los tipos simples podemos devolverlos modificados y reasignarlos:

def doblar\_valor(numero):

return numero \* 2

n = 10

n = doblar\_valor(n)

print(n)

20

Y en el caso de los tipos compuestos, podemos evitar la modificación enviando una copia:

def doblar\_valores(numeros):

for i,n in enumerate(numeros):

numeros[i] \*= 2

ns = [10,50,100]

doblar\_valores(ns[:]) # Una copia al vuelo de una lista con [:]

print(ns)

[10, 50, 100]

# **Argumentos indeterminados**

Quizá en alguna ocasión no sabemos de antemano cuantos elementos vamos a enviar a una función. En estos casos podemos utilizar los parámetros indeterminados por posición y por nombre.

## **Por posición**

Para recibir un número indeterminado de parámetros por posición, debemos crear una lista dinámica de argumentos (una tupla en realidad) definiendo el parámetro con un asterisco:

def indeterminados\_posicion(\*args):

for arg in args:

print(arg)

indeterminados\_posicion(5,"Hola",[1,2,3,4,5])

5

Hola

[1, 2, 3, 4, 5]

## **Por nombre**

Para recibir un número indeterminado de parámetros por nombre (clave-valor o en inglés keyword args), debemos crear un diccionario dinámico de argumentos definiendo el parámetro con dos asteriscos:

def indeterminados\_nombre(\*\*kwargs):

print(kwargs)

indeterminados\_nombre(n=5, c="Hola", l=[1,2,3,4,5])

{'n': 5, 'c': 'Hola', 'l': [1, 2, 3, 4, 5]}

Al recibirse como un diccionario, podemos iterarlo y mostrar la clave y valor de cada argumento:

def indeterminados\_nombre(\*\*kwargs):

for kwarg in kwargs:

print(kwarg, "=>", kwargs[kwarg])

indeterminados\_nombre(n=5, c="Hola", l=[1,2,3,4,5])

n => 5

c => Hola

l => [1, 2, 3, 4, 5]

## **Por posición y nombre**

Si queremos aceptar ambos tipos de parámetros simultáneamente, entonces debemos crear ambas colecciones dinámicas. Primero los argumentos indeterminados por valor y luego los que son por clave y valor:

def super\_funcion(\*args,\*\*kwargs):

total = 0

for arg in args:

total += arg

print("sumatorio => ", total)

for kwarg in kwargs:

print(kwarg, "=>", kwargs[kwarg])

super\_funcion(10, 50, -1, 1.56, 10, 20, 300, nombre="Hector", edad=27)

sumatorio => 390.56

nombre => Hector

edad => 27

Los nombres **args** y **kwargs** no son obligatorios, pero se suelen utilizar por convención.

Muchos frameworks y librerías los utilizan por lo que es una buena practica llamarlos así.

# **Funciones recursivas**

Se trata de funciones que se llaman a sí mismas durante su propia ejecución. Funcionan de forma similar a las iteraciones, pero debemos encargarnos de planificar el momento en que dejan de llamarse a sí mismas o tendremos una función rescursiva infinita.

Suele utilizarse para dividir una tarea en subtareas más simples de forma que sea más fácil abordar el problema y solucionarlo.

## **Ejemplo sin retorno**

Cuenta regresiva hasta cero a partir de un número:

def cuenta\_atras(num):

num -= 1

if num > 0:

print(num)

cuenta\_atras(num)

else:

print("Boooooooom!")

print("Fin de la función", num)

cuenta\_atras(5)

4

3

2

1

Boooooooom!

Fin de la función 0

Fin de la función 1

Fin de la función 2

Fin de la función 3

Fin de la función 4

## **Ejemplo con retorno**

El factorial de un número corresponde al producto de todos los números desde 1 hasta el propio número. Es el ejemplo con retorno más utilizado para mostrar la utilidad de este tipo de funciones:

* 3! = 1 x 2 x 3 = 6
* 5! = 1 x 2 x 3 x 4 x 5 = 120

def factorial(num):

print("Valor inicial ->",num)

if num > 1:

num = num \* factorial(num -1)

print("valor final ->",num)

return num

print(factorial(5))

Valor inicial -> 5

Valor inicial -> 4

Valor inicial -> 3

Valor inicial -> 2

Valor inicial -> 1

valor final -> 1

valor final -> 2

valor final -> 6

valor final -> 24

valor final -> 120

120

# **Funciones integradas**

La librería estándar de Python incluye muchas funciones. Las hay para hacer conversiones entre tipos, matemáticas, utilidades...

Aquí un resumen de las más utilizadas incluyendo algunas que ya conocemos:

## **int()**

Transforma una cadena a un entero (si no es posible da error):

n = int("10")

print(n)

10

## **float()**

Transforma una cadena a un flotante (si no es posible da error):

f = float("10.5")

print(f)

10.5

## **str()**

Transforma cualquier valor a una cadena:

c = "Un texto y dos números " + str(10) + " y " + str(3.14)

print(c)

Un texto y dos números 10 y 3.14

## **bin()**

Conversión de entero a binario:

bin(10)

'0b1010'

## **hex()**

Conversión de entero a hexadecimal:

hex(10)

'0xa'

## **int(numero, base)**

Reconversión a entero (base 10):

print(int('0b1010', 2))

print(int('0xa', 16))

10

10

## **abs()**

Valor absoluto de un número (distancia):

abs(-10)

10

## **round()**

Redondeo de un flotante a entero, menor de .5 a la baja, mayor o igual a .5 al alza:

print(round(5.5))

print(round(5.4))

6

5

## **eval()**

Evalúa una cadena como una expresión, acepta variables si se han definido anteriormente:

eval('2 + 5')

7

n = 10

eval('n \* 10 - 5')

95

## **len()**

Longitud de una colección o cadena:

print(len("Una cadena"))

print(len([]))

10

0

## **help()**

Invoca el menú de ayuda del intérprete de Python:

help()

Welcome to Python 3.6's help utility!

If this is your first time using Python, you should definitely check out

the tutorial on the Internet at http://docs.python.org/3.6/tutorial/.

....