

AYUDANTÍA 5

FUNCIONAL

¿Qué es una función?

- ▶ Como todo en Python, es un objeto “llamable” (*callable*).
- ▶ Programación funcional: composición de funciones

```
1 def funcion(x, y, *args, **kwargs):  
2     ... # Ejecución  
3     ... return x + y  
4  
5 funcion = lambda x, y, *args, **kwargs: x + y  
6
```

Built-in's de Python

- ▶ Incluidos en la librería estándar
- ▶ Comportamiento específico según tipo de argumento
- ▶ enumerate
- ▶ **filter**
- ▶ iter
- ▶ len
- ▶ **map**
- ▶ next
- ▶ **reduce**
- ▶ repr
- ▶ reversed
- ▶ sorted
- ▶ str
- ▶ sum
- ▶ zip

Built-in's de Python

```
1 class MiLista(list):~
2     ~
3     ... def __getitem__(self, i):~
4     ...     ... return super().__getitem__(i - 1)~
5     ~
6     ... def __repr__(self):~
7     ...     ... return "{0}".format(", ".join(self))~
8     ~
```

Iterables

- ▶ Objeto sobre el que se puede iterar
- ▶ Tiene definido el método `__iter__`

```
1  for elemento in iterable:~  
2      ...print(elemento)~  
3  ~
```

Iteradores

- ▶ Objetos que iteran sobre un iterable
- ▶ Retornados por `__iter__`
- ▶ Suelen ser iterables y su propio iterador a la vez

```
1 iterable = [9, 1, 3, 6]-  
2 iterador = iter(iterable)-  
3 next(iterador) # 9-  
4 next(iterador) # 1-  
5 next(iterador) # 3-  
6 next(iterador) # 6-  
7 next(iterador) # Error: StopIteration-  
8 |
```

Generadores

- ▶ Iterador sobre una estructura de datos
- ▶ No almacena los datos
- ▶ Usa menos memoria
- ▶ Retornado por una función generadora

```
1 with open("archivo.txt") as archivo:-  
2     ...generador = (linea for linea in archivo)-  
3     -
```

- ▶ Se pueden generar listas, diccionarios y sets por compresión

Función generadora

- ▶ Retorna un generador
- ▶ En vez de return, yield

```
1 def fibonacci():  
2     ... prev, curr = 0, 1  
3     ... while True:  
4         ... yield curr  
5         ... prev, curr = curr, prev + curr  
6  
7 generador_fib = fibonacci()  
8 print(next(generator_fib)) # 1  
9
```


Función generadora

```
1 def promedio_movil():~
2     ... total_acumulado = float(yield)~
3     ... cantidad_numeros = 1~
4     ... while True:~
5         ... nuevo = yield total_acumulado / cantidad_numeros~
6         ... cantidad_numeros += 1~
7         ... total_acumulado += nuevo~
8     ~
9     generador = promedio_movil()~
10    next(generador) # ¿Por qué?~
11    ~
12    for i in range(10):~
13        ... generador.send(i)~
```

a generator
expression

a generator

an iterator

is

is

always is

next()

*lazily produce
next value*

always is

iter()

typically is

produces

a generator
function

(an) iterable

a container

{list, set, dict}
comprehension

Fuente: <http://nvie.com/posts/iterators-vs-generators/>

map(funcion, iterable, ...)

- ▶ Aplica una función sobre los elementos de un iterable
- ▶ Retorna un iterador

```
2  gen = map(lambda x: x*3, [1, 2, 3, 4])
3
4  print(next(gen))    # 3
5  print(next(gen))    # 6
6
7  lista = list(map(lambda x: x*3, [1, 2, 3, 4]))
8
9  print(lista)        # [3, 6, 9, 12]
10
```

filter(funcion, iterable)

- ▶ Solo los elementos del iterable que entregan True al aplicar la funcion
- ▶ Retorna un iterador

```
11 gen = filter(lambda x: x % 2 == 0, range(0, 6))
12
13 print(next(gen))    # 0
14 print(next(gen))    # 2
15
16 lista = list(filter(lambda x: x % 2 == 0, range(0, 6)))
17
18 print(lista)        # [0, 2, 4]
19
```

`functools.reduce(f2, iterable)`

- ▶ Retorna un valor
- ▶ Aplica una función de a pares sobre un iterable, acumulando el resultado

```
1 import functools
2
3 def mi_factorial(n):
4     gen = range(1, n + 1)
5     return functools.reduce(lambda x, y: x * y, gen)
6
7 print(mi_factorial(3)) # 6
8 print(mi_factorial(4)) # 24
9 print(mi_factorial(5)) # 120
10
```

Documentación sugerida

Functional Programming HOWTO

<https://docs.python.org/3/howto/functional.html>

Built-in Functions

<https://docs.python.org/3/library/functions.html>

Iterables vs. Iterators vs. Generators

<http://nvie.com/posts/iterators-vs-generators/>