# Introducción a la Programación con PYTHON

José Rodriguez

## Unidad5 - Funciones

# UNIDAD 5 – FUNCIONES

## 1.- Concepto de función

Una función es un fragmento de código con un nombre asociado que realiza una serie de tareas y devuelve un valor. A los fragmentos de código que tienen un nombre asociado y no devuelven valores se les suele llamar procedimientos. En Python no existen los procedimientos, ya que cuando el programador no especifica un valor de retorno la función devuelve el valor None (nada), equivalente al null de Java.

Una de las utilidades más importantes de las funciones es que nos permiten reutilizar código.

En Python las funciones se declaran de la siguiente forma:

```
def mostrarlista(e1,e2,e3):
        print (e1)
        print (e2)
        print (e3)
```

Es decir, la palabra clave def seguida del nombre de la función y entre paréntesis los argumentos separados por comas. A continuación, en otra línea, indentado y después de los dos puntos tendríamos las líneas de código que conforman el código a ejecutar por la función.

También podemos encontrarnos con una cadena de texto como primera línea del cuerpo de la función. Estas cadenas se conocen con el nombre de docstring (cadena de documentación) y sirven, como su nombre indica, a modo de documentación de la función.

```
def mostrarlista(e1,e2,e3):
        """ Esta funcion muestra por pantalla la lista de parametros
        que le son pasados desde la linea de comandos """
        print (e1)
        print (e2)
        print (e3)
```

Esto es lo que imprime el opeardor ? de iPython o la función help del lenguaje para proporcionar una ayuda sobre el uso y utilidad de las funciones. Todos los objetos pueden tener docstrings, no solo las funciones, como veremos más adelante.

Volviendo a la declaración de funciones, es importante aclarar que al declarar la función lo único que hacemos es asociar un nombre al fragmento de código

que conforma la función, de forma que podamos ejecutar dicho código más tarde referenciándolo por su nombre. Es decir, a la hora de escribir estas líneas no se ejecuta la función. Para llamar a la función (ejecutar su código) se escribiría:

```
mostrarlista("uno","dos","tres")
```

Es decir, el nombre de la función a la que queremos llamar seguido de los valores que queramos pasar como parámetros entre paréntesis. La asociación de los parámetros y los valores pasados a la función se hace normalmente de izquierda a derecha. Así, la variable e1 tomará el valor de “uno”, e2 será “dos” etc.

Sin embargo, también es posible modificar el orden de los parámetros si indicamos el nombre del parámetro al que asociar el valor a la hora de llamar a la función:

```
mostrarlista(e2 = “dos”, e1 = “uno”, e3=”tres”)
```

El número de valores que se pasan como parámetro al llamar a la función tiene que coincidir con el número de parámetros que la función acepta según la declaración de la función. En caso contrario Python se quejará:

```
>>> mi_funcion(“hola”)
Traceback (most recent call last):
File “<stdin>”, line 1, in <module>
TypeError: mi_funcion() takes exactly 2 arguments (1 given)
```

También es posible, no obstante, definir funciones con un número variable de argumentos, o bien asignar valores por defecto a los parámetros para el caso de que no se indique ningún valor para ese parámetro al llamar a la función.

Los valores por defecto para los parámetros se definen situando un signo igual después del nombre del parámetro y a continuación el valor por defecto:

```
def tabla(op1,op2=11):
        for x in range(1,op2):
                print (op1*x)

tabla(8)
```

en este caso, el rango utilizado en el bucle es (1,11) ya que el valor por defecto de op2 es 11. Sin embargo, si se le indica otro valor, será este el que se utilice:

```
def tabla(op1,op2=11):
        for x in range(1,op2):
                print op1*x

tabla(8,20)
```

Para definir funciones con un número variable de argumentos colocamos un último parámetro para la función cuyo nombre debe precederse de un signo *:

```
def datos(e1, e2, *otros):
        for val in otros:
                print (val)

datos(1,2,3,4,5)
```

Esta sintaxis funciona creando una tupla (de nombre otros en el ejemplo) en la que se almacenan los valores de todos los parámetros extra pasados como argumento. Para la llamada, datos(1,2,3,4,5), la tupla otros estaría formada por (3,4,5) así que, la llamada imprimiría los valores 3,4,y 5.

También se puede preceder el nombre del último parámetro con **, en cuyo caso en lugar de una tupla se utilizaría un diccionario. Las claves de este diccionario serían los nombres de los parámetros indicados al llamar a la función y los valores del diccionario, los valores asociados a estos parámetros.

En el siguiente ejemplo se utiliza la función items de los diccionarios, que devuelve una lista con sus elementos, para imprimir los parámetros que contiene el diccionario.

```
def varios(param1, param2, **otros):
        for i in otros.items():
                print i
varios(1, 2, tercero = 3)
```

El resultado sería

```
('tercero', 3)
```

Los que conozcáis algún otro lenguaje de programación os estaréis preguntando si en Python al pasar una variable como argumento de una función estas se pasan por referencia o por valor. En el paso por referencia lo que se pasa como argumento es una referencia o puntero a la variable, es decir, la dirección de memoria en la que se encuentra el contenido de la variable, y no el contenido en si. En el paso por valor, por el contrario, lo que se pasa como argumento es el valor que contenía la variable.

La diferencia entre ambos estriba en que en el paso por valor los cambios que se hagan sobre el parámetro no se ven fuera de la función, dado que los argumentos de la función son variables locales a la función que contienen los valores indicados por las variables que se pasaron como argumento. Es decir, en realidad lo que se le pasa a la función son copias de los valores y no las variables en si.

Si quisiéramos modificar el valor de uno de los argumentos y que estos cambios se reflejaran fuera de la función tendríamos que pasar el parámetro por referencia.

En C los argumentos de las funciones se pasan por valor, aunque se puede simular el paso por referencia usando punteros. En Java también se usa paso por valor, aunque para las variables que son objetos lo que se hace es pasar por valor la referencia al objeto, por lo que en realidad parece paso por referencia.

Hasta ahora hemos diseñado funciones que realizan operaciones y muestran los resultados pero, en muchos casos, lo que nos interesa es poder procesar estos resultados. Parra ello disponemos de la clausula return que devuelve el parámetro indicado al punto en el cual fue llamada la función.

Veamoslo con un ejemplo, la siguiente función obtiene y muestra la serie de fibbonacci hasta un valor indicado

```
def fib(n): # Devolver la serie de Fibonacci hasta n
        """Devolver una lista con los numeros de la serie de Fibonacci hasta n."""
        a= 0
        b= 1
        while b < n:
                print b
                aux= a
                a =b
                b=aux+b

fib(20)
```

El resultado sería

```
1
1
2
3
5
8
13
```

Si redefinimos ahora la función utilizando la cláusula return tendremos el siguiente código

```
def fib(n): # Devolver la serie de Fibonacci hasta n
        """Devolver una lista con los numeros de la serie de Fibonacci hasta n."""
        a= 0
        b= 1
        resultado = []
        while b < n:
                resultado.append(b)
                aux= a
                a =b
                b=aux+b
```

```
        return resultado

resul=fib(20)
print resol
```

Y el resultado sería una lista como la siguiente

```
[1,1,2,3,5,8,13]
```

- La sentencia return devuelve la ejecución al que llamó a la función, devolviendo un valor. Si se utiliza return sin argumento se devuelve None. Si se acaba el código de la función, también se devuelve None.

- La sentencia resultado.append(b) llama a un método del objeto lista resultado. Un método es una función que 'pertenece' a un objeto y se llama obj.nombreMétodo, donde obj es un objeto (que puede resultar de una expresión) y nombreMétodo es el nombre del método definido por el tipo del objeto. Los métodos de diferentes tipos pueden tener el mismo nombre sin ambigüedad. Es posible definir tus propios tipos de objetos y métodos, utilizando clases, según se discute más adelante en esta guía. El método append() (añadir), mostrado en el ejemplo, está definido para objetos lista: Añade un elemento nuevo al final de la lista. En este ejemplo es equivalente a 'resultado = resultado + [b]', pero más eficaz.

A petición popular, se han añadido a Python algunas características comúnmente halladas en los lenguajes de programación funcional y Lisp. Con la palabra clave lambda es posible crear pequeñas funciones anónimas. Ésta es una función que devuelve la suma de sus dos argumentos: 'lambda a, b: a+b'. Las formas lambda se pueden utilizar siempre que se necesite un objeto función. Están sintácticamente restringidas a una expresión simple.

Semánticamente son un caramelo sintáctico para una definición de función normal. Al igual que las definiciones de funciones anidadas, las formas lambda pueden hacer referencia a las variables del ámbito que las contiene:

Veamoslo con algunos ejemplos

```
def incrementar(n):
        return lambda x: x + n
f = incrementar(42)
print f(4)
```

la función incrementar() devuelve otra función f() cuya definición es, en este caso,

f(x)=42 +x

Por tanto, f(4) devolverá 42+4=46