Clase 10 - Funciones 2

# **Argumentos y parámetros**

Como sabemos, durante la definición de la función, las variables o valores se denominan parámetros:

**def suma(numero1, numero2):**

**return numero1 + numero2**

Y durante la llamada se le denominan argumentos, como los argumentos de los scripts.

**resultado = suma(7, 5)**

En esta clase estaremos viendo los distintos tipos de argumentos y parámetros.

## **Argumento por posición**

Cuando se envían argumentos a una función, se reciben por orden en los parámetros definidos:

**def suma(numero1, numero2):**

**return numero1 + numero2**

**resultado = suma(7, 5)**

El argumento 7 es la posición 0, por consiguiente es el parámetro de la función numero1, seguidamente el argumento 5 es la posición 1 por consiguiente es el parámetro de la función numero2.

Si tomamos el siguiente ejemplo sabremos que la resta nos dará 3:

**def resta(a, b):**

**return a - b**

**resultado = resta(15, 12)**

**resultado**

Pero, si modificamos el orden de los argumentos nos dará otro resultado:

**resultado = resta(12, 15)**

## **Argumentos por nombre**

Como vimos, si pasamos ordenado el argumento, se verá reflejado ordenadamente el parámetro.

Para cambiar esto se utiliza la asignación de argumentos por nombre, si indicamos durante la llamada que valor tiene cada parámetro a partir de su nombre:

**def resta(a, b):**

**return a - b**

**resultado = resta(b=15, a=12)**

**resultado**

Recordemos que al utilizar argumentos por nombre, no importa el orden:

****

## **Llamada sin argumentos**

Veamos qué pasa si llamamos una función con parámetros ya definidos:

**def resta(a, b):**

**return a - b**

**resultado = resta()**

**Traceback (most recent call last):**

**File "<stdin>", line 1, in <module>**

**TypeError: resta() takes exactly 2 arguments (0 given)**

¿Cómo solucionamos el TypeError al momento de llamar una función sin argumento?

## **Parámetros por defecto**

Python, nos deja asignar unos valores por defecto a los parámetros, es decir, indicarle que tendrán un valor por defecto si no viene ningún valor.

**def resta(a=10, b=5):**

**return a - b**

**resultado = resta()**

**>>>5**

## **Argumento por valor y referencia**

### **Argumentos**

Si hablamos de argumentos tenemos que tener algo en cuenta:

Cuando enviamos información a una función generalmente estos datos se envían por valor.

Eso significa que se crea una copia dentro de la función de los valores que enviamos en sus propias variables. Pero, hay casos excepcionales, las colecciones, listas, diccionarios, conjuntos. Estos datos se envían por referencia.

### **Referencia**

¿Que significa que los conjuntos como listas, tuplas, etc, se envíen por referencia?

Significa que en lugar de una copia dentro de la función, estaremos manejando el dato original, y si lo modificamos también se verá reflejado en el exterior, es decir, en el conjunto original y no en una copia en la función. Esto debido a que hacen referencia a la variable externa, algo así como un acceso directo.

### **Dependiendo del tipo de dato:**

Paso por valor: Se crea una copia local de la variable dentro de la función.

**Los tipos simples se pasan por valor: Enteros, flotantes, cadenas, lógicos…**

Paso por referencia: Se maneja directamente la variable, los cambios realizados dentro de la función le afectarán también fuera.

**Los tipos compuestos se pasan por referencia: Listas, diccionarios, tuplas, conjuntos…**

### **Paso por valor**

Los números se pasan por valor y crean una copia dentro de la función, no les afecta externamente lo que hagamos con ellos en la función:

**def doblar\_valor(numero):**

**numero \*= 2**

**numero = 10**

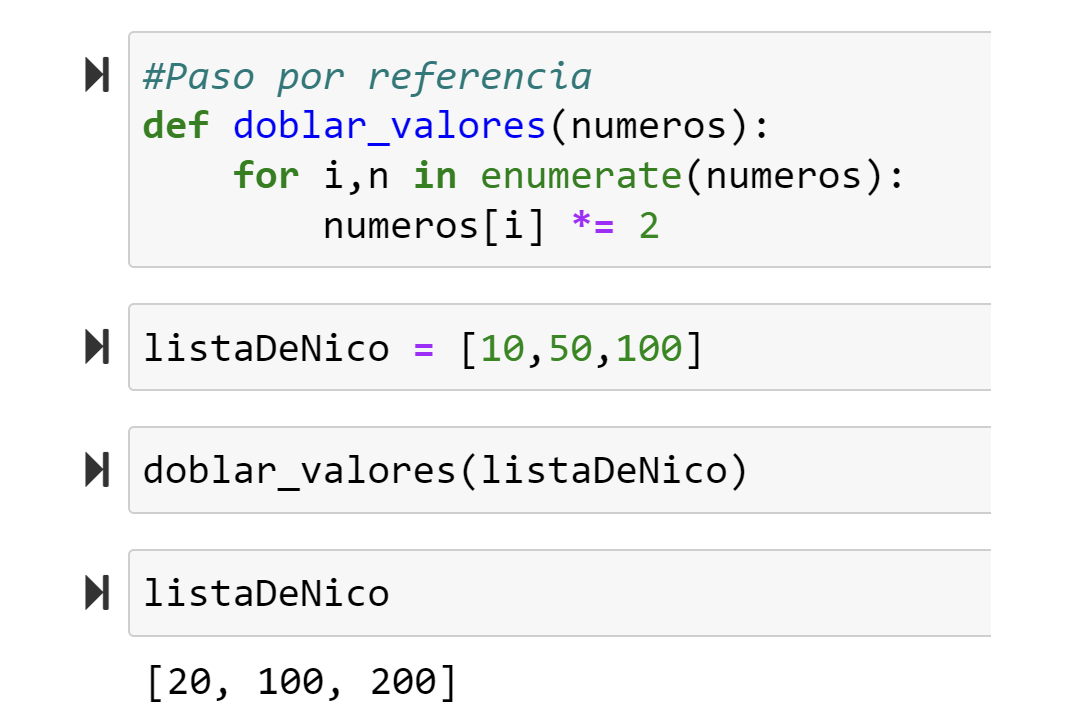
**doblar\_valor(numero)**

**print(numero)**

**>>> 10**

### **Paso por referencia**

Las listas u otras colecciones son del tipo compuesto, por lo que se pasa por referencia, las modificaciones dentro de la función también se harán por fuera.

****

### **Argumentos valor-referencia**

Como vimos, las listas en este caso, hacen referencia a su variable original mientras que los números o tipos de datos más simples “pasan” directamente por valor.

A continuación una pregunta clave...

### Argumentos valor - referencia

La respuesta es NO. En Python no se pueden utilizar punteros como en otros lenguajes.

**>>> def doblar\_valor(numero):**

**return numero \*= 2**

**>>> numero = 10**

**>>> numero = doblar\_valor(numero)**

Aunque podemos utilizar trucos, como devolver el valor modificado dentro de la función y volverlo a asignar a la misma variable en caso de desear que sea “referencia”.

## Argumentos indeterminados

## Uso de \*Args y \*\*Kwargs

¿Para qué se usan?

Lo primero de todo es que en realidad no tienes por qué usar los nombres args o kwargs, ya que se trata de una mera convención entre programadores.

Sin embargo, lo que sí debes usar es el asterisco simple \* o doble \*\*.

Es decir, podrías escribir \*variable y \*\*variables.

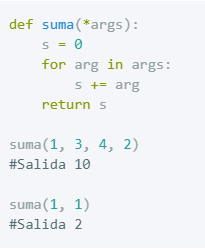
## Uso de \*Args

Gracias a los \*args en Python, podemos definir funciones cuyo número de argumentos es variable. Es decir, podemos definir funciones genéricas que no aceptan un número determinado de parámetros, sino que se “adaptan” al número de argumentos con los que son llamados.

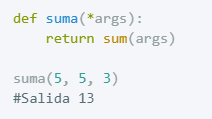
## Ejemplo de uso de \*Args

Veamos aquí como \*args puede ser iterado, ya que en realidad es una tupla.

Por lo tanto, iterando la tupla podemos acceder a todos los argumentos de entrada, y en nuestro caso sumarlos y devolverlos.



Una forma más sencilla de escribir el código anterior. 👏



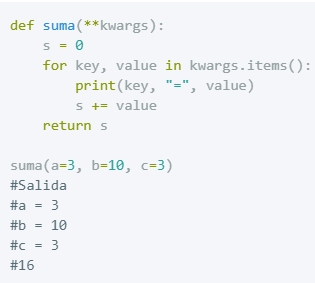
## Uso de \*\*Kwargs

Al igual que en \*args, en \*\*kwargs el nombre es una mera convención entre los usuarios de Python. Puedes usar cualquier otro nombre siempre y cuando respetes el \*\*.

A diferencia de \*args, los \*\*kwargs nos permiten dar un nombre a cada argumento de entrada, pudiendo acceder a ellos dentro de la función a través de un diccionario.

## Ejemplo uso de \*Kwargs

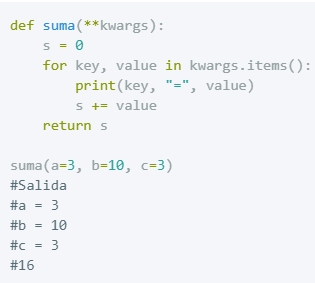
Podemos ver que es posible iterar los argumentos de entrada con items(), y podemos acceder a la clave key (o nombre) y el valor o value de cada argumento.

****

## Ejemplo uso de \*Kwargs

👉El uso de los \*\*kwargs es muy útil si además de querer acceder al valor de las variables dentro de la función, quieres darles un nombre que de una información extra.

Pensar como un diccionario.

****

## 

## **Funciones recursivas**

### **Recursividad**

La recursión o recursividad es un proceso de repetición en el que algo se repite a sí mismo. Es el efecto que sucede cuando se ponen dos espejos frente al otro.

En la informática la recursividad es una técnica muy utilizada, la cual se basa en dividir un problema en partes más pequeñas para poder solucionarlo de forma más simple.

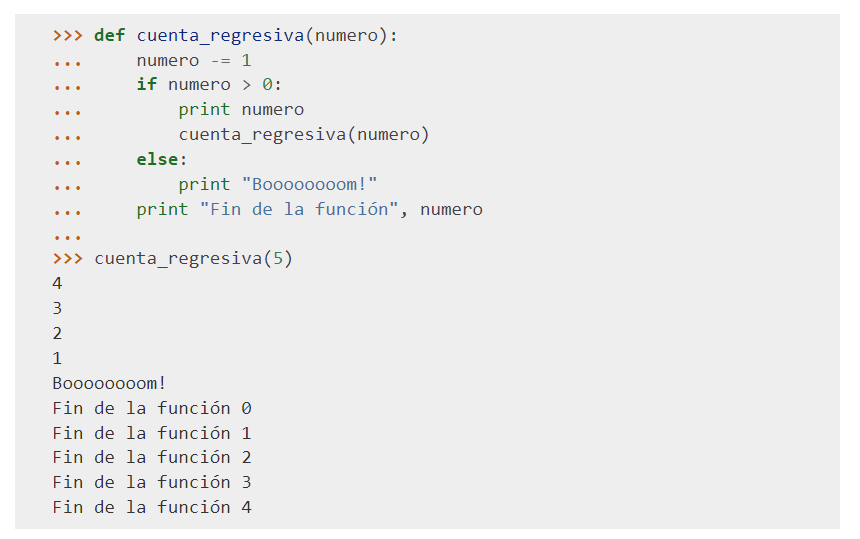
Donde más se suele utilizar es en las funciones.

Cuando una función se llama a sí misma, tenemos una función recursiva con un comportamiento muy similar al de una sentencia iterativa (if, while, etc.) pero debemos encargarnos de planificar el momento en que dejan de llamarse a sí mismas o tendremos una función recursiva infinita.

Podríamos dividir las funciones recursivas en dos: discursivas sin retorno y discursivas con retorno.

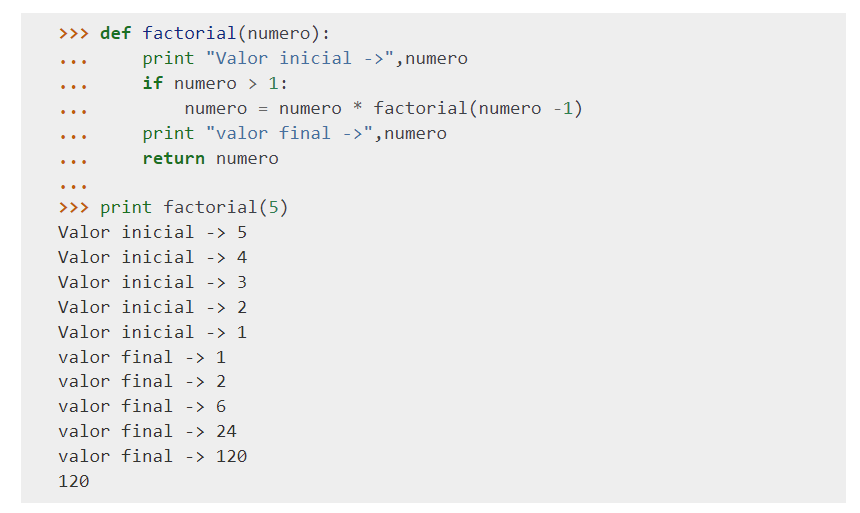
### **Función recursiva sin retorno**

Un ejemplo de una función recursiva sin retorno es el de una cuenta regresiva hasta cero a partir de un número dado.

****

### **Función recursiva con retorno**

Un ejemplo de una función recursiva con retorno, es el ejemplo del cálculo del factorial de un número corresponde al producto de todos los números desde 1 hasta el propio número.

****

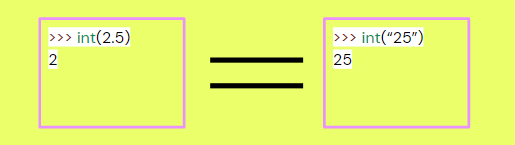
## **Funciones integradas**

Ahora que conocemos las funciones, no podemos acabar sin comentar varias de las integradas en Python. Muchas de ellas son para hacer conversiones entre tipos de datos, otras para manipular información, matemáticas, y demás.

👉Veremos un resumen de las más utilizadas incluyendo algunas ya conocidas.

### **Int**

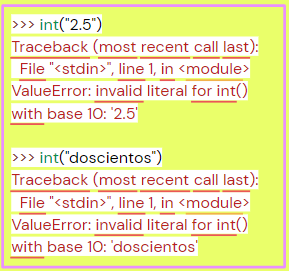
La función int() devuelve un número entero. Es un constructor, que crea un entero a partir de un entero float, entero complex o una cadena de caracteres que sean coherentes con un número entero.

****

### **Int**

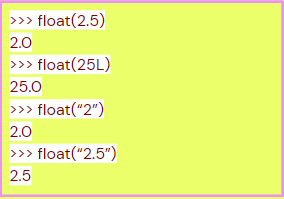
La función int() solo procesa correctamente cadenas que contengan exclusivamente números.

Si la cadena contiene cualquier otro carácter, la función devuelve una excepción ValueError.



### **Float**

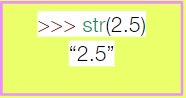
La función float() devuelve un número coma flotante float. Es un constructor, que crea un coma flotante a partir de un entero, entero long, entero float (cadenas de caracteres formadas por números y hasta un punto) o una cadena de caracteres que sean coherentes con un número entero.



### **Str**

La función str() es el constructor del tipo de cadenas de caracteres, se usa para crear un carácter o cadenas de caracteres mediante la misma función str().

Puede convertir un número entero a una cadena de caracteres, de la siguiente forma:



### **Str**

Puede convertir un número float a una cadena de caracteres, de la siguiente forma:

**>>> str(2.5)**

**“2.5”**

**>>> str(-2.5)**

**“-2.5”**

Puede convertir un número complex a una cadena de caracteres, de la siguiente forma:

**>>> str(2.3+0j)**

**“(2.3+0j)”**

Puede convertir un tipo booleano a una cadena de caracteres, de la siguiente forma:

**>>> str(True)**

**“True”**

**>>> str(False)**

**“False”**

### **Round**

La función round() redondea un número flotante a una precisión dada en dígitos decimal (por defecto 0 dígitos). Esto siempre devuelve un número flotante. La precisión tal vez sea negativa.

👉En el siguiente ejemplo veremos el redondeo de un número flotante a entero, mayor o igual a .5 al alza:

**>>> round(2.5)**

**3**

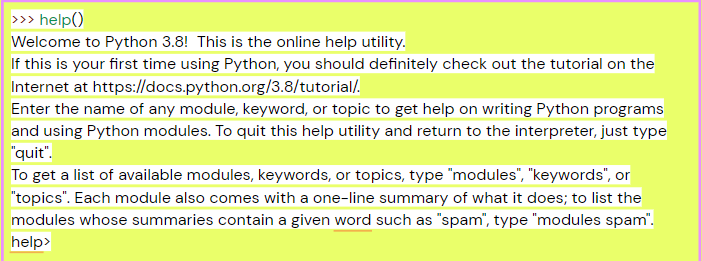
**👉**En este otro ejemplo veremos el redondeo de un número flotante a entero, menor de .5 a la baja:

**>>> round(2.4)**

**2**

### **Help**

Invoca el menú de ayuda del intérprete de Python

****