

Manipulación de Colecciones

Map, Filter, Zip y Reduce





Introducción

Una vez presentadas las funciones de primer orden, funciones lambda y funciones de orden superior, se desarrollan a continuación a través de ejemplos los procedimientos para manipular grandes colecciones de datos:

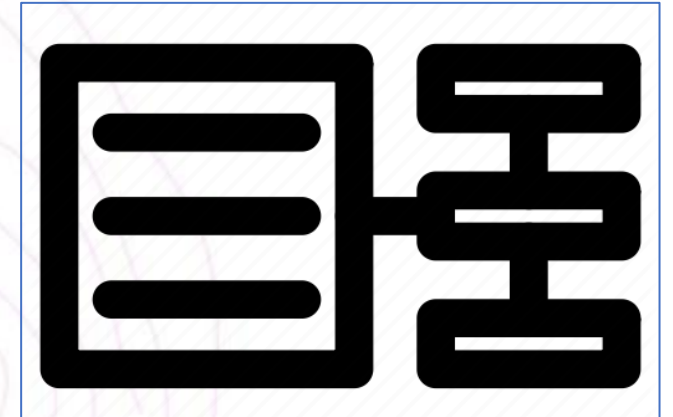
- Map
- Filter
- Reduce
- Zip





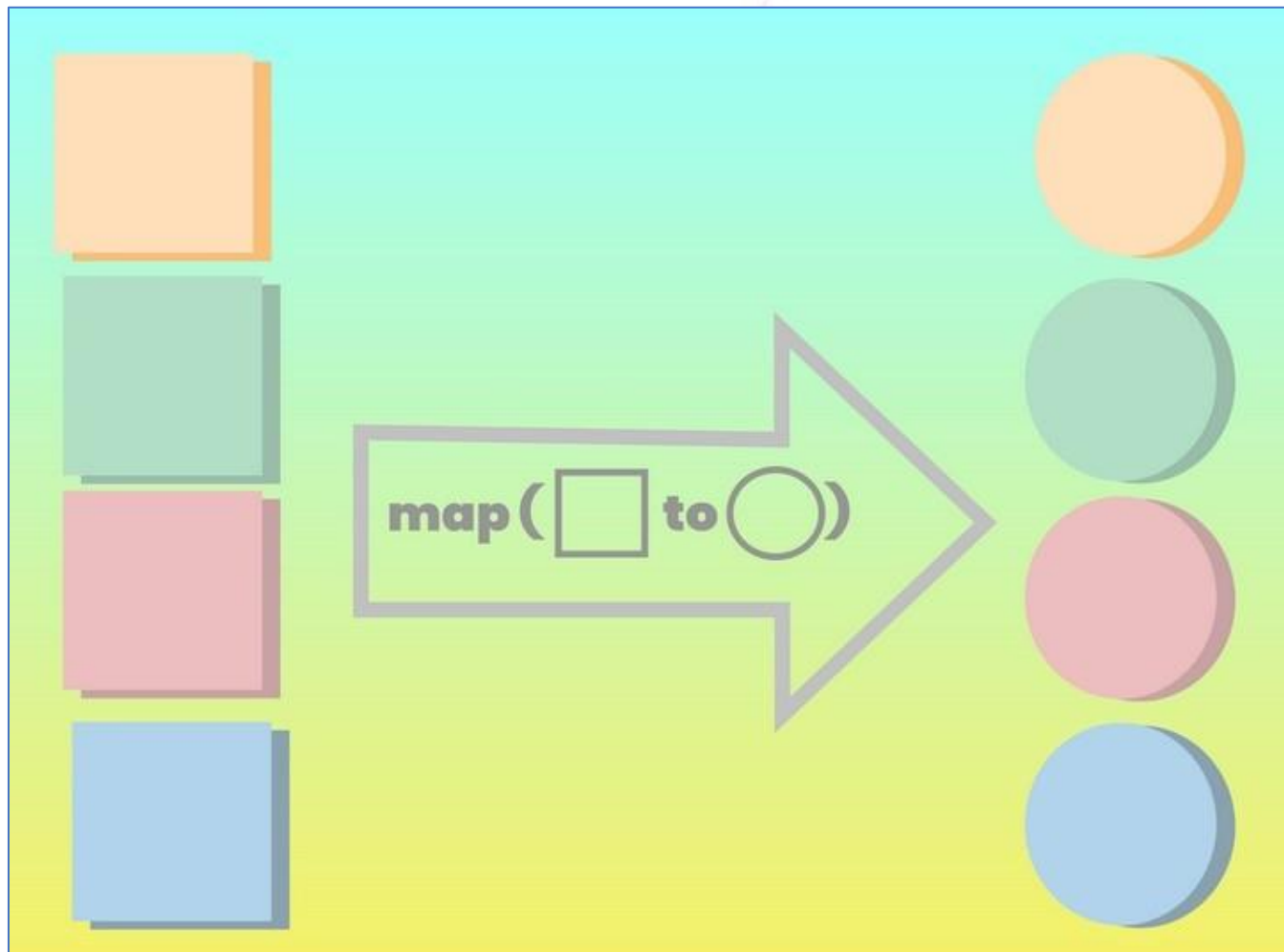
Función Map

- La función **map** nos permite aplicar una función sobre cada uno de los elementos de un colección (listas, tuplas, etc, ...).
- Haremos uso de esta función siempre que tengamos la necesidad de transformar el valor de cada elemento en otro.





Función Map





Función Map

- La estructura de la función es la siguiente:

```
map(función a aplicar, objeto iterable)
```

- La *función a aplicar* debe retornar un nuevo valor. Es a partir de estos nuevos valores que obtendremos una nueva colección.



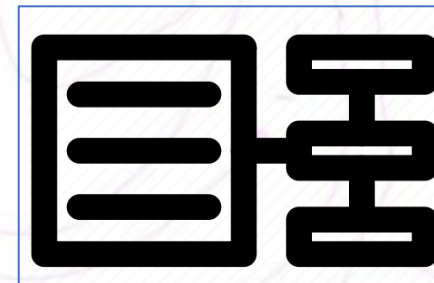
Ejemplo Función Map

```
1 #Obtener el cuadrado de todos los elementos en la lista.
2
3 def cuadrado(elemento=0):
4     return elemento * elemento
5
6 lista = [1,2,3,4,5,6,7,8,9,10]
7 resultado = list( map(cuadrado, lista) )
8 print(resultado)
```



Ejemplo Función Map

- A Partir de la versión 3 de Python, la función **map** retorna un objeto **map object**. Objeto que fácilmente podemos convertir a una lista.
- En este caso, como la función que aplicamos sobre los elementos, es una función sencilla, podemos reemplazarla por una función **lambda**.

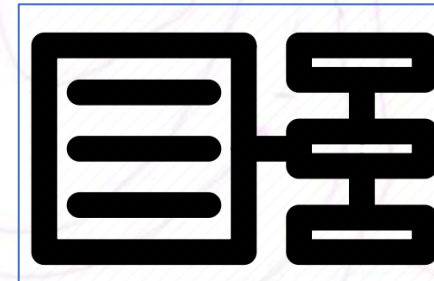




Ejemplo Función Map y Lambda

- El código se reduce y se obtiene el mismo resultado:

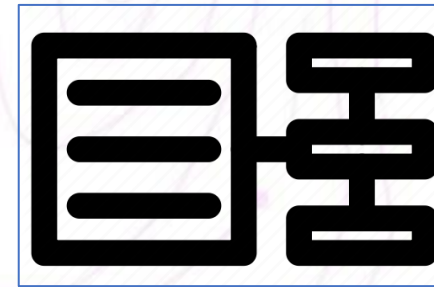
```
resultado = list( map( lambda elemento : elemento * elemento , lista) )
```





Ejemplo 2 Función Map

- Map también puede ser utilizado con funciones de más de un argumento y más de una lista.





Ejemplo 2 Función Map

```
#importamos pow.  
from math import pow
```

```
#como vemos la función pow toma dos argumentos, un número y su potencia.  
pow(2, 3)
```

```
8.0
```

```
#si tenemos las siguientes listas  
numeros = [2, 3, 4]  
potencias = [3, 2, 4]
```

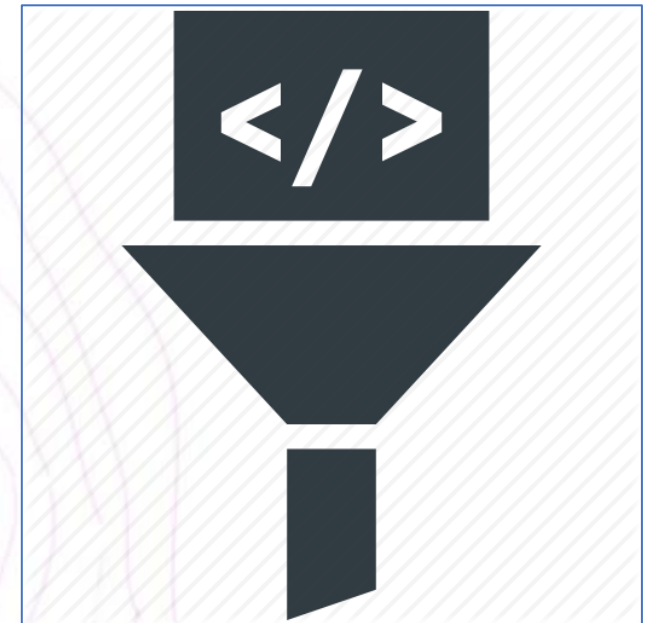
```
#podemos aplicar map con pow y las dos listas.  
#nos devolvera una sola lista con las potencias aplicadas sobre los números.  
potenciados = map(pow, numeros, potencias)  
potenciados
```

```
[8.0, 9.0, 256.0]
```



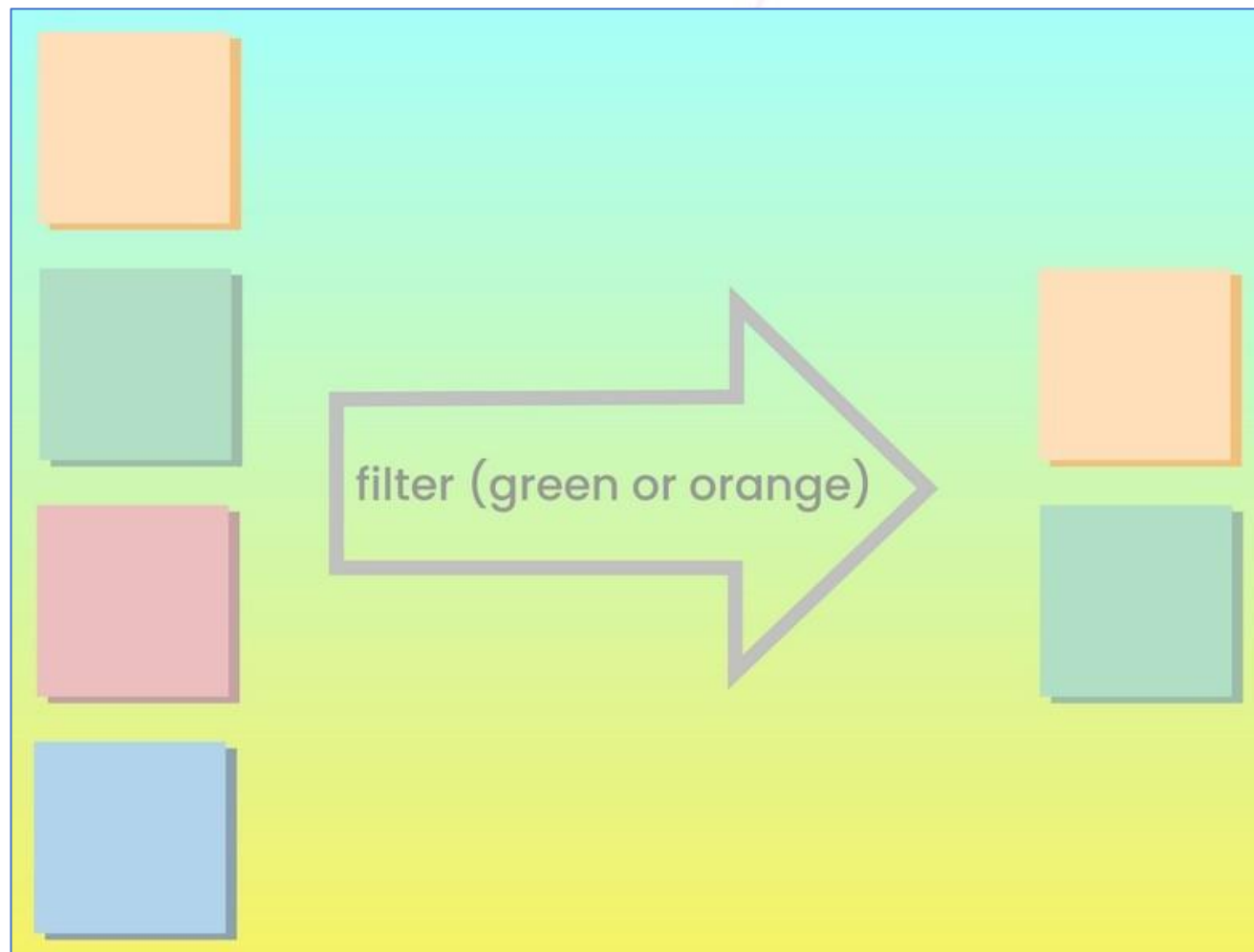
Función Filter

- La función **filter**, es quizás, una de las funciones más utilizadas al momento de trabajar con colecciones.
- Como su nombre lo indica, esta función nos permite realizar un filtro sobre los elementos de la colección.





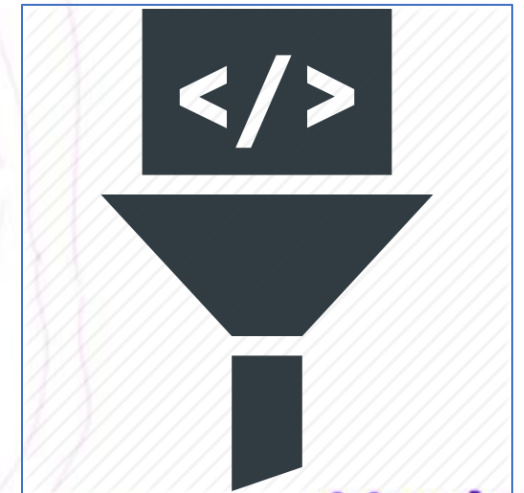
Función Filter





Función Filter

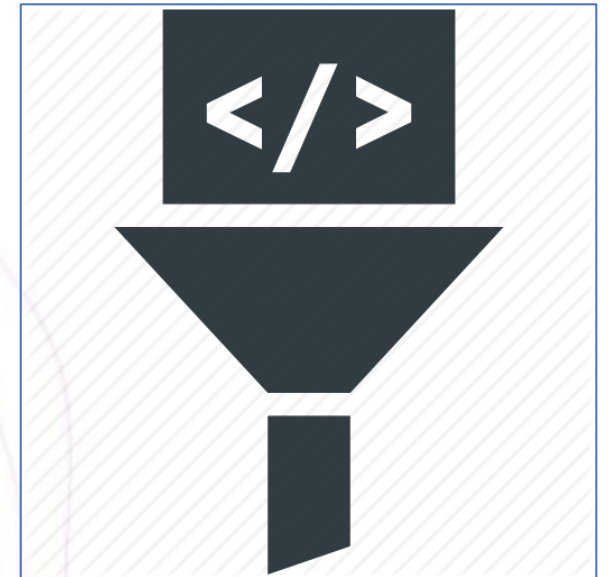
```
filter(función a aplicar, objeto iterable)
```





Función Filter

- ✓ La *función a aplicar* será aplicada a cada uno de los elementos de la colección.
- ✓ Esta función siempre deberá retornar un valor booleano.
- ✓ Todos aquellos elementos que tengan como resultado *True* después de aplicar dicha función, serán los elementos que pasen el filtro.
- ✓ A partir de estos elementos se creará una nueva colección.





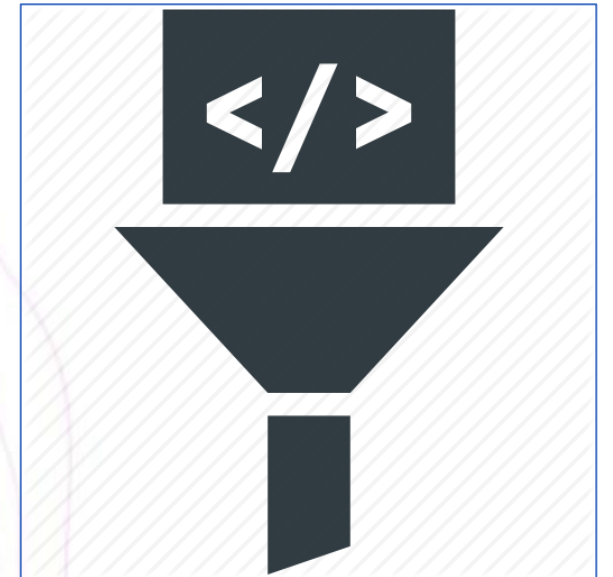
Ejemplo Función Filter

```
1 #Obtener la cantidad de elementos mayores a 5 en la tupla.  
2  
3 def mayor_a_cinco(elemento):  
4     return elemento > 5  
5  
6 tupla = (5,2,6,7,8,10,77,55,2,1,30,4,2,3)  
7 resultado = tuple(filter( mayor_a_cinco, tupla))  
8 resultado = len(resultado)  
9 print(resultado)
```



Ejemplo Función Filter

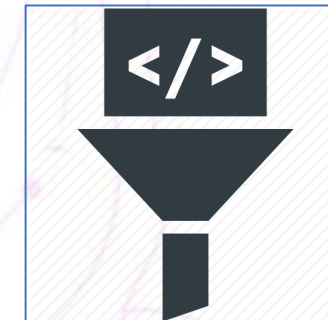
- A Partir de la versión 3 de Python, la función **filter** retorna un objeto **filter object**. Objeto que fácilmente podemos convertir a una tupla.
- De igual forma, si nuestra función a aplicar realiza una tarea sencilla, podemos reemplazarla por una función lambda.





Ejemplo Función Filter Lambda

```
1 resultado = tuple(filter( lambda elemento: elemento > 5, tupla))
```





Ejemplo 2 Función Filter Lambda

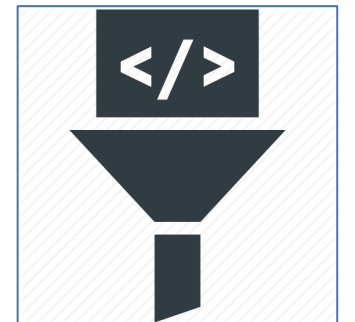
```
#Numeros pares de la lista items.  
#Forma imperativa.  
pares = []  
for i in items:  
    if i % 2 == 0:  
        pares.append(i)
```

pares

[2, 4, 6, 8, 10]

```
#Pares utilizando Filter  
#Forma funcional.  
pares = filter(lambda x: x % 2 == 0, items)  
pares
```

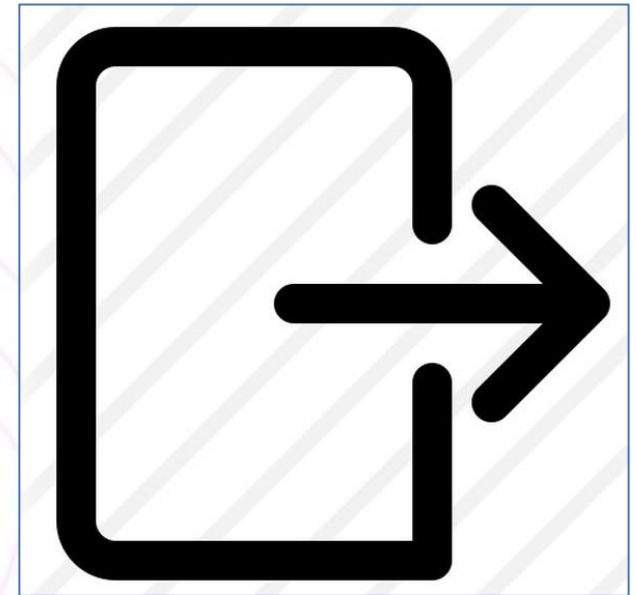
[2, 4, 6, 8, 10]





Función Reduce

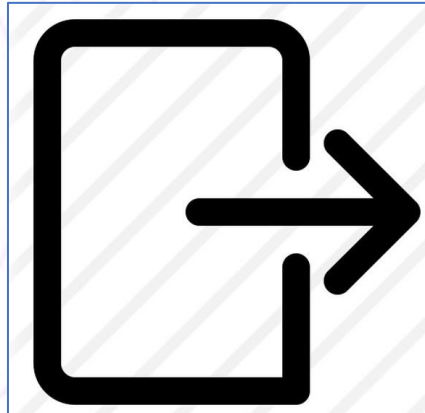
- Usaremos la función **reduce** cuando poseamos una colección de elementos y necesitemos generar un único resultado.
- **reduce** nos permitirá reducir los elementos de la colección.
- Podemos ver a esta función como un *acumulador*.





Función Reduce

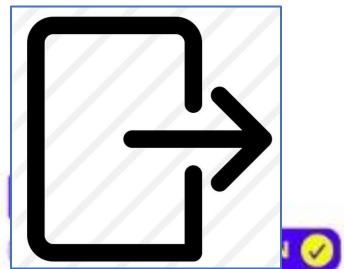
reduce(función a aplicar, objeto iterable)





Función Reduce

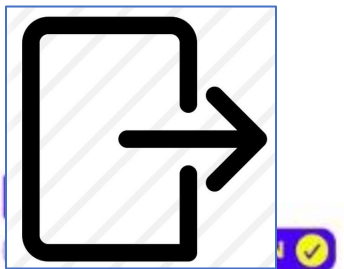
- Aquí lo importante es detallar la *función a aplicar*. Esta función debe poseer, obligatoriamente, dos parámetros.
- El primer parámetro hará referencia al *acumulador*, un variable que irá modificando su valor por cada uno de los elementos en la colección.
- Por otro lado, el segundo parámetro hará referencia a cada elemento de la colección. La función debe retornar un nuevo valor, será este nuevo valor el que será asignado al acumulador.





Ejemplo Acumulador Función Reduce

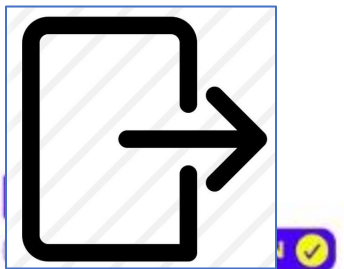
```
1 #Obtener la suma de todos los elementos en la lista
2
3 lista = [1,2,3,4]
4 acumulador = 0;
5
6 for elemento in lista:
7     acumulador += elemento
8
9 print(acumulador)
```





Ejemplo Función Reduce

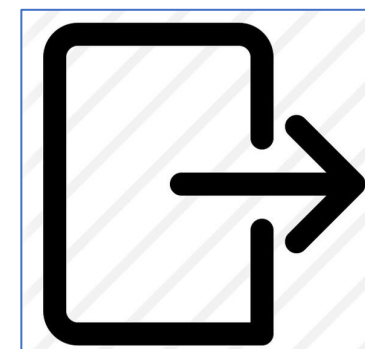
```
1 from functools import reduce
2
3 lista = [1,2,3,4]
4
5 def funcion_acumulador(acumulador=0, elemento=0):
6     return acumulador + elemento
7
8 resultado = reduce(funcion_acumulador, lista)
9 print(resultado)
```





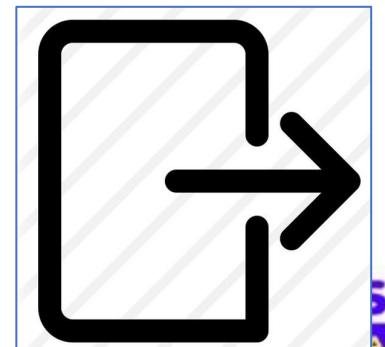
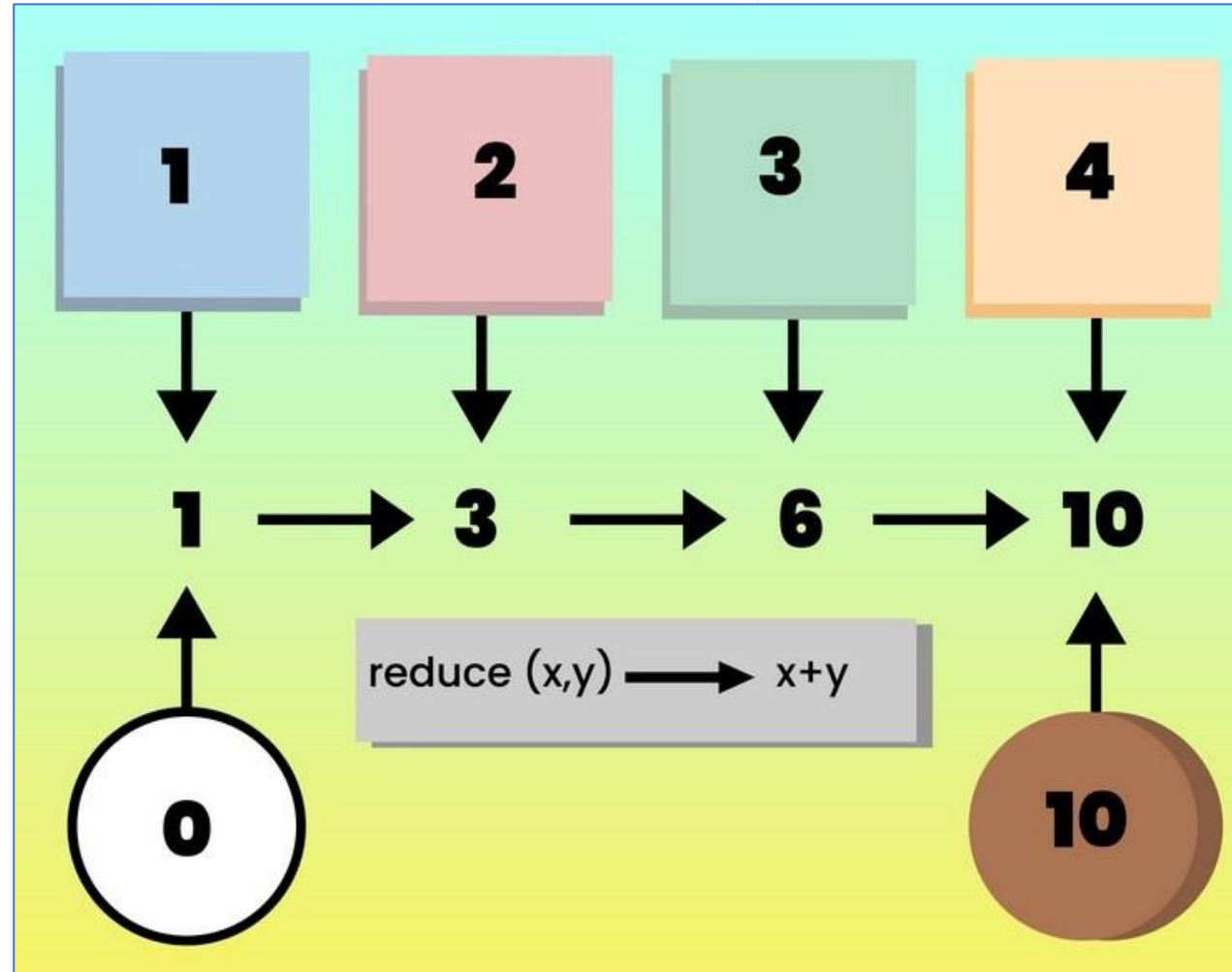
Ejemplo Función Reduce

- Por cada elemento de la colección se ejecuta la función, *funcion_acumulador*.
- La función retorna la suma de los parámetros, este valor es almacenado en nuestro acumulador.
- Al finalizar la iteración de todos los elementos, **reduce** retornará el valor del acumulador.





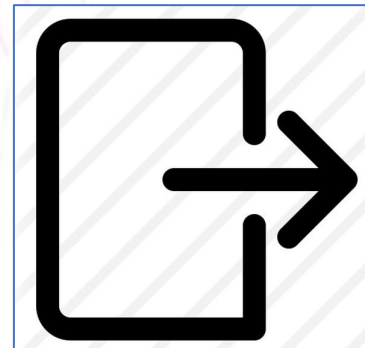
Función Reduce





Ejemplo Función Reduce Lambda

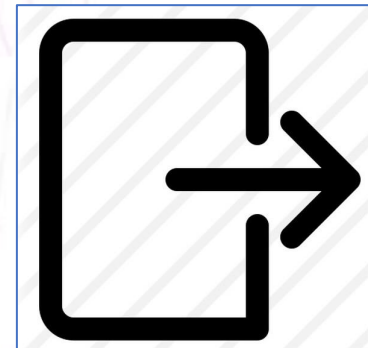
```
1 resultado = reduce(  
2     lambda acumulador=0, elemento=0: acumulador + elemento  
3     , lista  
4 )
```





Ejemplo Función Reduce Lambda

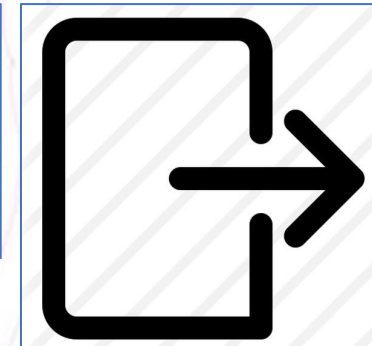
- En este caso el resultado será de tipo entero, ya que así lo he especificado al momento de asignar un valor default al acumulador.
- Sin embargo, no estamos limitados únicamente a trabajar con valores de tipo entero.





Ejemplo 2 Función Reduce Lambda

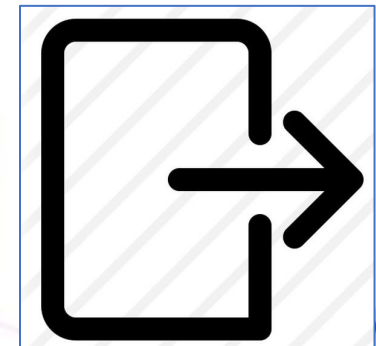
```
1 #Concatenar todos los elementos de la lista
2
3 from functools import reduce
4
5 lista = ['Python', 'Java', 'Ruby', 'Elixir']
6 resultado = reduce(lambda acumulador='', elemento='': acumulador + " - " + elemento,
7                     lista
8                     )
9
10 print(resultado)
```





Ejemplo 3 Función Reduce Lambda

- La función Reduce también cuenta con un tercer argumento que es el valor inicial o default.
- Por ejemplo si quisiéramos sumarle 10 a la suma de los elementos de la lista items, solo tendríamos que agregar el tercer argumento.





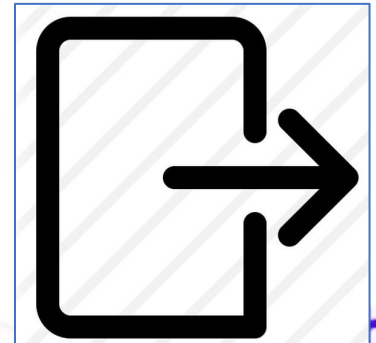
Ejemplo 3 Función Reduce Lambda

```
#10 + suma items
```

```
suma10 = reduce(lambda x, y: x + y, items, 10)
```

```
suma10
```

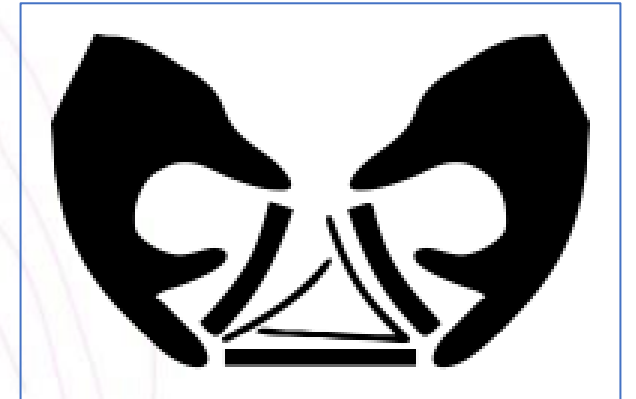
```
65
```





Función Zip

- Zip es una función para reorganizar listas.
- Como parámetros admite un conjunto de listas.
- Lo que hace es tomar el elemento i -ésimo de cada lista y unirlos en una **tupla**, después une todas las **tuplas** en una sola lista.





Ejemplo Función Zip

```
#Ejemplo de zip
```

```
nombres = ["Raul", "Pedro", "Sofia"]
```

```
apellidos = ["Lopez Briega", "Perez", "Gonzalez"]
```

```
#zip une cada nombre con su apellido en una lista de tuplas.
```

```
nombreApellido = zip(nombres, apellidos)
```

```
nombreApellido
```

```
[('Raul', 'Lopez Briega'), ('Pedro', 'Perez'), ('Sofia', 'Gonzalez')]
```