Itérateurs...

L'adjectif "*itérable*" fait référence à tout objet sur lequel on peut "boucler" ; une liste, un dictionnaire, une chaîne, un fichier, …

ces objets disposent de la méthode __iter__()

```
x = [1, 3, 7, 11]
for i in x:
    print(i)

print(dir(x))
```

```
['__add__', '__class__', '__contains__', '__delattr__', '__delitem__',
'__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__',
'__getitem__', '__gt__', '__hash__', '__iadd__', '__imul__', '__init__',
'__init_subclass__', '__iter__', '__le__', '__len__', '__lt__', '__mul__',
'__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__',
'__reversed__', '__rmul__', '__setattr__', '__setitem__', '__sizeof__',
'__str__', '__subclasshook__', 'append', 'clear', 'copy', 'count', 'extend',
'index', 'insert', 'pop', 'remove', 'reverse', 'sort']
```

Itérateurs...

Un *itérateur* est un objet ayant la propriété de

1) se souvenir de l'état courant (dans lequel il se trouve)
2) __next__() renvoie l'état courant et passe à l'état "suivant"

Remarque: une liste est itérable mais n'est pas *a priori* un itérateur, elle ne possède pas la méthode __next__()

```
lis = [1, 3, 7, 11]

x = iter(lis)  # on en fait un itérateur

print(next(x))
print(next(x))
print(next(x))

print(dir(x))
```

```
1
3
7
['__class__', '__delattr__', '__dir__', '__doc
'__format__', '__ge__', '__getattribute__', '_
'__hash__', '__init__', '__init_subclass__', '
'__le__', '__length_hint__', '__lt__', '__ne__
'__next__', '__reduce__', '__reduce_ex__', '__
'__setattr__', '__setstate__', '__sizeof__', '
'__subclasshook__']
```

Itérateurs...

Si on arrive “à la fin” de l’*itérateur*, dans son “dernier” état, alors Python génère une exception “*StopIteration*” quand on appellera __next__:

```
lis = [1, 3, 7, 11]
x = iter(lis)

print(next(x))
print(next(x))
print(next(x))
print(next(x))
print(next(x))
```

```
In [7]: runcell(0, '/home/mrigo/cours/python/fichiers2020/
untitled0.py')
1
3
7
11
Traceback (most recent call last):

  File "/home/mrigo/cours/python/fichiers2020/untitled0.py",
line 10, in <module>
    print(next(x))

StopIteration
```

Itérateurs...

On peut définir ses propres itérateurs :

```
class MonIterateur:

    def __init__(self,debut,fin,increment):
        self.etat = debut # on définit l'état courant
        self.fin = fin  # on retient pour plus tard
        self.inc = increment

    def __next__(self):
        if self.etat<self.fin:
            temp = self.etat
            self.etat +=self.inc
            return(temp)
        else:
            raise StopIteration

x = MonIterateur(2,7,2)

print(next(x))
print(next(x))
print(next(x))
print(next(x))
```

```
2
4
6
Traceback (most recent ca

  File "/home/mrigo/cours
line 23, in <module>
    print(next(x))

  File "/home/mrigo/cours
line 16, in __next__
    raise StopIteration

StopIteration
```

Itérateurs...

Intérêts :

Avec un itérateur, on peut passer en revue/travailler sur chaque élément contenu

Il s'agit en général d'un procédé plus efficace qu'une boucle

Dans l'exemple précédent, MonIterateur(2,100000,2) a l'avantage de ne pas stocker de "grande liste" en mémoire, l'itérateur n'utilise jamais que l'état courant – on stocke un entier vs une liste d'un million d'éléments.

Python dispose de nombreux itérateurs prédéfinis...

Itertools

```python
from itertools import count

un_iterateur = count(10)

print(next(un_iterateur))
print(next(un_iterateur))
print(next(un_iterateur))
print(next(un_iterateur))
```

```
10
11
12
13
```

Avec *count*, on a ici un itérateur “infini”
(on retient juste la valeur actuelle à utiliser)

Iterator	Arguments	Results	Example
`count()`	start, [step]	start, start+step, start+2*step, …	`count(10) --> 10 11 12 13 14 ...`
`cycle()`	p	p0, p1, … plast, p0, p1, …	`cycle('ABCD') --> A B C D A B C D ...`
`repeat()`	elem [,n]	elem, elem, elem, … endlessly or up to n times	`repeat(10, 3) --> 10 10 10`

Itertools

```python
from itertools import product

x='abc'
y=[1,2]
# produit cartésien
un_iterateur = product(x,y)

print(list(un_iterateur))
```

On construit un nouvel itérateur pour passer en revue les éléments d'un produit cartésien.

On peut utiliser des listes, des chaînes, des ensembles, … des objets itérables

Cela revient ici à 2 boucles *for* imbriquées.

```
[('a', 1), ('a', 2), ('b', 1), ('b', 2), ('c', 1), ('c', 2)]
```

Itertools

```python
from itertools import product

x=[1,3,5]
y=[1,2]
# produit cartésien
un_iterateur = product(x,y)

for i in un_iterateur:
    print(i[0],i[1],i[0]**2-i[1]**2)
```

```
1 1 0
1 2 -3
3 1 8
3 2 5
5 1 24
5 2 21
```

```python
from itertools import product

x="abc"
y="de"
z="fg"
# produit cartésien
un_iterateur = product(x,y,z)

print(list(un_iterateur))
```

```
[('a', 'd', 'f'), ('a', 'd', 'g'), ('a', 'e', 'f'), ('a',
'e', 'g'), ('b', 'd', 'f'), ('b', 'd', 'g'), ('b', 'e',
'f'), ('b', 'e', 'g'), ('c', 'd', 'f'), ('c', 'd', 'g'),
('c', 'e', 'f'), ('c', 'e', 'g')]
```

Itertools

Combinatoric iterators:

Iterator	Arguments	Results
`product()`	p, q, … [repeat=1]	cartesian product, equivalent to a nested for-loop
`permutations()`	p[, r]	r-length tuples, all possible orderings, no repeated elements
`combinations()`	p, r	r-length tuples, in sorted order, no repeated elements
`combinations_with_replacement()`	p, r	r-length tuples, in sorted order, with repeated elements

Examples	Results
`product('ABCD', repeat=2)`	`AA AB AC AD BA BB BC BD CA CB CC CD DA DB DC DD`
`permutations('ABCD', 2)`	`AB AC AD BA BC BD CA CB CD DA DB DC`
`combinations('ABCD', 2)`	`AB AC AD BC BD CD`
`combinations_with_replacement('ABCD', 2)`	`AA AB AC AD BB BC BD CC CD DD`

Itertools

Un exemple rudimentaire

```python
from itertools import product

mots = product("ab", repeat=4)

for m in mots:
    vide=""
    chaine=vide.join(m)
    if chaine[0:2]==chaine[2:]:
        print(chaine,'est un carré')
    else:
        print(chaine)
```

```
aaaa est un carré
aaab
aaba
aabb
abaa
abab est un carré
abba
abbb
baaa
baab
baba est un carré
babb
bbaa
bbab
bbba
bbbb est un carré
```

Itertools

Retour sur *count()* et autres

Infinite iterators:

Iterator	Arguments	Results	Example
`count()`	start, [step]	start, start+step, start+2*step, …	`count(10) --> 10 11 12 13 14 ...`
`cycle()`	p	p0, p1, … plast, p0, p1, …	`cycle('ABCD') --> A B C D A B C D ...`
`repeat()`	elem [,n]	elem, elem, elem, … endlessly or up to n times	`repeat(10, 3) --> 10 10 10`

Itertools

itertools.**accumulate**(*iterable*[, *func*, *, *initial=None*])

Make an iterator that returns accumulated sums, or accumulated results of other binary functions (specified via the optional *func* argument).

```python
from itertools import accumulate

x = [1,4,5,7,8]

for i in accumulate(x):
    print(i)

def f(x,y):
    return (2*x+y)

# ajout d'une fonction comme argument
for i in accumulate(x, f):
    print(i)

# 1, f(1,4)=6, f(6,5)=17, f(17,7)=41, f(41,8)=90
```

```
1
5
10
17
25
1
6
17
41
90
```

Itertools

itertools.**chain**(**iterables*)

Make an iterator that returns elements from the first iterable until it is exhausted, then proceeds to the next iterable, until all of the iterables are exhausted. Used for treating consecutive sequences as a single sequence. Roughly equivalent to:

```
from itertools import chain

x = [1,7,8]
y = "abc"

# construire un itérateur à partir
# de plusieurs itérables

print(list(chain(x,y,x)))
```

```
[1, 7, 8, 'a', 'b', 'c', 1, 7, 8]
```

Pour ses propres classes

On peut souhaiter qu'une classe supporte l'itération...

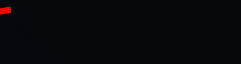
```
class Mammifere:

    def __init__(self,parole,ans):
        self.cri = parole
        self.age = ans
        self.aptitudes = []

    def __iter__(self):
        return iter(self.aptitudes)

mon_animal = Mammifere("miaou", 5)
mon_animal.aptitudes.append("manger")
mon_animal.aptitudes.append("dormir")
mon_animal.aptitudes.append("griffer")

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



```
manger
dormir
griffer
```

Fonction anonyme

```python
print((lambda x : x**2)(3))

f = lambda x : x+3

print(f(5))

print((lambda x,y : x**y)(3,5))
```

```
9
8
243
```

```python
from itertools import accumulate

x = [1,2,5,7]

for i in accumulate(x, lambda x,y: 2*x+y):
    print(i)
```

map()

```
def f(x):
    return(x**2)

l=list(map(f,range(10)))

print(l)

l=list(map(lambda x : x**2, range(10)))

print(l)
```

La fonction *map()* s'applique à un objet itérable : on applique la fonction à chaque élément de l'itérable.

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

map() & filter()

```
l=list(map(lambda x: x.capitalize(), ['cat', 'dog', 'cow']))
print(l)


def test(x):
    return(x%2==0)

l=list(filter(test, range(11)))
print(l)

l=list(filter(lambda x : x%2==0, range(11)))
print(l)
```

```
['Cat', 'Dog', 'Cow']
[0, 2, 4, 6, 8, 10]
[0, 2, 4, 6, 8, 10]
```

reduce()

```python
from functools import reduce

l = [1, 3, 8]
print( reduce(lambda x,y : 2*x+3*y , l) )

# f(1,3)=2*1+3*3=14, f(14,8)=2*14+3*8=46

def test(x,y):
    if x>y:
        return(x)
    else:
        return(y)

l = [1, 3, 9, 4, 8, 17]
print( reduce(test , l) ) # renvoie 17
```

Getting Started With Python's `reduce()`

Python's **`reduce()`** implements a mathematical technique commonly known as **folding** or **reduction**. You're doing a fold or reduction when you reduce a list of items to a single cumulative value. Python's `reduce()` operates on any iterable—not just lists—and performs the following steps:

1. **Apply** a function (or callable) to the first two items in an iterable and generate a partial result.
2. **Use** that partial result, together with the third item in the iterable, to generate another partial result.
3. **Repeat** the process until the iterable is exhausted and then return a single cumulative value.