Las funciones en Python (ii)

Tipos de argumentos en las funciones

En el tema anterior sobre funciones hablamos de los argumentos de las funciones: los argumentos posicionales y los argumentos por clave. En realidad existen dos tipos más de argumentos.

Las funciones tienen 4 tipos de *argumentos*:

- **posicionales**
- **por clave**
- **argumentos agrupados**
  - **tupla de argumentos posicionales (*args)**
  - **diccionario de argumentos accedidos por clave (**kwargs)**

Usamos ***args** para representar una tupla arbitraria de argumentos agrupados. No es necesario que el nombre sea **args**:

```
def suma_varios(x, y, *otros):
    print( "x:", x )
    print( "y:", y )
    print("otros:", otros)
```

```
suma_varios(1,2,3,4,5,6,7,8,9)
```

```
x: 1
y: 2
otros: (3, 4, 5, 6, 7, 8, 9)
```

Se pueden definir los argumentos agrupados después de los argumentos posiciones y por clave. Usamos ****kwargs** para representar una lista arbitraria de argumentos agrupados representada como un diccionario. Como en el caso anterior, no es necesario que el nombre sea kwargs:

```
def suma_varios(x, y, *otros, **mas):
    print( "x:", x )
    print( "y:", y )
    print("otros:", otros)
    print("mas:", mas)
```

```
suma_varios(1,2,3,4,5,6,7,8,9, cien = 100 , mil = 1000 )
```

```
x: 1
y: 2
otros: (3, 4, 5, 6, 7, 8, 9)
mas: {'cien': 100, 'mil': 1000}
```

Funciones como argumentos de otras funciones.

Supongamos que tenemos una lista de ciudades que necesitamos 'limpiar' o 'formatear'.

```
lista_ciudades = ['   Madrid', ' BARcelona', 'SeVILLA  ' ]
```

El uso de funciones como argumentos de otras funciones es una característica de los lenguajes funcionales. La función **map** de los lenguajes funcionales también está accesible en Python. Esta función aplica una función a una colección de objetos:
En el siguiente ejemplo, la función `map` aplica la función `upper` a cada una de las ciudades en la lista de ciudades.

```
m1 = map(str.upper , lista_ciudades)
list(m1)
```

```
['   MADRID', ' BARCELONA', 'SEVILLA  ']
```

Podemos aplicar varias funciones a la vez:

```
m2 = map(str.title, map(str.strip , lista_ciudades))
list(m2)
```

```
['Madrid', 'Barcelona', 'Sevilla']
```

Otra alternativa es crear una función que reciba como paraámetro una lista de funciones a aplicar:

```
def formatear(funciones, lista):
    for fun in funciones:
        lista = map(fun,lista)
    return list(lista)
```

```
formatear([str.title, str.strip], lista_ciudades)
```

```
['Madrid', 'Barcelona', 'Sevilla']
```

Funciones anónimas (*lambda functions*)

Las funciones anónimas son aquellas que no tienen nombre y se refieren a una única instrucción. Se declaran con la palabra reservada **lambda**.

Son funciones cortas. Están sintácticamente restringidas a una sola expresión.

- Las funciones Lambda pueden ser usadas en cualquier lugar donde sea requerido un objeto de tipo función. Semánticamente, son solo azúcar sintáctica para definiciones normales de funciones. Al igual que las funciones anidadas, las funciones lambda pueden hacer referencia a variables desde el ámbito que la contiene.

```
# función normal
def producto(a):
    return a * 2

# la función anónima equivalente:
f = lambda x: x * 2
```

```
resultado = map(f, [4, 7])
list(resultado)
```

```
[8, 14]
```

```
resultado = map(lambda x: x * 2, [4, 7])
list(resultado)
```

```
[8, 14]
```

- Las **funciones lambda** se utilizan mucho en análisis de datos ya que es muy usual transformar datos mediante funciones que tienen a otras funciones en sus argumentos.
- Se usan **funciones lambda** en lugar de escribir funciones normales para hacer el código más claro y más corto.