1 Iteradores

1.1 Definición

Cuando un proceso se debe repetir muchas veces conviene usar Iteradores.

Por ejemplo si hago:

for x in algo: Lo que se repite va acá

la variable **algo** es un iterable mientras **x** es un iterador.

Por ejemplo si tengo una lista:

Puedo iterar sobre la lista.

[1]:
```
lista = [1,2,3]
for elemento in lista:
    print(elemento)
```

```
1
2
3
```

Un secuencia es un iterable que tenga un orden, por lo cual puede ser una lista, una tupla, un set, un diccionario, un string o directamente bytes.

Un iterable es un objeto que puede recordar cual fue el último valor que entregó una vez que la iteración fue activada.

Por ejemplo, hago un loop sobre la variables que en este caso son una tupla:

[2]:
```
for elemento in ('azul','rojo', 'verde', 'amarillo'):
    print(elemento)
```

```
azul
rojo
verde
amarillo
```

O sobre las letras de una palabra:

[3]:
```
for letra in 'Computación':
    print(letra)
```

```
C
o
m
p
u
t
a
c
i
```

ó
n

Inluso podemos iterar sobre Bytes.

[4]:
```
for bytes in b'Binario':
    print(bytes)
```

```
66
105
110
97
114
105
111
```

Pregunta para el lector ¿Qué son esos números?

Pero no puedo iterar sobre los díigitos de un número entero, pero si lo convierto en un string si puedo.

[5]:
```
I = 1093647
I_texto = str(I)

for letra in I_texto:
    print(int(letra))

# otra forma de hacerlo mucho más compacta sería
print('')
digits = [int(d) for d in str(I)]
for digito in digits:
    print(digito)
```

```
1
0
9
3
6
4
7

1
0
9
3
6
4
7
```

Los *set´*s también son iterables, pero recordemos que son elementos no tienen orden, ni están numera-

dos. Veamos el ejemplo anterior pero ahora con estructura de set:

[6]:
```
for elemento in {'azul','rojo', 'verde', 'amarillo'}:
    print(elemento)
```

```
azul
verde
rojo
amarillo
```

1.2 Desde el punto de vista del Objeto

iter dispara el proceso pero los items subsiguientes hay que pedirlos con el comando **next** hasta que los elementos se acaban y da error.

[7]:
```
lista = ['gato','perro','tortuga','canario']

i_pepe=iter(lista)   # <- activo la iteracción
print(next(i_pepe))
print(next(i_pepe))
print(next(i_pepe))
print(next(i_pepe))
```

```
gato
perro
tortuga
canario
```

También es posible crear una estructura de control usando **try**/**except** como vimos antes. Pero necesito usar en este caso las órdenes **iter**/**item** para generar el conjuto de los elementos en los cuales se itera.

Ejemplo:

[8]:
```
nums = [1,2,3,4]
i_nums = iter(nums)

while True:
    try:
        item = next(i_nums)
        print(item)
    except StopIteration:
        break
```

```
1
2
3
4
```

En este caso la activación del error al llamar a un valor cuando ya se acabaron activa el **except** y obliga a terminar la iteración.

2 Generadores

2.1 Definición

Pero como hacemos en el caso de que la iteración deba hacerse con datos que tengo en un archivo el cual por su tamaño no es posibile cargarlo en la memoria de la computadora? O en el caso de que se busque una opción posible dentro de infinitas posibilidades ¿cómo se maneja esta situación en Python?

Para ello se usan los generadores ya que un método for-loop no serviría.

Un generador es una función que actúa como los iteradores y "genera" los elementos para ser utilizados en el loop. Es decir que trabaja con una iteración donde los elementos de la iteración se solicitan uno a uno. Y de esta manera no consume tantos recursos de memoria. Con otras palabras, el generador me va dando de a uno los valores en la medida de que se los vaya pidiendo.

¿Cómo funcionan estos generadores? Como indicamos son funciones y van devolviendo los valores de la iteración en demanda. Veamos un ejemplo abstracto:

[9]:
```
def f():
    yield 1
    yield 2
    yield 3
```

Note que ahora la función no termina su ejecución con un "return", sino con "yield" y si la ejecuto:

[10]:
```
f()
```

[10]:
```
<generator object f at 0x7fa6a029fba0>
```

Me dice que la funcion f() es un generador, así que puedo iterar con ella:

[11]:
```
for x in f():
    print(x)
```

```
1
2
3
```

Veamos un ejemplo no tan abstracto. Calculemos los cuadrados de una lista, pero de uno a uno.

[12]:
```
# defino el generador
def cuadrados(numeros):
    for i in numeros:
        yield (i*i)

# armo la lista de números que voy a procesar
lista=[1,2,3,4]
```

```
# construyo la función que genera los números
# esta orden construye el generador, pero no la ejecuta!!!
mi_lista = cuadrados(lista)

print(mi_lista)
print("")

# y ahora la recorro
for numero in mi_lista:
    print(numero)
```

```
<generator object cuadrados at 0x7fa6a029fdd0>

1
4
9
16
```

[13]:
```
def numeros_primos():
    yield 2 # devuelvo 2 como el primer primo
    primo_cache = [2]

    for n in range(3,1000000):

        es_primo=True

        for p in primo_cache:
            if n%p == 0:
                es_primo = False
                break
        if es_primo:
            primo_cache.append(n)
            yield n

for p in numeros_primos():
    print(p)
    if p> 10:
        break
```

```
2
3
5
7
11
```

2.2 Expresión Generadora o "Generator Expression"

Son una manera más corta de construir generadores. Son un pariente de los List Comprehension y en cierta medida tiene notación de tupla, pero no tiene nada que ver con estas últimas.

Se escriben entre parentésis (las List comprehension entre corchetes) con una gramática similar.

Veamos una:

[14]:
```
def natural_numbers(i=1):
#  i = 0
  while True:
    yield i
    i += 1

#for num in natural_numbers():
#  print(num)
```

[15]:
```
cubos = (x**3 for x in natural_numbers())

for x in cubos:
    print(x)

# Si lo dejo así no terminará nunca
# Lo detengo a las 10 valores

    if( x> 1000):
           cubos.close()

# close() es un método asociado al generador para deternerlo
```

```
1
8
27
64
125
216
343
512
729
1000
1331
```

Utilizando las mismas reglas de un List Comprehension podria ponerle un final en su propia construcción asi:

[16]:
```
cubos = (x**3 for x in range(5))
```

```
for x in cubos:
    print(x)
```

```
0
1
8
27
64
```