FUNCIONES

En programación estructurada, las funciones son uno de los elementos básicos. Una *función* es un conjunto de sentencias que pueden ser invocadas varias veces durante la ejecución de un programa. Las ventajas de su uso son claras, entre ellas, la minimización de código, el aumento de la legibilidad y la fomentación de la reutilización de código.

Una de las principales diferencias de las funciones en Python con respecto a lenguajes compilados, como C, es que estas no existen hasta que son invocadas y el intérprete pasa a su ejecución. Esto implica que la palabra reservada *def*, empleada para definir una función, es una sentencia más. Como consecuencia de ello, otras sentencias pueden contener una función. Así pues, podemos utilizar una sentencia *if*, definiendo una función cuando una determinada condición se cumple.

Internamente, al definir una función, Python crea un nuevo objeto y le asigna el nombre dado para la función. De hecho, una función puede ser asignada a una variable o almacenada en una lista.

Como hemos comentado previamente, la palabra reservada *def* nos servirá para definir una función. Seguidamente deberemos emplear un nombre y, opcionalmente, una serie de argumentos. Esta será nuestra primera función:

```
def test():
    print("test ejecutada")
```

Para invocar a nuestra nueva función, basta con utilizar su nombre seguido de paréntesis. En lugar de utilizar el intérprete para comprobar su funcionamiento, lo haremos a través de un fichero. Basta con abrir nuestro editor de textos favorito y crear un fichero llamado *test.py* con el siguiente código:

```
def test():
    print("test ejecutada")
test()
nueva = test
nueva()
```

Una vez que salvemos el fichero con el código, ejecutaremos el programa desde la línea de comandos a través del siguiente comando:

```
python test.py
```

Como resultado veremos cómo aparece dos veces la cadena de texto *test ejecutada*.

Paso de parámetros

En los ejemplos previos hemos utilizado una función sin argumentos y, por lo tanto, para invocar a la misma no hemos utilizado ningún parámetro. Sin embargo, las funciones pueden trabajar con parámetros y devolver resultados. En Python, el paso de parámetros implica la asignación de un nombre a un objeto. Esto significa que, en la práctica, el paso de parámetros ocurre por *valor* o por *referencia* en

función de si los tipos de los argumentos son mutables o inmutables. En otros lenguajes de programación, como por ejemplo C, es el programador el responsable de elegir cómo serán pasados los parámetros. Para ilustrar este funcionamiento, observemos el siguiente ejemplo:

```
>>> def test2(a, b):
...         a = 2
...         b = 3
...
>>> c = 5
>>> d = 6
>>> test2(c, d)
>>> print("c={0}, d={1}". format(x, y))
c=5, d=6
```

Como podemos observar, el valor de las variables *c* y *d*, que son inmutables, no ha sido alterado por la función.

Sin embargo, ¿qué ocurre si utilizamos un argumento inmutable como parámetro? Veámoslo a través del siguiente código de ejemplo:

```
>>> def variable(lista):
...         lista[0] = 3
>>> lista = [1, 2, 3]
>>> print(variable(lista))
[3, 2, 3]
```

Efectivamente, al pasar como argumento una lista, que es de mutable, y modificar uno de sus valores, se modificará la variable original pasada como argumento.

Internamente, Python siempre realiza el paso de parámetros a través de referencias, es decir, se puede considerar que el paso se realiza siempre por *variable* en el sentido de que no se hace una copia de las variables. Sin embargo, tal y como hemos mencionado, el comportamiento de esta asignación de referencias depende de si el parámetro en cuestión es mutable o inmutable. Este comportamiento en funciones es similar al que ocurre cuando hacemos asignaciones de variables. Si trabajamos con tipos inmutables y ejecutamos las siguientes sentencias, observaremos cómo el valor de la variable *a* se mantiene:

```
>>> a = 3
>>> b = a
>>> b = 2
>>> print("a={0}, b={1}".format(a, b))
a=3, b=2
```

Por otro lado, si aplicamos el mismo comportamiento a un tipo mutable, veremos cómo varía el resultado:

```
>>> a = [0, 1]
>>> b = a
>>> b[0] = 1
>>> print("a={0}; b={1}".format(a, b))
a=[1, 1]; b=[1, 1]
```

Si necesitamos modificar una o varias variables inmutables a través de una función, podemos hacerlo utilizando una técnica, consistente en devolver una tupla y asignar el resultado de la función a las variables. Partiendo del ejemplo anterior, reescribiremos la función de la siguiente forma:

```
def test(a, b):
    a = 2
    b = 3
    return(a, b)
```

Posteriormente, realizaremos la llamada a la función y la asignación directamente con una sola sentencia:

```
>>> c, d = test(c, d)
```

A pesar de ser el comportamiento por defecto, es posible no modificar el parámetro mutable pasado como argumento. Para ello, basta con realizar, cuando se llama a la función, una copia explícita de la variable:

```
>>> variable(llista[:])
```

Dado el manejo que realiza Python sobre el paso de parámetros a funciones, en lugar de utilizar los términos *por valor* o *por referencia*, sería más exacto decir que Python realiza el paso de parámetros *por asignación*.

Valores por defecto y nombres de parámetros

En Python es posible asignar un valor a un parámetro de una función. Esto significa que, si en la correspondiente llamada a la función no pasamos ningún parámetro, se utilizará el valor indicado por defecto. El siguiente código nos muestra un ejemplo:

```
>>> def fun(a, b=1):
...         print(b)
>>> fun(4)
1
```

Hasta ahora hemos visto cómo pasar diferentes argumentos a una función según la posición que ocupan. Es decir, la correspondencia entre parámetros se realiza según el orden. Sin embargo, Python también nos permite pasar argumentos a funciones utilizando nombres y obviando la posición que ocupan. Comprobémoslo a través del siguiente código:

```
>>> def fun(a, b, c):
...         print("a={0}, b={1}, c={2}".format(a, b, c))
>>> fun(c=5, b=3, a=1)
a=1, b=3, c=5
```

Obviamente, podemos combinar valores por defecto y nombres de argumentos para invocar a una función. Supongamos que definimos la siguiente función:

```
>>> def fun(a, b, c=4):
...         print("a={0}, b={1}, c={2}".format(a, b, c))
```

Dada la anterior función, las siguientes sentencias son válidas:

```
>>> fun(1, 2, 4)
>>> fun(a=1, b=2, c=4)
>>> fun(a=1, b=2)

>>> fun(1, 2)
```