Informàtica
Curs 2023-2024

Iteradors – part 1 de 2

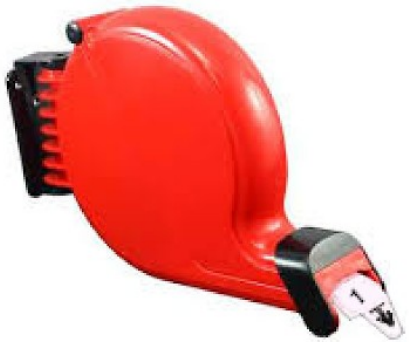
D.Tost

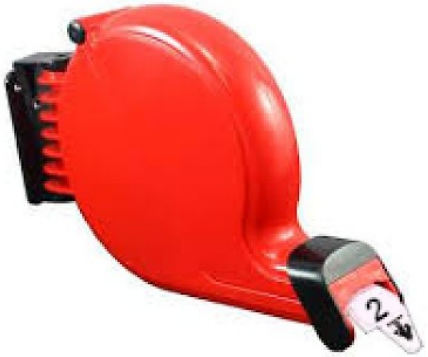
Iteradors

Sovint en informàtica, volem utilitzar mecanismes que ens permeten iterar sobre una seqüència d'objectes.

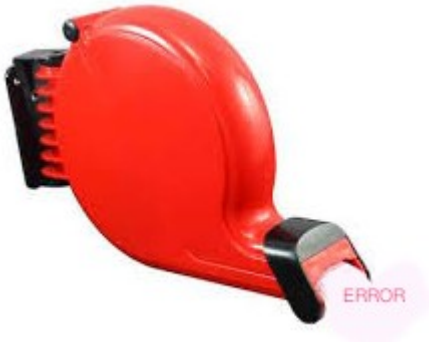
Aquests mecanismes s'anomenen **iteradors.**

La manera d'utilitzar-los és:
- quan es necessita un element, demanar a l'iterador que ens doni l'element en curs (funció **`next`**)
- si hi ha un element en curs
  - → l'iterador el dóna
  - → el nou element en curs passa a ser a ser el següent
- si no queden elements, l'iterador dóna error. (stop iteration)





.....



Iteradors - La funció `next`

Iterador: objecte que proporciona un mecanisme per recorre una seqüència d'elements. Suporta la funció *next()* que permet accedir a l'element següent de la seqüència i dóna una excepció de *StopIteration* quan no queden més elements per iterar.

Exemple: *it és un objecte iterador sobre la seqüència 34, 56, 12*

```
>>> next(it)
34
>>> next(it)
56
>>> next(it)
12
>>> next(it)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
>>> next(it)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

Ja ha iterat sobre tots els elements
A partir d'ara sempre donarà
un StopIteration error si fem next(it)

Composició `for` amb iteradors

Els iteradors es poden recorre amb la composició:

`for` *`element`* **`in iterador`**

`element` pren per valors successius els elements de la seqüència de `iterador`. La composició s'aturarà amb el darrer element, sense donar error.

Exemple: *it és un objecte iterador sobre la seqüència 34, 56, 12*

```
>>> for e1 in it:
...     print(e1)
...
34
56
12

>>> for e2 in it:
...     print(e2)

>>> next(it)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

El darrer valor de `e1` és 12. El `for` s'acaba.

Ja s'ha iterat sobre tots els elements de `it`. No s'entra dins del `for`. Si `e2` no s'havia inicialitzat prèviament, és una variable no definida.

Ja s'ha iterat sobre tots els elements de `it`. La funció `next` dóna error.

Contenidors i iterables

Objecte contenidor: objecte que fa referència a d'altres objectes, per exemple tuples, llistes, diccionaris, strings. Està format per diversos “elements”.

Objecte iterable: objecte que suporta la funció **`iter`** que retorna un iterador

Tots els objectes contenidors que coneixem fins ara són iterables. Proveu:

```
>>> l = [3, 4, 7]
>>> it = iter(l)
>>> next(it)
3
>>> next(it)
4
>>> next(it)
7
>>> next(it)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

Iterables - La funció `iter`

iter(llista): crea un iterador dels elements de la llista (idem tuple)

```
>>> it = iter([23, -7, 26, 90, 3])
>>> it
<list_iterator object at 0x7fcbc7b2afa0>
>>> list(it)
[23, -7, 26, 90, 3]
```

iter(string): crea un iterador dels caràcters del string

```
>>> it = iter('abcd')
>>> it
<str_iterator object at 0x7fcbc849e040>
>>> list(it)
['a', 'b', 'c', 'd']
```

iter(dict): crea un iterador de les claus del diccionari

```
>>> it = iter({'hola': 'hello', 'adeu':'bye',
'gràcies':'thank you'})
>>> it
<dict_keyiterator object at 0x7fcbc7b09950>
>>> list(it)
['hola', 'adeu', 'gràcies']
```

Composició `for` amb iterables

Els **iterables** (que no són iteradors) es poden recorre amb la composició:

`for` element **`in`** **iterable**

Per implementar la composició, Python utilitza l'iterador retornat per `iter(iterable)` i executa la funció `next` d'aquest iterador a cada iteració. La composició anterior és equivalent a :

`for` element **`in iter(`**iterable**`)`**

Exemple: *m és un objecte iterable*

```
>>> for e in m:
...     print(e)
...
34
56
12
>>> for e in m:
...     print(e)
...
34
56
12
```

A cada vegada que es fa un for es crea un nou iterador

Iteradors - La funció `iter`

La funció **iter** aplicada a un iterador retorna aquest mateix iterador. Si `it` és un iterador llavors es cumpleix `it==iter(it)`.

```
>>> llista = [23, -7, 26, 90, 3]
>>> it = iter(llista)
>>> iter(it) == it
>>> it == llista
False
>>> list(it) == llista
True
```

La llista no és el mateix que l'iterador de la llista

List d'una llista retorna una còpia (no un àlias) de la llista

`iter(iterador)`
Retorna l'iterador

```
>>> it = iter(llista)
>>> it2 = iter(it)
>>> it2 == it
True
>>> list(it)
[23, -7, 26, 90, 3]
>>> list(it2)
[]
```

¡MOLT IMPORTANT!
En llistar `it`, l'esgotem
i per tant també esgotem
`it2` ja que són el mateix

Funcions generadores

Les funcions generadores són funcions especials que creen iteradors.

No fa cap return sinó que utilitzen la instrucció **`yield`**. Aquesta instrucció especifica els elements a lliurar cada vegada que es cridi el mètode **`next`** sobre un iterador creat per la funció generadora. Habitualment, les funcions generadores s'estructuren com una composició iterativa amb 1 o més `yield` a cada iteració.

```
>>> a = generador_1()
>>> next(a)
'pere'
>>> next(a)
'joan'
>>> next(a)
'anna'
>>> next(a)
Traceback (most recent call last):
  File "<stdin>", line 1, in
<module>
StopIteration
```

`a` és un iterador

s'ha executat fins yield 'joan'

s'ha executat fins yield 'anna'

s'ha acabat el codi, no queden més `yield`

L'iterador està «esgotatt»

```
def generadora_1():
    yield  'pere'
    yield' joan'
    yield 'anna'
```

Funcions generadores

Les funcions generadores són funcions especials que creen iteradors.

No fa cap return sinó que utilitzen la instrucció **`yield`**. Aquesta instrucció especifica els elements a lliurar cada vegada que es cridi el mètode **`next`** sobre un iterador creat per la funció generadora. Habitualment, les funcions generadores s'estructuren com una composició iterativa amb 1 o més `yield` a cada iteració.

```
>>> a = generador_1()
>>> for e in a:
...       print(e)
...
pere
joan
Anna
```

S'aplica la composició for a cada iteració s'executa el codi fins el següent yield

```
def generadora_1():
    yield  'pere'
    Yield  'joan'
    yield 'anna'
```

```
>>> next(a)
Traceback (most recent call last):
  File "<stdin>", line 1, in
<module>
StopIteration
```

`a` està esgotat

`a` està esgotat

```
>>> for e in a:
...     print(e)
...
```

Funcions generadores

A cada `next`, s'executa tot el codi fins al següent `yield` inclós.

```
def gen(n):
     yield ‘a’
     for i in range (3):
        n = n+ 1
     yield n
     if n > 3:
        yield ‘End’
```

```
>>> it = gen(3)
>>> next(it)
'a'
>>> next(it)
6
>>> next(it)
'End'
>>> next(it)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

Primer next

```
def gen(n):
 ini  yield ‘a’    fi
     for i in range (3):
        n = n+ 1
     yield n
     if n > 3:
        yield ‘End’
```

Segon next

```
def gen(n):
     yield ‘a’
 ini for i in range (3):
        n = n+ 1
     yield n    fi
     if n > 3:
        yield ‘End’
```

Tercer next

```
def gen(n):
     yield ‘a’
     for i in range (3):
        n = n+ 1
     yield n
ini  if n > 3:
        yield ‘End’ fi
```

Funcions generadores

Exemple: Un iterador dels 100 primers múltiples de 7

```
>>> def Mult7_v2():
...     y = 7
...     for i in range(100) :
...       yield y
...       y = y + 7
...
>>> it = Mult7_v2()
>>> next(it)
7
>>> next(it)
14
>>> for i in range(4):
...    next(it)
...
21
28
35
42
>>> next(it)
49
>>> for i in it:
...    next(it)
...
56
[...]
700
```

El primer

El segon

Quatre més

Un més

La resta

Funcions generadores

Generalitzem l'exemple dels múltiples de 7

Un iterador dels n primers múltiples de m

```
>>> def MultM(m, n):
...     y = m
...     for i in range(n) :
...         yield y
...         y = y + m
...
```

Un iterador infinit dels múltiples de m

```
>>> def MultMInfinit(m):
...     y = m
...     while True:
...         yield y
...         y = y + m
...
```

Proveu :

```
>>> next(MultM(9, 6))
9
>>> next(MultM(9, 6))
9
```

Enteneu perquè?

Proveu:

```
>>> it = MultM(9, 6)
>>> next(it)
9
>>> next(it)
18
```

Quín valor espereu per next(it) ara?

Funcions generadores

Exemple: una funció generadora que crea un **iterador infinit** dels múltiples de 7

```
>>> def Mult7_v4():
...     y = 7
...     while True :
...       yield y
...       y = y + 7
...
>>> it = Mult7_v4()
>>> for i in range(40):
...    next(it)
...
7
...
280
>>> next(it)
287
>>> for i in it:
...    next(it)
...
294
301
....
....
```

El 40 primers

Un més

¿La resta ?

NOOOO és infinit!!! Ctrl-C per parar!!!

Funcions generadores

Dissenyeu una funció generadora que permeti de recorre els dígits d'un enter d'un en un d'esquerra a dreta.

```
def genera_xifres(n) :
    for x in str(n) :
        yield int(x)
```

Cada vegada que es crida genera_xifres es crea un nou iterador.

```
>>> for x in genera_xifres(7423) :
            print x
7
4
2
3

>>> sum(genera_xifres(7423))
16

>>> max(genera_xifres(7423))
7

>>> min(genera_xifres(7423))
2
```

Definicions

Objecte contenidor: objecte que fa referència a d'altres objectes, per exemple tuples, llistes, diccionaris, strings. Està format per diversos “elements”.

Objecte iterable: objecte contenidor que disposa d’un mecanisme per lliurar els seus elements d'un en un (e.g. list, tuple, dict, str, set)

Iterador: Objecte capaç d’iterar sobre una seqüencia d’elements. Els iteradors són iterables però no tots els objectes iterables són iteradors (e.g. una llista)

Iterador d’un iterable: iterador construït a partir d’un iterable: un *objecte iterable* permet d’obtenir un *iterador* del seus elements amb la funció: `iter(iterable)`.

Funció generadora: funció que, utilitzant la instrucció yield, retorna un iterador

http://www.learningpython.com/2009/02/23/iterators-iterables-and-generators-oh-my/

La setmana vinent ...

Funcions estàndard i del mòdul itertools que creen iteradors