Functional

Listas por compresión

La fórmula general

```
lista = [ ]

for <elemento> in <iterable>:

    lista.append(<retorno_iteración>)
```

```
lista = [<retorno_iteración> for <elemento> in <iterable>]
```

Doble `for`*: ¿qué retorna?*

```
[ (i, j) for i in range(2)  for j in range(2) ]
```

Diccionario por compresión

La fórmula general

```
dic = dict()

for <elemento> in <iterable>:

    dic[elemento.nombre] = elemento
```

```
dic = {elemento.nombre: elemento for <elemento> in <iterable>}
```

enumerate y zip

Enumerate

```
lista1 = [1, 2, 3, 4, 5]
lista = list()
c = 0
for j in lista1:
    lista.append((c, j))
    c += 1
```

```
list(enumerate(lista1))
```

Zip

```
lista1 = ['Bastian', 'Flo', 'Freddie', 'Anders']

lista2 = ['Coordinador', 'Jefe', 'Mentor', 'TPD']

lista = list()

for j in range(min(len(lista1), len(lista2))):

    lista.append((lista1[j], lista2[j]))
```

```
list(zip(lista1, lista2))
```

lambda functions

Map

```
def funcion(valor):
    return valor**2
lista = [funcion(i) for i in iterable]
```

```
lista = list(map(funcion, iterable))
```

Lambda

```
def funcion(valor):

    return valor**2

lista = list(map(funcion, iterable))
```

```
lista = list(map(lambda x: x**2, iterable))
```

Lambda

- Función “on the fly”: no fue definida
- Permite recibir múltiples iterables:

```
lista = list(map(lambda x, y, z: x + y + z, iterable1, iterable2, iterable3))
```

- Permite trabajar condiciones:

```
lista = list(map(lambda x, y, z: x + y + z if x + y + z > 10 else 0, iterable1, iterable2, iterable3))
```

Actividad: Convertir a funcional

```
lista1 = [1, 2, 3, 4, 5]

lista2 = [1, 2, 3, 4]

lista3 = [1, 2, 3]

largo = min(len(lista1), len(lista2), len(lista3))

lista = []

for i in range(largo):

    lista.append(lista1[i] + lista2[i] + lista3[i])
```

Filter

```
lista1 = [1, 2, 3, 4, 5]

lista = list()

for j in lista1:

    if j%2 == 0:

        lista.append(j)
```

```
list(filter(lambda x: x%2 == 0, lista1))
```

Reduce

```
lista1 = [1, 2, 3, 4, 5]

total = 1

for i in lista1:

    total *= i
```

```
from functools import reduce

reduce(lambda x, y: x * y, lista1)
```

Actividad

Calcular el promedio de notas de un curso.

Calcular el promedio de notas de un curso de todos los alumnos que tienen promedio arriba de 5.

Built - In Functional

__lt__

```
class Numero:
    def __init__(self, a):
        self.a = a

    def __lt__(self, otra_instancia):
        return self.a < otra_instancia.a

n1, n2, n3 = Numero(1), Numero(2), Numero(3)
print(n1 < n2) # true
lista_numeros = [n3, n2, n1]
print(sorted(lista_numeros)) # [n1, n2, n3]
```

__getitem__

```
class Palabra:
     def __init__(self, palabra):
          self.palabra = palabra
     def __getitem__(self, index):
          return self.palabra[index]


p = Palabra(‘hola’)
print( c[1] ) # ‘o’
print( c[:2] ) # ‘hol’
for letra in p: print(letra, end=” “) # ‘h o l a’
```

Actividad: Indexar un número

```
num = Numero(9087)

print(num[0]) # 9

print(num[:2]) # 908

for n in num: print(n)

#9

#0

#8

#7
```

Iterable

v/s

Iterador

Iterador vs. Iterable

Iterable:

- ➔ Cualquier objeto sobre el cual se puede iterar
- ➔ Aparece al lado derecho de un for loop
- ➔ Contiene el método iter() o getitem()

Iterador:

- ➔ Es un objeto que se acuerda en qué punto va de la iteración
- ➔ Es el objeto retornado por el método iter(). (método __iter__ retorna self)
- ➔ Contiene el método next(), que nos retorna próximo elemento de iteración

Iterable

```
lista = [1, 2, 3 , 4, 5]
```

- Se puede indexar (tiene el método __getitem__)
- No tiene estado

Convertimos a Iterador

```
iterador = iter(lista)
```

- No se puede indexar
- Tiene un estado (punto dentro de la iteración)
- Se puede llamar método next(iterador)

Generadores

Nos permiten iterar sobre secuencias de datos sin la necesidad de almacenarlos en alguna estructura de datos.

Ejemplos

```
generador = ( b for b in range(10) )



for b in range(10):

     print( next(generador) )



print( next(b) ) # lanza error
```

Ejemplos

```
file = open(“archivo.txt”)

for line in file:

     print( line )



# para un archivo, hacer file.readline() es equivalente

# a usar next() en un generador
```

Funciones Generadoras

Yield

```
def generador_id():
     id = 0
     while True:
          yield id
          id += 1
# para hacer funcionar un generador...

generador = generador_id()

id = next(generador)
```

Yield: recibe parámetros y retorna

```
def sumatoria():
    total = 0
    while True:
        nuevo_numero = yield total
        total += nuevo_numero


suma = sumatoria()
next(suma) # para avanzar al primer yield
print(suma.send(10)) # 10
print(suma.send(10)) # 20
```

Actividad: crecimiento poblacional

Hacer un generador que dado una tasa de crecimiento y población inicial vaya retornando el tamaño de la población anual.

A partir del generador anterior, hacer uno que calcule el tamaño de la población dado una cantidad de muertes anuales.

__iter__ y __next__

```
class Fib:
    def __init__(self):
        self.prev = 0
        self.actual = 1

    def __iter__(self):
        return self

    def __next__(self):
        valor = self.actual
        self.actual += self.prev
        self.prev = valor
        return valor

f = Fib()
print( next(f) ) # 1
```