AYUDANTÍA 5 FUNCIONAL

¿Qué es una función?

- Como todo en Python, es un objeto "Ilamable" (callable).
- Programación funcional: composición de funciones

```
def funcion(x, y, *args, **kwargs):-

''' # Ejecución-
'''

return x + y-

funcion = lambda x, y, *args, **kwargs: x + y-
```

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

```
class MiLista(list):-

def __getitem__(self, i):-

return super().__getitem__(i - 1)-

def __repr__(self):-

return "{{0}}".format(", ".join(self))-
```

Iterables

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

Iteradores

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

```
iterable = [9, 1, 3, 6]-
iterador = iter(iterable)-
next(iterador) # 9-
next(iterador) # 1-
next(iterador) # 3-
next(iterador) # 6-
next(iterador) # Error: StopIteration-
```

Generadores

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

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

 Se pueden generar listas, diccionarios y sets por compresión

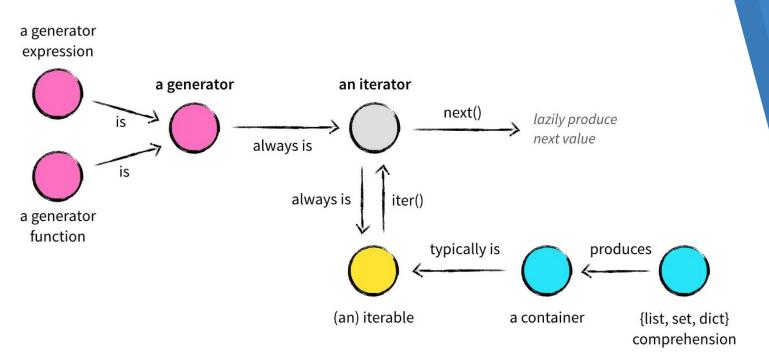
Función generadora

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

```
def fibonacci():-
   prev, curr = 0, 1
   while True:
   vield curr
   prev, curr = curr, prev + curr
6
  generador_fib = fibonacci()-
  print(next(generador_fib)) # 1-
```

Función generadora

```
def promedio_movil():-
   total_acumulado = float(yield)
3
    cantidad numeros = 1
   while True:
   nuevo = yield total_acumulado / cantidad_numeros
    ----cantidad_numeros += 1
    total acumulado += nuevo
8
9
   generador = promedio_movil()-
   next(generador) # ¿Por qué?-
10
11
   for i in range(10):-
    generador.send(i)
13
```



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

map(funcion, iterable, ...)

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

```
gen = map(lambda x: x*3, [1, 2, 3, 4])

print(next(gen)) # 3
print(next(gen)) # 6

lista = list(map(lambda x: x*3, [1, 2, 3, 4]))

print(lista) # [3, 6, 9, 12]
```

filter(funcion, iterable)

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

```
gen = filter(lambda x: x % 2 == 0, range(0, 6))

print(next(gen)) # 0
print(next(gen)) # 2

lista = list(filter(lambda x: x % 2 == 0, range(0, 6)))

print(lista) # [0, 2, 4]
```

functools.reduce(f2, iterable)

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

```
import functools

def mi_factorial(n):
    gen = range(1, n + 1)
    return functools.reduce(lambda x, y: x * y, gen)

print(mi_factorial(3)) # 6
print(mi_factorial(4)) # 24
print(mi_factorial(5)) # 120
```

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/