

Compréhensions de listes

Exemple 1 :

```
>>> chaine = 'ABC'
>>> modele = [ ord( elem ) for elem in chaine ]
>>> chaine
'ABC'
>>> modele
[ 65, 66, 67 ]
```

- expression : `ord(s)`

Compréhensions de listes

Exemple 2 :

```
>>> symboles = '$&€&f'
>>> lascii = [ ord( s ) for s in symboles if ord( s ) > 127 ]
>>> lascii
[8364, 163]
```

- expression : `ord(s)`


- condition : `ord(s) > 127`

Compréhensions de listes

Exemple 3 : produit cartésien

```
>>> couleurs = [ 'Noir', 'Blanc' ]
>>> tailles= [ 'S', 'M' ]
>>> tshirts = [ ( c, t ) for c in couleurs
                for t in tailles ]
>>> tshirts
[ ( 'Noir', 'S' ), ( 'Noir', 'M' ), ( 'Blanc', 'S' ), ( 'Blanc', 'M' ) ]
```

```
>>> for c in couleurs
    for t in tailles
        print ( c, t )
```



```
>>> tshirts
( 'Noir', 'S' )
( 'Noir', 'M' )
( 'Blanc', 'S' )
( 'Blanc', 'M' )
```

Compréhensions de listes

Listcomps Versus map() et filter()

Les listcomps font tout ce que les fonctions `map` et `filter` font, sans les contorsions de l'expression `lambda`

```
>>> symboles = '$&€&f'
>>> lascii = [ord( s ) for s in symboles if ord( s ) > 127]
>>> lascii
[8364, 163]

>>> lascii = list( filter( lambda s: s > 127, map( ord, symbols )))
>>> lascii
[8364, 163]
```

Compréhensions de listes

La fonctionnelle map

Une fonction générique `map` peut cependant offrir une alternative aux compréhensions pour exprimer de façon concise des transformations de listes.

Signature de `map (fct, L)` :

$$\begin{array}{ccccc} \underline{(\text{alpha} \rightarrow \text{beta})} & * & \underline{\text{list}[\text{alpha}]} & \rightarrow & \underline{\text{list}[\text{beta}]} \\ \downarrow & & \downarrow & & \downarrow \\ \text{fct}(\text{alpha}) \rightarrow \text{beta} & & \text{liste L} & & \text{liste résultat RL} \end{array}$$

Compréhensions de listes

Exemple :

```
>>> def map ( fct, L ) :  
    """ Retourne la transformation de L par la fonction fct """  
    maListe = []  
    for k in L :  
        maListe.append( fct( k ) )  
    return maListe  
  
>>> map ( carre, [ 1, 2, 3, 4, 5 ] )  
[ 1, 4, 9, 16, 25 ]
```

Compréhensions de listes

De même que **les listes en compréhension** sont un moyen de créer des listes,

- les **sets en compréhension** sont un moyen de créer des sets,
- les **dictionnaires en compréhension** sont un moyen de créer des dictionnaires,

facilement et rapidement.

Si les listcomps construisent des listes, utiliser les **expressions génératrices ou genexp** pour les autres types de séquences.

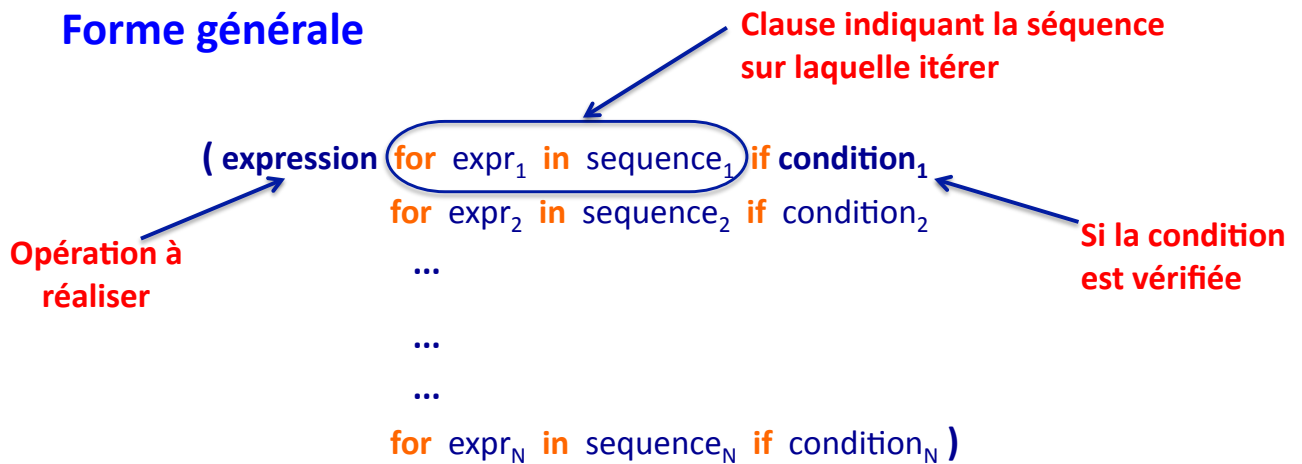
Expressions Génératrices

Une **expression génératrice (genexp)** construit des types de séquences (tuples, tableaux, ...) en **filtrant** et **transformant** les éléments d'une séquences grâce à un itérateur.

- Une **genexp** renvoie un générateur (fonction) qui **produira les éléments à la demande**.
- Une **genexp** utilise la même syntaxe qu'une **listcomp**, mais les éléments produits sont entre parenthèses.

Expressions Génératrices

Forme générale



NB : La clause `if` est facultative

Expressions Génératrices

Exemple 1 : Initialiser un tuple/array à partir d'une genexp

```
>>> symboles = '$&€&f'
>>> ( ord( s ) for s in symboles )
<generator object <genexp> at 0x...>
>>> tuple( ord( s ) for s in symboles )
(36, 38, 163, 38, 8364)

>>> import array
>>> array.array( 'l', ( ord( s ) for s in symboles ) )
array( 'l', [36, 38, 163, 38, 8364])
```

Expressions Génératrices

Exemple 2 : produit cartésien dans une genexp

```
>>> couleurs = [ 'Noir', 'Blanc' ]
>>> tailles= [ 'S', 'M' ]
>>> for tshirts in ('%s %s' % ( c, t ) for c in couleurs
                    for t in tailles ) :
    print(tshirt)
Noir S
Noir M
Blanc S
Blanc M
```

NB : aucune liste n'est jamais produite par la genexp

Expressions Génératrices

Expression génératrice Vs compréhension de liste

- La compréhension de liste renvoie une liste Python, contrairement à l'itérateur.
- ➔ Toujours préférer les expressions génératrices si on travaille sur :
 - des itérateurs infinis,
 - des itérateurs produisant une très grande quantité de données.

Expressions Génératrices

Exemple :

```
>>> def genere_AB( ) :  
    print( 'start' )  
    yield 'A'  
    print( 'continue' )  
    yield 'B'  
    print( 'end.' )
```

```
| >>> res1 = [ x*3 for x in genere_AB ( ) ]
```

```
start  
continue  
end
```

```
| >> res1  
['AAA', 'BBB']
```

```
>>> res2 = ( x*3 for x in genere_AB ( ) )
```

```
>>> res2  
<generator object <genexp> at 0x...>
```

Expressions Génératrices

../..

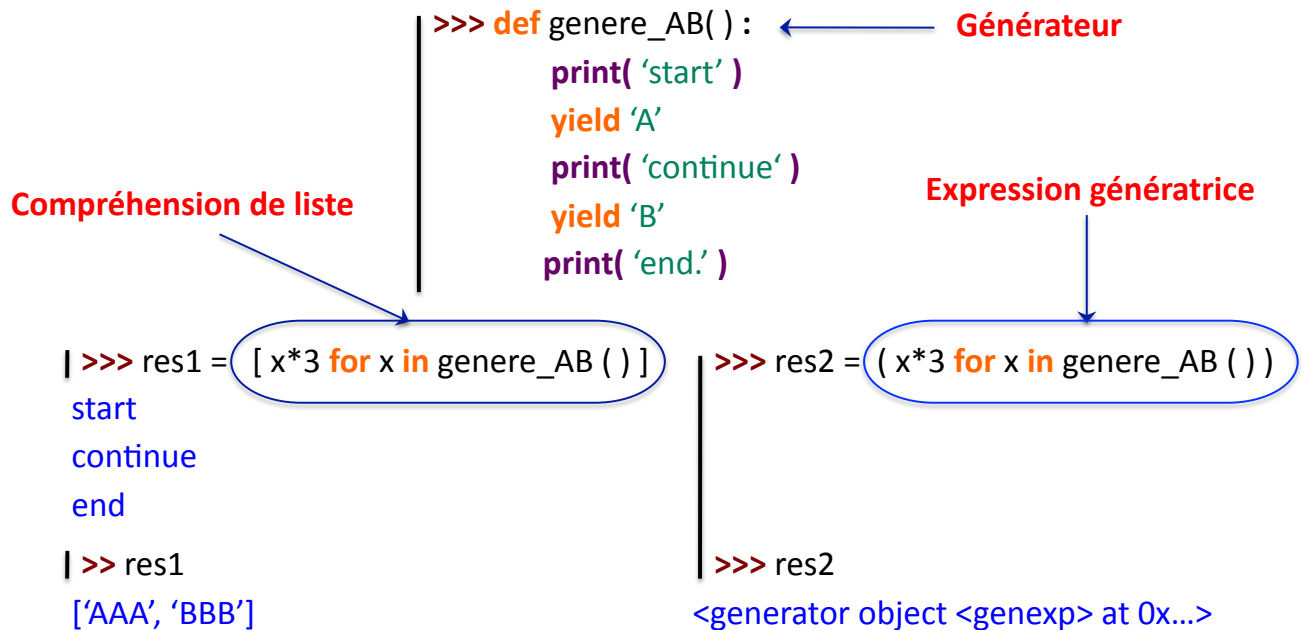
```
| >> res1  
['AAA', 'BBB']
```

```
| >>> res2  
<generator object <genexp> at 0x...>
```

```
| >>> for elem in res2 :  
    print( elem )
```

```
start  
AAA  
continue  
BBB  
end
```

Expressions Génératrices



Expressions Génératrices

Générateur Vs Expression génératrice

- Une **expression génératrice** est un raccourci syntaxique pour créer un générateur sans définir et appeler une fonction
- Les **générateurs** sont **beaucoup plus flexibles** car ils permettent de coder des logiques complexes

➔ préférer les *expressions génératrices* dans les cas les plus simples et faciles à lire

Expressions Génératrices

Expression génératrice Versus `map()` et `filter()`

- Deux fonctions natives qui **clonent les propriétés des expressions génératrices**
- Le même comportement peut être obtenu à l'aide d'une compréhension de liste

Expressions Génératrices

Exemple 1 : application de la fonction `map()`

```
>>> def upper( s ) :  
    return s.upper( )  
  
>>> list( map( upper, [ 'sentence', 'fragment' ] ) )  
['SENTENCE', 'FRAGMENT']
```

Utilisation d'une compréhension de liste

```
>>> [ upper( s ) for s in [ 'sentence', 'fragment' ] ]  
['SENTENCE', 'FRAGMENT']
```

Expressions Génératrices

Exemple 2 : application de la fonction `filter()`

```
>>> def is_even( x ) :  
        return ( x % 2 ) == 0  
  
>>> list( filter( is_even, range( 10 ) ) )  
[ 0, 2, 4, 6, 8 ]
```

Utilisation d'une compréhension de liste

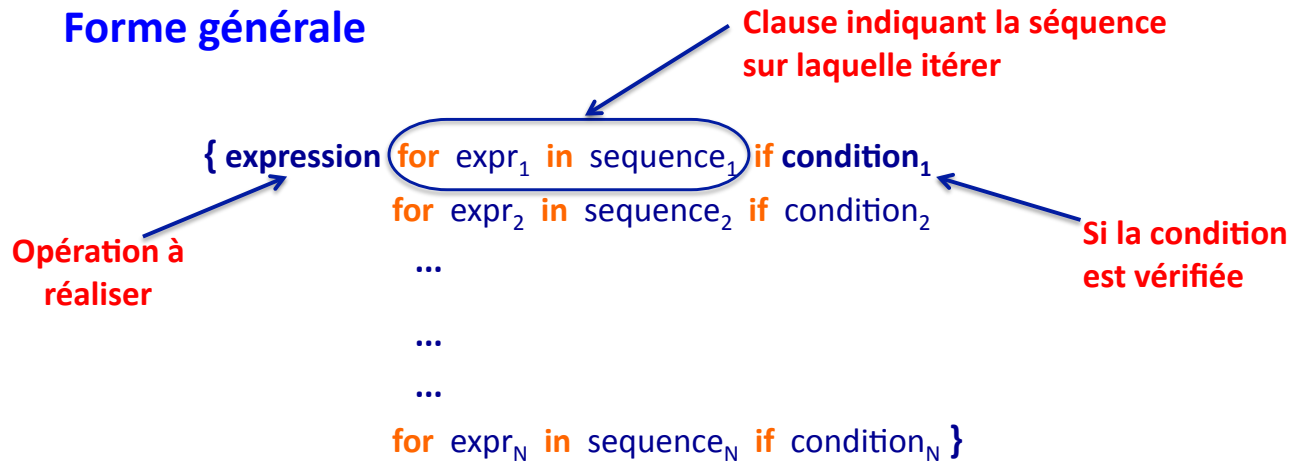
```
>>> [filter( x ) for x in range( 10 ) ]  
[ 0, 2, 4, 6, 8 ]
```

Compréhension d'ensembles

Les *compréhensions d'ensembles* (*setcomp*) ou *constructions d'ensembles par compréhension*, permettent de construire des ensembles complexes à partir d'itérables :

- autres ensembles,
- séquences (intervalles d'entiers, chaînes de caractères, listes)
- ou dictionnaires (itérables sur les clés ou les associations).

Compréhensions d'ensembles



NB : La clause `if` est facultative

Compréhension d'ensembles

Exemple 1(a) : A partir d'une liste

```
>>> def elements ( L ) :  
    "Retourne l'ensemble des éléments apparaissant  
    dans la liste L"  
    Ens = set( )           # ensemble résultat  
    for e in L :  
        Ens.add( e )  
    return Ens  
  
>>> elements ( [ 1, 3, 5, 5, 7, 9, 1, 11, 13, 13] )  
{ 1, 3, 5, 7, 9, 11, 13 }
```

Compréhension d'ensembles

Exemple 1(b) : A partir d'une liste en utilisant un setcomp

```
>>> def elements2 ( L ) :  
    "Retourne l'ensemble des éléments apparaissant  
    dans la liste L"  
    return { k for k in L }  
  
>>> elements2 ( [ 1, 3, 5, 5, 7, 9, 1, 11, 13, 13] )  
{1, 3, 5, 7, 9, 11, 13}
```

Compréhension d'ensembles

Exemple 2(a) : A partir d'une chaîne de caractères

```
>>> def elementsC ( chaine ) :  
    "Retourne l'ensemble des éléments apparaissant  
    dans la chaîne de caractères chaine"  
    Ens = set( )          # ensemble résultat  
    for e in chaine :  
        Ens.add( e )  
    return Ens  
  
>>> elementsC ( 'abracadabra' )  
{ 'b', 'c', 'd', 'r', 'a' }
```

Compréhension d'ensembles

Exemple 2(b) : A partir d'une chaîne de caractères avec un setcomp

```
>>> def elementsC2 ( chaine ) :  
    "Retourne l'ensemble des éléments apparaissant  
    dans la chaîne de caractères chaine"  
    return { k for k in chaine }  
  
>>> elementsC2 ( 'abracadabra' )  
{ 'b', 'c', 'd', 'r', 'a' }
```

Compréhension d'ensembles

Exemple 3(a) : A partir d'un intervalle

```
>>> def nombrePairs ( n ) :  
    "Retourne l'ensemble des nombres paires  
    dans l'intervalle [1;n["  
    Ens = set( )          # ensemble résultat  
    for elem in range( 1, n ) :  
        if elem %2 == 0:  
            Ens.add( elem )  
    return Ens  
  
>>> nombrePairs ( 6 )  
{ 2, 4 }
```

Compréhension d'ensembles

Exemple 3(b) : A partir d'un intervalle en utilisant un setcomp

```
>>> def nombrePairs2 ( n ) :  
    "Retourne l'ensemble des nombres paires  
    dans l'intervalle [1;n["  
    return { elem for elem in range(1, n) if elem %2==0 }  
  
>>> nombrePairs2 ( 6 )  
{ 2, 4 }
```

Compréhension d'ensembles

Exemple 4(a) : A partir d'un intervalle

```
>>> def nombreFiltres ( n ) :  
    "Retourne l'ensemble des carrés des nombres divisibles  
    par 3 dans l'intervalle [1;n["  
    Ens = set( )          # ensemble résultat  
    for elem in range( 1, n ) :  
        if elem %3 == 0:  
            Ens.add( elem**2 )  
    return Ens  
  
>>> nombreFiltres ( 16 )  
{ 225, 36, 9, 144, 81 }
```

Compréhension d'ensembles

Exemple 4(b) : A partir d'un intervalle en utilisant un setcomp

```
>>> def nombreFiltres2 ( n ) :  
    "Retourne l'ensemble des carrés des nombres  
    divisibles par 3 dans l'intervalle [1;n]"  
    return { elem**2 for elem in range(1, n) if elem %3==0 }  
  
>>> nombreFiltres ( 16 )  
{ 225, 36, 9, 144, 81 }
```

Compréhension de dictionnaires

Les *compréhensions de dictionnaires* ou *constructions de dictionnaires par compréhension (dictcomp)*, permettent de construire des dictionnaires une paire **clé : valeur** à partir de n'importe quel itérable :

- un autre dictionnaire,
- une séquence (intervalle d'entier, chaîne de caractères, liste),
- ou un ensemble.

Compréhension de dictionnaires

Compréhension simple

{ <cle> : <valeur> **for** <var> **in** <iterable> }

- **<cle>** : une expression contenant éventuellement une ou plusieurs occurrences de **<var>**
- **<valeur>** : une expression contenant éventuellement une ou plusieurs occurrences de **<var>**
- **<var>** : la variable de compréhension
- **<iterable>** : une expression retournant la structure itérée pour construire le dictionnaire.

Compréhension de dictionnaires

Exemple 1(a) : construction à partir d'une liste de tuples

```
>>> def dicoNoms ( L ) :  
    "Retourne le dictionnaire construit à partir de L"  
    BD = {}          # dictionnaire résultat  
    for couple in L :  
        BD[ couple[0] ] = couple[1]  
    return BD  
  
>>> dicoNoms( [ ('Dupont', 41), ('Dupond', 27), ('Dupons', 31) ] )  
{ 'Dupont' : 41, 'Dupond' : 27, 'Dupons' : 31 }
```


Compréhension de dictionnaires

Exemple 1(b) : construction à partir d'une liste de tuples en utilisant dictcomp

```
>>> def dicoNoms2 ( L ) :  
    "Retourne le dictionnaire construit à partir de L"  
    return { nom: age, for (nom, age) in L }  
  
>>> dicoNoms2( [ ('Dupont', 41), ('Dupond', 27), ('Dupons', 31) ] )  
{ 'Dupont' : 41, 'Dupond' : 27, 'Dupons' : 31 }
```

Compréhension de dictionnaires

Exemple 2(a) : construction à partir d'un autre dictionnaire

```
>>> DicoPersonnel = { 331: ('Dupont', 'Paul', 41),  
                     288: ('Dupond', 'Marcelle', 27),  
                     441: ('Dupons', 'Gaston', 31) }  
  
>>> def NomsAges( dico ):  
    BD = {}  
    for (cle, (nom, prenom, age)) in dico.items():  
        BD[nom] = age  
    return BD  
  
>>> NomsAges( DicoPersonnel )  
{ 'Dupont' : 41, 'Dupond' : 27, 'Dupons' : 31 }
```

Compréhension de dictionnaires

Exemple 2(b) : construction à partir d'un autre dictionnaire en utilisant les compréhensions

```
>>> DicoPersonnel = { 331: ('Dupont', 'Paul', 41),
                      288: ('Dupond', 'Marcelle', 27),
                      441: ('Dupons', 'Gaston', 31) }

>>> def NomsAges2( dico ):
    return { nom: age for (cle, (nom, prenom, age)) in dico.items() }

>>> NomsAges2( DicoPersonnel )
{'Dupont': 41, 'Dupond': 27, 'Dupons': 31 }
```

Compréhension de dictionnaires

Compréhensions différenciées clé/valeur

```
{ <cle>:<valeur> for (<varC>, <varV>) in zip(<iterableC>, <iterableV>) }
```

- **<cle>** : une expression contenant éventuellement une ou plusieurs occurrences de **<varC>**
- **<valeur>** : une expression contenant éventuellement une ou plusieurs occurrences de **<varV>**
- **<varC>** : la variable de compréhension pour les clés
- **<varV>** : la variable de compréhension pour les valeurs
- **<iterableC>** : une expression retournant la structure itérée pour les clés
- **<iterableV>** : une expression retournant la structure itérée pour les valeurs

Compréhension de dictionnaires

Exemple 1(a) : construction différenciée en utilisant les compréhensions

```
>>> { chiffre: mot for chiffre in range(0, 3)
      for mot in ['zero', 'un', 'deux'] }
{ 0: 'deux', 1: 'deux', 2: 'deux' }
```

Compréhension de dictionnaires

Exemple 1(b) : construction différenciée en utilisant zip et les compréhensions

```
>>> { chiffre: mot for (chiffre, mot) in zip(range(0, 3),
      ['zero', 'un', 'deux']) }
{ 0: 'zero', 1: 'un', 2: 'deux' }
```

Compréhension de dictionnaires

Exemple 2 : construction différenciée en utilisant zip et les compréhensions

```
>>> { chiffre: mot for (chiffre, mot) in zip(range(0, 3),  
                                           ['zero', 'un', 'deux', 'trois']) }  
  
{ 0: 'zero', 1: 'un', 2: 'deux' }
```

Compréhension de dictionnaires

Exemple 3 : construction différenciée en utilisant zip et les compréhensions

```
>>> { chiffre: mot for (chiffre, mot) in zip(range(0, 10),  
                                           ['zero', 'un', 'deux', 'trois']) }  
  
{ 0: 'zero', 1: 'un', 2: 'deux', 3: 'trois' }
```

Compréhension de dictionnaires

Compréhensions avec filtrage

{ <cle>:<valeur> **for** <var> **in** <iterable> **if** <condition> }

- **<cle>** : une expression contenant éventuellement une ou plusieurs occurrences de **<var>**
- **<valeur>** : une expression contenant éventuellement une ou plusieurs occurrences de **<var>**
- **<var>** : la variable de compréhension
- **<iterable>** : une expression retournant la structure itérée pour construire le dictionnaire
- **<condition>** : condition de compréhension

Compréhension de dictionnaires

Exemple 1 : construction en filtrant les clés

```
>>> BD = { chiffre: mot for (chiffre, mot) in zip(range(0, 10),  
                                                [ 'zero', 'un', 'deux', 'trois', 'quatre' ] ) }  
>>> BD  
{ 0: 'zero', 1: 'un', 2: 'deux', 3: 'trois', 4: 'quatre' }  
  
>>> { cle:BD[cle] for cle in BD if cle % 2 == 0 }  
{ 0: 'zero', 4: 'quatre', 2: 'deux' }
```

Compréhension de dictionnaires

Exemple 2 : construction en filtrant les associations

```
>>> BD = { chiffre: mot for (chiffre, mot) in zip(range(0, 10),  
                                                [ 'zero', 'un', 'deux', 'trois', 'quatre' ] ) }  
>>> BD  
{ 0: 'zero', 1: 'un', 2: 'deux', 3: 'trois', 4: 'quatre' }  
  
>>> { mot for (cle, mot) in BD.items() if cle % 2 == 0 }  
{ 'deux', 'quatre', 'zero' }
```

Compréhension de dictionnaires

Exemple 3 : construction en filtrant les valeurs

```
>>> DP = { 331: ('Dupont', 'Paul', 41),  
          288: ('Dupond', 'Marcelle', 27),  
          441: ('Dupons', 'Gaston', 31) }  
  
>> { nom for (num, (nom, prenom, age)) in DP.items() if age >= 30 }  
{ 'Dupont', 'Dupons' }
```