

$$F(x)$$

Cohesion: Descomponer tareas en funciones

Coupling: Cómo deben comunicarse tus funciones

lambda

```
lambda arguments: expression
```

```
1. def your_function():  
2.     pass
```

nombre de la función
la expresión que crea la función

```
1. lambda argumento: lo que afecta al argumento
```

la expresión que crea la función

Dado que lambda es una expresión en vez de una declaración, puede aparecer en lugares donde def no

```
1. a = lambda argumento: lo que afecta al argumento
```

la expresión que crea la función
nombre de la función

Toma n argumentos

```
1. lambda arg_1, arg_2: arg_1 + arg_2
```

Accede al entorno local donde haya sido expresada

```
1. def treatment(name, gender):  
2.     response = lambda name, gender: f"Sr. {name} if gender == 'm' else  
   f'Sra. {name}'"  
3.     return response(name, gender)
```

Ej lambda en un diccionario

```
1. matematica = {"cuadrado": lambda a: a**2}  
2. matematica["cuadrado"](3) # 9
```

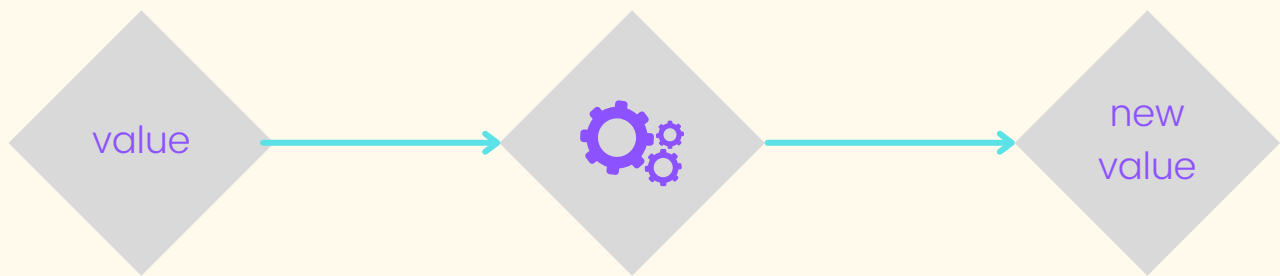
functional programming built-in tools

map

```
map(callback, iterable)
```

Por cada valor de un iterable se le aplicará una función devolviendo cada resultado

for value in iterable:



pre func

```
1.values = [1,2,3]
2.def pow(a):
3.     return a**2
4.list(map(pow, values))
```

func in situ

```
1.values = [1,2,3]
2.list(map(lambda a: a**2, values))
```

map, toma n iterables

```
1.values1 = [1,2,3]
2.values2 = [2,2,2]
3.list(map(lambda a, b: a**b, values1, values2))
```

filter

```
1.values = [-2,-1,1,2,3]
2.list(filter(lambda x: x>0, values)) # [1,2,3]
```

reduce

Devuelve un único valor aplicando una función que sea común para cada valor

El primer parametro de la callback será el primer valor del iterable y el segundo parametro será el siguiente valor del iterable

```
1.from functools import reduce
2.values = [1,2,3]
3.reduce(lambda a, b: a+b, values) # 6
```

Ternary

```
resultado_1 if condition else resultado_2
```

old way

```
1.if condition:
2.    Do something
3.else:
4.    Better no
```


with ternary operator

```
1.valor = some value
2.result = True if condition else False
```

```
1.age = 10
2.adult = True if age >= 21 else False
```

```
1.print("positive" if number >= 0 else "negative")
```

Estructuras de control



```
1.count = 0
2.while count < 5:
3.    print(count)
4.    count += 1
```


break

Salta a la siguiente línea FUERA del bucle

```
1.a = [1,2,3,4]
2.for element in a:
3.    if a == 4:
4.        break
5.    print(element) # 1,2,3
6.print("next")
```

continue


Salta al encabezado del bucle



```
1.a = [1,2,3,4]
2.for element in a:
3.    if a == 3:
4.        print("You found the magic number!")
5.        continue
6.    print(element) # 1,2, "You found...", 4
```

pass – la nada explícita

Cada vez leído, avanza al siguiente punto y además da argumento a if vacíos (placeholder)



```
1.a = [1,2,3,4]
2.for element in a:
3.    if a%2 == 0:
4.        pass
5.    else:
6.        print(element) # 1, 3
```


loop else

Siendo el bucle la primer condición de un if, loop else atrapará el caso opuesto

```
1.def loop_else():  
2.    count = 0  
3.    while count < 5:  
4.        if count == 6:  
5.            print("Found")  
6.            break  
7.    else:  
8.        print("Not Found")
```

next

```
next(iterator, [default])
```

el método next, está asociado a iteradores

```
1.example = iter([1,2,3])  
2.next(example) # 1  
3.next(example) # 2
```

yield

Método que nos permite generar iterators de un iterable (?)

Iterable: objeto que soporta la llamada iter

iterator: objeto devuelto por un iterable que soporta next

```
1.order_nums = [1,2,3,4]
2.def pow(given_list):
3.     result = []
4.     for num in given_list:
5.         result.append(num**2)
6.     return result
```

```
1.order_nums = [1,2,3,4]
2.def pow(given_list):
3.     for num in given_list:
4.         yield num ** 2
5.b = pow(order_nums) # <generator object pow yield at 0x7777657>
6.for num in b:
7.     print(num) # 1, 4, 9, 16
```

@decorators

Objetos que pueden ser nuevamente llamados y que procesan otros objetos de mismas características

```
1.order_nums = [1,2,3,4]
2.def pow(given_list):
3.     for num in given_list:
4.         yield num ** 2
5.b = pow(order_nums) # <generator object pow yield at 0x7777657>
6.for num in b:
7.     print(num) # 1, 4, 9, 16
```