# Métodos map, filter y reduce.


***Este documento es un resumen de la información que mi persona, Daniela Martínez Madrid, encontró relevante sobre los métodos map, filter y reduce.***

Es importante tener presente que los métodos que vamos a consultar son inmutables, nunca modifican el objeto original, sino que devuelven una copia modificada que puede ser almacenada en otra variable.

## Método Map.

Es una función de transformación que permite procesar y transformar elementos de forma iterada, sin hacer uso de un for o loop. Los elementos del nuevo objeto iterable son producto de una función aplicada a cada uno de los elementos originales.

La función map tiene la siguiente presentación de formato:

```
Python

map(function, iterable[, iterable1, iterable2,..., iterableN])
```

**Ejemplo 1:** En nuestro primer ejemplo vamos a definir una función para ver como se aplica el método map a esta, acompañada de una lista de números:

```
17  def square(number):
18      return number ** 2
19
20  numbers = [1, 2, 3, 4, 5]
21
22  squared = map(square, numbers)
23  print(list(squared))
```

Retorna: `[1, 4, 9, 16, 25]`

**Ejemplo 2:** Para hacer menos uso de la memoria, podemos apoyarnos en las funciones lambda y generar el mismo objeto, sin la necesidad de guardar una función, que en el ejemplo anterior se llama “square”:

```
26  numbers1 = [0, 1, 3, 5, 7]
27  squareds = list(map(lambda x: x ** 2, numbers1))
28  print(squareds)
```

Retorna: `[0, 1, 9, 25, 49]`

**Observación:** Notemos que en los dos ejemplos anteriores tuvimos que dar el formato de lista a nuestro nuevo objeto iterable, pues de otra forma solo nos mostraría el lugar de la memoria donde queda guardado.

**Ejemplo 3:** También podemos hacer uso de las funciones internas que Python ya trae consigo, como mostraremos con la función “int”, cambiaremos de formato todos los números ingresados como tipo “string” a formato entero tipo “int”:

```
31  stri_nums = ['0', '1', '2', '3']
32  int_nums = print(list(map(int,stri_nums)))
```

Retorna: `[0, 1, 2, 3]`

**Ejemplo 4:** Además, Map permite hacer operaciones con múltiples intputs iterables, por ejemplo, escogeremos la función “pow” para calcular la potencia de cada número en un primer objeto iterable (llamado “firts_it”), a una potencia indicada en un segundo objeto iterable (llamado “second_it”):

```
35  firts_it = [1 ,2 , 3]
36  second_it = [3 ,4, 2, 10, 7]
37
38  print(list(map(pow ,firts_it, second_it)))
```

Retorna: `[1, 16, 9]`

Observemos que la operación únicamente se realiza para los primeros 3 elementos de cada lista, que es donde tiene sentido realizarse.

Las operaciones de múltiples inputs también se pueden realizar haciendo uso de una función lambda, por ejemplo, si queremos sumas los elementos de dos objetos iterables, podemos usar el siguiente código:

```
42  print(list(map(lambda x, y: x - y, [2, 4, 6], [1, 3, 5])))
```

Retorna: `[1, 1, 1]`

La función Map se vuelve más interesante, cuando en el segundo parámetro seleccionamos una lista de funciones, y no de valores como lo habíamos hecho en los ejemplos anteriores. Para ilustrar esto, consideramos las funciones elevar al cuadrado y al cubo, consideramos el siguiente código:

**Ejemplo 5:**

```
integers=[1, 2, 3, 4]
def square(number):
    return number ** 2


def cube(number):
    return number ** 3

functions = [square, cube]
for e in integers:
    valors = list(map(lambda number : number(e), functions ))
    print(valors)
```

Retorna:

```
[1, 1]
[4, 8]
[9, 27]
[16, 64]
```

En este ejemplo tuvimos que crear un For para iterar los valores en la lista de funciones y retornando por cada For una lista de dos entradas. Este For se puede evitar, redefiniendo la función con el retorno de dos valores, como se muestra a continuación:

```
numbers2 = [0, 1, 2]
def powers(number):
    return number**2, number**3

print(list(map(powers, numbers2)))
```

Retorna:

```
[(0, 0), (1, 1), (4, 8)]
```

En este caso, el retorno es una lista de tuplas, donde la primera entrada corresponde a el cuadrado y el segundo al cubo de cada elemento de la lista.

## Método Filter.

Es un método que permite aplicar una función booleana a un objeto iterable y nuevamente generar un objeto iterable. Los elementos del nuevo objeto iterable son obtenidos haciendo un filtro de la original y retornando los valores True modificados de la función Booleana.

La presentación de la función filter es:

```
1.  filter(una_funcion, una_lista)
```

Por ejemplo, supongamos que queremos filtrar de una lista de números aquellos que sean pares:

```
66  numbers3 = [1, 2, 3, 4, 5, 6, 7, 8, 9]
67  pares = list(filter(lambda x_: x % 2 == 0, numbers3))
68  print(pares)
```

Retorna: `[2, 4, 6, 8]`

Se puede combinar el uso de la función Map y Reduce, cuando queremos aplicar una función a un filtro de un objeto iterado, y no a todos los elementos, por ejemplo, supongamos que de una lista queremos sacar la raíz de una lista de números, entonces debemos de ser cuidadosos filtrando solo aquellos que son positivos:

```
71  import math
72
73
74  def is_positive(number):
75      return number >= 0
76
77  def sanitized_sqrt(numbers):
78      cleaned_iter = map(math.sqrt, filter(is_positive, numbers))
79      return list(cleaned_iter)
80
81
82  squared_numbers = [25, 9, 81, -16, 0]
83  print(sanitized_sqrt(squared_numbers))
```

Retorna: `[5.0, 3.0, 9.0, 0.0]`

## Método Reduce.

Es un método que se aplica a un objeto iterable y produce solo valor acumulativo.

La función reduce está incluida en la biblioteca Functools, esta biblioteca permite realizar operaciones con funciones; modificar el comportamiento de una sin tener

que perpetuar su estructura código. Además, está diseñada para funciones de orden superior, que son las que actúan o retornan otras funciones. Esto ayuda a ahorrar código y optimizar recursos.

**Ejemplo:** Supongamos que queremos sumar todos los elementos de una lista, sin necesidad de usar un bucle.

```
86  from functools import reduce
87  valores = [2, 4, 6, 5, 4]
88  sum = reduce(lambda x, y: x+y, valores)
89  print(sum)
```

Retorna:

```
21
```