Programación funcional

En Python las funciones son objetos de primera clase, es decir, que pueden pasarse como argumentos de una función, al igual que el resto de los tipos de datos.

```
>>> def aplica(funcion, argumento):
...     return funcion(argumento)
...
>>> def cuadrado(n):
...     return n*n
...
>>> def cubo(n):
...     return n**3
...
>>> aplica(cuadrado, 5)
25
>>> aplica(cubo, 5)
125
```

Funciones anónimas (`lambda`)

Existe un tipo especial de funciones que no tienen nombre asociado y se conocen como **funciones anónimas** o **funciones lambda**.

La sintaxis para definir una función anónima es

```
lambda <parámetros> : <expresión>
```

Estas funciones se suelen asociar a una variable o parámetro desde la que hacer la llamada.

```
>>> area = lambda base, altura : base * altura
>>> area(4, 5)
10
```

Aplicar una función a todos los elementos de una colección iterable (`map`)

`map(f, c)` : Devuelve una objeto iterable con los resultados de aplicar la función `f` a los elementos de la colección `c`. Si la función `f` requiere `n` argumentos entonces deben pasarse `n` colecciones con los argumentos. Para convertir el objeto en una lista, tupla o diccionario hay que aplicar explícitamente las funciones `list()`, `tuple()` o `dic()` respectivamente.

```
>>> def cuadrado(n):
...     return n * n
...
>>> list(map(cuadrado, [1, 2, 3])
[1, 4, 9]

>>> def rectangulo(a, b):
...     return a * b
...
>>> tuple(map(rectangulo, (1, 2, 3), (4, 5, 6)))
```

```
(4, 10, 18)
```

Filtrar los elementos de una colección iterable (filter)

`filter(f, c)` : Devuelve una objeto iterable con los elementos de la colección `c` que devuelven `True` al aplicarles la función `f`. Para convertir el objeto en una lista, tupla o diccionario hay que aplicar explícitamente las funciones `list()`, `tuple()` o `dic()` respectivamente.

`f` debe ser una función que recibe un argumento y devuelve un valor booleano.

```
>>> def par(n):
...     return n % 2 == 0
...
>>> list(filter(par, range(10))
[0, 2, 4, 6, 8]
```

Combinar los elementos de varias colecciones iterables (`zip`)

`zip(c1, c2, ...)` : Devuelve un objeto iterable cuyos elementos son tuplas formadas por los elementos que ocupan la misma posición en las colecciones `c1`, `c2`, etc. El número de elementos de las tuplas es el número de colecciones que se pasen. Para convertir el objeto en una lista, tupla o diccionario hay que aplicar explícitamente las funciones `list()`, `tuple()` o `dic()` respectivamente.

```
>>> asignaturas = ['Matemáticas', 'Física', 'Química', 'Economía']
>>> notas = [6.0, 3.5, 7.5, 8.0]
>>> list(zip(asignaturas, notas))
[('Matemáticas', 6.0), ('Física', 3.5), ('Química', 7.5), ('Economía', 8.0)]
>>> dict(zip(asignaturas, notas[:3]))
{'Matemáticas': 6.0, 'Física': 3.5, 'Química': 7.5}
```

Operar todos los elementos de una colección iterable (`reduce`)

`reduce(f, l)` : Aplicar la función `f` a los dos primeros elementos de la secuencia `l`. Con el valor obtenido vuelve a aplicar la función `f` a ese valor y el siguiente de la secuencia, y así hasta que no quedan más elementos en la lista. Devuelve el valor resultado de la última aplicación de la función `f`.

La función `reduce` está definida en el módulo `functools`.

```
>>> from functools import reduce
>>> def producto(n, m):
...     return n * m
...
>>> reduce(producto, range(1, 5))
24
```

Comprensión de colecciones

En muchas aplicaciones es habitual aplicar una función o realizar una operación con los elementos de una colección (lista, tupla o diccionario) y obtener una nueva colección de elementos transformados. Aunque esto se puede hacer recorriendo la secuencia con un bucle iterativo, y en programación funcional mediante la función `map`, Python incorpora un mecanismo muy potente que permite esto mismo de manera más simple.

Comprensión de listas

[*expresion* `for` *variable* `in` *lista* `if` *condicion*]

Esta instrucción genera la lista cuyos elementos son el resultado de evaluar la expresión *expresion*, para cada valor que toma la variable *variable*, donde *variable* toma todos los valores de la lista *lista* que cumplen la condición *condición*.

```
>>> [x ** 2 for x in range(10)]
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
>>> [x for x in range(10) if x % 2 == 0]
[0, 2, 4, 6, 8]
>>> [x ** 2 for x in range(10) if x % 2 == 0]
[0, 4, 16, 36, 64]
>>> notas = {'Carmen':5, 'Antonio':4, 'Juan':8, 'Mónica':9, 'María': 6,
'Pablo':3}
>>> [nombre for (nombre, nota) in notas.items() if nota >= 5]
['Carmen', 'Juan', 'Mónica', 'María']
```

Comprensión de diccionarios

{*expresion-clave*`:`*expresion-valor* `for` *variables* `in` *lista* `if` *condicion*}

Esta instrucción genera el diccionario formado por los pares cuyas claves son el resultado de evaluar la expresión *expresion-clave* y cuyos valores son el resultado de evaluar la expresión *expresion-valor*, para cada valor que toma la variable *variable*, donde *variable* toma todos los valores de la lista *lista* que cumplen la condición *condición*.

```
>>> {palabra:len(palabra) for palabra in ['I', 'love', 'Python']}
{'I': 1, 'love': 4, 'Python': 6}
>>> notas = {'Carmen':5, 'Antonio':4, 'Juan':8, 'Mónica':9, 'María': 6,
'Pablo':3}
>>> {nombre: nota +1 for (nombre, nota) in notas.items() if nota >= 5])
{'Carmen': 6, 'Juan': 9, 'Mónica': 10, 'María': 7}
```