

# Cours n°5

Les listes  
(données mutables  
/non mutables)  
les “compréhensions”

# Les listes

- une liste est une autre donnée composée
- une liste est une séquence (comme les chaînes ou les tuples)

## →Exemples:

```
>>> noms_mois= ["janvier", "fevrier", "mars", "avril", "mai",  
"juin", "juillet", "aout", "septembre", "octobre", "novembre",  
"decembre"]
```

```
>>> nbjours=[31,28,31,30,31,30,31,31,30,31,30,31]
```

```
>>> entierspremiers=[2,3,5,7,11,13,17,19]
```

# Les listes

→Exemples:

```
>>> jeucarte = [["valet", "coeur"], ["valet", "pique"],  
["dame", "carreau"], ["as", "trefle"]]
```

```
>>> etud=["dupont", "pierre", [12,3,1987], "info",  
"0231675423", 12,4,15]
```

*On dit parfois 'tableaux'  
mais les tableaux sont plutôt dans numpy*

→ longueur: `len(l)` renvoie le nombre d'éléments de la liste `l`.  
Il s'agit du nombre d'éléments au "premier niveau".

`len(jeucarte) --> 4`

`len(entierspremiers) --> 8`

`len(etud)--> 8`

→ accès à un élément (comme les tuples)

`jeucarte[1] --> ["valet", "pique"]`

`entierspremiers[-1]--> 19`

→ sous-listes (comme pour les tuples)

`>>>entierspremiers[2:-1]--> [5,7,11,13,17]`



→ d'où par exemple la fonction pour convertir un entier de 1 à 12 en nom de mois:

```
def mois(i):  
    return noms_mois[i-1]    # si pas de tests
```



$L1, l2, l3 = [1, 3, 5, 7], [8, 6, 4, 2], [0]$

$l1 + l2 \rightarrow [1, 3, 5, 7, 8, 6, 4, 2]$

$l1 + l2 + l3 \rightarrow [1, 3, 5, 7, 8, 6, 4, 2, 0]$

$l3 * 6 \rightarrow [0, 0, 0, 0, 0, 0]$

$l1 + [9] \rightarrow [1, 3, 5, 7, 9]$

$l1 + 9$  provoquerait une erreur

(pb de type car  $l1$  est une liste et 9 est un entier)

il faut faire  $l1 + [9]$  pour rajouter l'élément 9 à la fin de la liste

# des fonctions sur les listes

Il existe beaucoup de fonctions sur les listes.

exemples:

`min(l)`: qui renvoie le plus petit élément de la liste `l`

`max(l)`: qui renvoie le plus grand élément de la liste `l`

Attention il faut que les éléments de la liste soient de même type

L'ordre utilisé est `<=` (ceci marche aussi pour des tuples en fait)

```
>>> l = [1,2,7,4,9,2,4,7,3]
```

```
>>> max(l)
```

```
9
```

```
>>> min(l)
```

```
1
```

```
>>> ll= ["banane","abricot","orange", "pomme", "poire","ananas"]
```

```
>>> min(ll)
```

```
'abricot'
```

```
>>> max(ll)
```

```
'pomme'
```

sum(l) fait la somme des éléments de l (marche aussi pour un tuple) si l est une liste de nombres

```
>>> l = [1,2,7,4,9,2,4,7,3]
>>> sum(l)
39
```

```
>>> sum(["banane","abricot","orange", "pomme",
"poire","ananas"])
```

Traceback (most recent call last):

```
File "<pyshell#24>", line 1, in <module>
    sum(["banane","abricot","orange", "pomme",
"poire","ananas"])
```

TypeError: unsupported operand type(s) for +: 'int' and 'str'



# Transformation d'une séquence en liste

La fonction **list** transforme une séquence en une liste:

par exemple pour une chaîne de caractères:

```
>>> list("est")  
['e', 's', 't']
```

Mais on ne peut pas créer une liste à partir d'un nombre:

```
>>> list(12345)
```

Traceback (most recent call last):

```
File "<pyshell#22>", line 1, in -toplevel-  
    list(12345)
```

TypeError: iteration over non-sequence

On peut évidemment intercaler la fonction **str**:

```
>>> list(str(12345))  
['1', '2', '3', '4', '5']
```

## Et pour des tuples:

```
>>> list((2,5))
```

```
[2, 5]
```

```
>>> list (('poire', 'pomme', 'citron'))
```

```
['poire', 'pomme', 'citron']
```

```
>>> list(((1,2),(3,5),(3,6)))
```

```
[(1, 2), (3, 5), (3, 6)]
```

La transformation ne se fait qu'au "premier niveau"

Beaucoup de programmes s'écrivent comme  
pour les tuples ou les chaînes

→ compter le nombre d'éléments égaux à un  
élément x donné dans une liste l:

```
def compte(x,l):  
    cp=0  
    for el in l:  
        if el==x:  
            cp+=1  
    return cp
```

On peut remarquer que c'est exactement le même  
programme qu'avec les chaînes!

## → extraire les entiers pairs d'une liste d'entiers

Il faut une variable pour “construire” la nouvelle liste.  
Cette variable est initialisée à [] (liste vide).

```
def extraire(l):  
    res=[] # on part d'une liste vide  
    for x in l:  
        if x%2==0: # test de parité  
            res=res+ [x] # attention aux []  
    return res
```

Là le programme est très semblable à celui des tuples mais ce n'est **pas le même** car l'initialisation de res n'est pas la même:  
() pour construire un tuple, [] pour une liste  
De même l'ajout d'un terme n'a pas la même syntaxe.

## Attention à l'utilisation de la fonction

```
>>>l=[1,2,4,7,9]  
>>>extraire(l)  
[2,4]
```

```
>>>extraire([1,2,3,8])  
[2,8]
```

```
>>>extraire(1,2,3,8)
```

**Provoque une erreur**



→ un exemple de construction d'une liste:

on veut faire une saisie de notes par l'utilisateur  
(ici sans tests)

```
def saisie():  
    l=[]  
    note= int(input("nombre suivant ou -1 pour stopper "))  
    while note!= -1:  
        l= l+[note] # toujours avec les crochets  
        note= int(input("nombre suivant ou -1 pour stopper "))  
    return l
```

pourquoi cette nouvelle structure?

Cela ressemble beaucoup aux tuples à part les [],

Donc qu'est-ce qui est différent?

La liste est une donnée **mutable**  
(possibilité de modification  
physique)

# une liste est une donnée mutable

Ce qui veut dire: modifiable de façon *interne*

l'instruction: `l[i]= val`

met la valeur `val` à la place d'index `i` de la liste `l`

La liste est **MODIFIEE** pas recopiée

```
>>> liste=[1, 6, 3, 8, 2]
```

```
>>> liste[4]=9    # ne renvoie rien
```

```
>>> liste
```

```
[1, 6, 3, 8, 9]
```

# une liste est une donnée mutable

```
>>> liste[4]=[1,1,1,1,1]
```

```
>>> liste
```

```
[1, 6, 3, 8, [1, 1, 1, 1, 1]]
```

rappel: on ne pouvait pas le faire pour une string  
ou un tuple, il fallait recopier ....

## Remarque:

`maliste[i]= val`

Ne renvoie rien (None)  
mais maliste a été modifiée



# attention aux données mutables !!!

```
>>> x=[0,1,2,3,4,5,6]
```

```
>>> y=x
```

```
>>> x[1]="*"
```

x et y sont modifiés en même temps!!!

```
>>> x
```

```
[0, '*', 2, 3, 4, 5, 6]
```

```
>>> y
```

```
[0, '*', 2, 3, 4, 5, 6]
```

```
>>> y[2]="$"
```

```
>>> y
```

```
[0, '*', '$', 3, 4, 5, 6]
```

```
>>> x
```

```
[0, '*', '$', 3, 4, 5, 6]
```

Pour les tranches:

pas de problème car une copie est faite  
automatiquement

```
>>> l=[1,2,3,4,5,6,7,8,9]
```

```
>>> k=l[4:7]
```

```
>>> k
```

```
[5, 6, 7]
```

```
>>> k[1]=4
```

```
>>> k
```

```
[5,4,7]
```

```
>>> l
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

`m=l[:]` crée une copie “indépendante” de la liste `l`

## Suppression

`del l[i]` supprime la valeur d'indice i de la liste l

```
>>> l=[1, 6, 3, 8, 2]
```

```
>>> del l[2]
```

la liste est modifiée et None est renvoyé

```
>>> l
```

[1, 6, 8, 2] # c'est la case 2 qui a 'disparue' pas la valeur 2

remarque: on ne doit pas (et on ne peut pas) écrire:

`liste=del liste[i]`

## Et les tranches?

`l[i:j]= l2` remplace la tranche entre i et j (j exclus) par la liste l2 ;  
la liste l2 peut être de taille quelconque ainsi que la "tranche"

```
>>> l=[1, 6, 3, 8, 2]
```

```
>>> l[1:3]= [0,0,0,0,0]
```

```
>>> l
```

```
[1,0,0,0,0,0,8,2]
```

```
>>> l[2:]=[7]
```

```
>>> l
```

```
[1,0,7]
```

## Suppression de tranches

`del l[i:j]` supprime la tranche comprise entre les index `i` (inclus) et `j` (exclus)

```
>>> l=[1, 2, 3, 4, 5, 6, 7]
```

```
>>> del l[1:4]
```

```
>>> l
```

```
[1, 5, 6, 7]
```

Dans tous les cas, la liste est modifiée et `None` est renvoyé



# Modifier ou créer une liste ?

dans tous les programmes, il faudra donc voir si on veut

→ *renvoyer une nouvelle liste*

→ *modifier celle existante....*

exemple: étant donnée une liste de nombres, on peut renvoyer la liste des carrés avec une nouvelle liste , ou modifier la liste de départ pour obtenir celle des carrés de ces nombres:

## Les codes

```
def carres(l): # avec création d'une nouvelle liste
```

```
    res=[]
```

```
    for x in l:
```

```
        res+=[x**2]
```

```
    return res    #indispensable
```

```
def carres_modif(l): # avec modification de la liste de départ
```

```
    for i in range(len(l)):
```

```
        l[i]= l[i]**2
```

```
    return l    # pas indispensable l a été modifiée
```

# Aléatoire et listes

→ on veut prendre un élément de la liste au hasard:

On peut toujours faire un randint en utilisant la longueur de la liste (là c'est comme les tuples et chaînes):

```
def prendre(l):  
    i= randint(0,len(l)-1)  
    return l[i]
```

Remarque: ne pas oublier d'importer les fonctions du module random

## Un plus pour les listes:

on peut même supprimer l'élément choisi dans la liste, avant de renvoyer cet élément (pour modéliser une “pioche” par exemple )

```
def prendre_retire(l):  
    i= randint(0,len(l)-1)  
    val=l[i]  
    del l[i]  
    return val
```

L'élément tiré au hasard est renvoyé et la liste est modifiée pour supprimer cet élément.

# Mélange

→ et si on veut mélanger la liste on peut prendre au hasard successivement un élément et construire ainsi une nouvelle liste

```
def melange(l):  
    res=[]  
    for i in range(len(l): # on tire autant de fois qu'il y a d'éléments  
        j= randint(0,len(l)-1)  
        res+= [l[j]]  
        del l[j] # la liste l perd un élément à chaque fois  
    return res
```



## Fonctions du module random

→ `choice(seq)`: renvoie un élément au hasard dans la séquence non vide `seq` (si `seq` est vide cela provoque une erreur) ; la séquence peut être une liste, une chaîne...

```
>>> choice("bonjour tout le monde")
```

```
't'
```

```
>>> choice(range(2,89,5))
```

```
27
```

```
>>> choice([["valet","coeur"],["valet","trefle"],["valet","pique"]])
```

```
["valet","pique"]
```

## Fonctions du module random

→ La fonction shuffle “mélange” une liste en utilisant des permutations entre éléments; la liste est alors modifiée

```
>>> l = [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
>>> shuffle(l) # surtout ne pas faire l=shuffle(l)
```

```
>>> l
```

```
[8, 4, 3, 5, 1, 6, 7, 2, 9]
```

## Fonctions du module random

→ la fonction sample

La fonction sample (l , p) renvoie aléatoirement un échantillon de p éléments (différents sauf si plusieurs exemplaires dans l) de la liste l

```
>>>l=[1,2,3,4,5,6,7,8,9,10]
```

```
>>> sample(l,3)
```

```
[4, 2, 9]
```

si p est plus grand que la longueur de l alors une erreur est déclenchée:

ValueError: sample larger than population

# range et les listes

```
>>> l=range(1,15)
```

```
>>> l
```

```
range(1, 15)    #ce n'est pas une liste c'est un "range"
```

```
>>> l[3]
```

```
4
```

```
>>> for x in l:
```

```
print(x,end="")
```

```
1234567891011121314
```

```
>>> l+l
```

```
Traceback (most recent call last):
```

```
File "<pyshell#10>", line 1, in <module>
```

```
l+l
```

```
TypeError: unsupported operand type(s) for +: 'range' and 'range'
```

## Range et les listes:

```
>>> l=range(1,15)
```

```
>>> m=list(l) # on transforme le range en liste
```

```
>>> m
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
```

```
>>> m+m
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 1, 2, 3, 4, 5, 6, 7,  
 8, 9, 10, 11, 12, 13, 14]
```

```
>>>
```



# Des méthodes pour les listes

quelques exemples:

```
lis=[1,2,3,4,5,6,7,8,9,10]
```

`lis.count(x)` renvoie le nombre d'occurrence de `x` dans `lis`

```
>>> lis.count(9)
```

```
1
```

`lis.reverse()` “retourne” (écrit à l'envers) la liste `lis` ;  
attention il s'agit d'une modification physique: rien n'est renvoyé mais la liste `lis` est modifiée.

```
>>> lis.reverse()
```

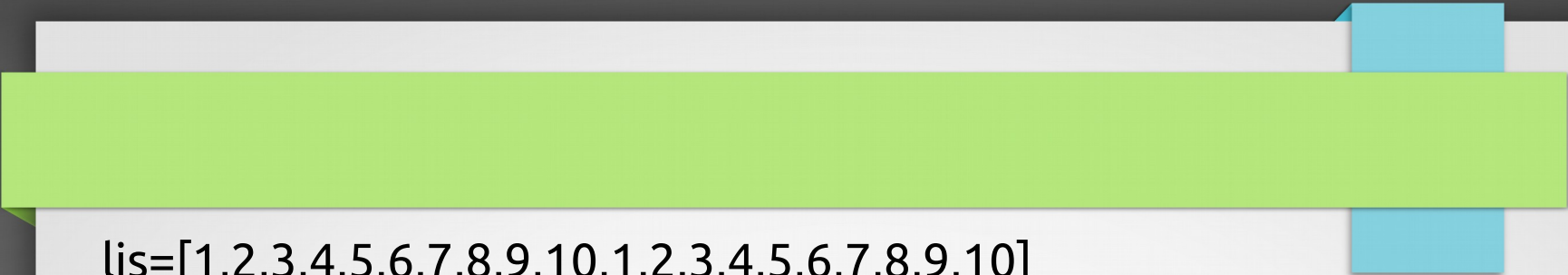
```
>>> lis
```

```
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
```

```
>>> lis.reverse()
```

```
>>> lis
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```



```
lis=[1,2,3,4,5,6,7,8,9,10,1,2,3,4,5,6,7,8,9,10]
```

lis.remove(x) supprime la première occurrence de x dans lis (attention x doit être présent!).

attention il s'agit d'une modification physique

```
>>> lis.remove(6)
```

```
>>> lis
```

```
[1, 2, 3, 4, 5, 7, 8, 9, 10, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

```
>>> lis.remove(11)
```

```
Traceback (most recent call last):
```

```
File "<pyshell#11>", line 1, in -toplevel-
```

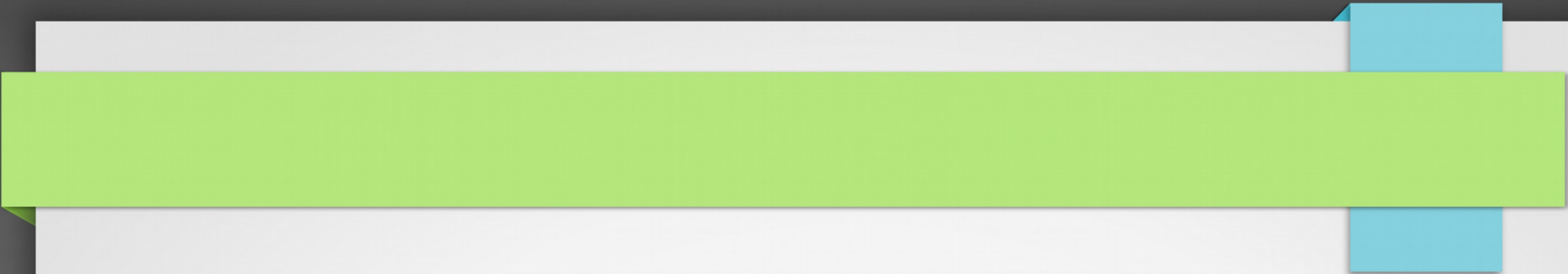
```
lis.remove(11)
```

```
ValueError: list.remove(x): x not in list
```

remarque: del supprime à partir de l'index alors que remove supprime à partir d'une valeur (et la syntaxe n'est pas la même)

lis.sort () modifie la liste lis pour qu'elle soit triée pour l'ordre < (avec modification physique). Evidemment il faut que cela soit possible c'est à dire que les éléments puissent être ordonnés.

```
>>> p=[8,5,9,1,0,4,6,2]
>>> p.sort()
>>> p
[0, 1, 2, 4, 5, 6, 8, 9]
>>> lf=["pomme", "ananas","poire","banane"]
>>> lf.sort()
>>> lf
['ananas', 'banane', 'poire', 'pomme']
```



lis.append ajoute un élément à la fin de la liste lis:  
la différence par rapport à une instruction lis + [x],  
est que c'est une modification physique: lis est  
modifiée et non simplement recopiée avec un  
élément de plus à la fin.

```
>>> l=[1,2,3,4,5,6,7,8,9]
>>> l.append(4)
>>> l
[1, 2, 3, 4, 5, 6, 7, 8, 9, 4]
```

lis.extend(item) permet de rajouter à la fin de lis (toujours avec une modification physique) les éléments de la liste item (qui elle n'est pas modifiée)  
Cela reviendrait à faire lis.append(t) pour chaque élément t de la liste item

```
>>> lis  
[1, 2, 3, 4, 5, 6, 7, 8, 9]  
>>> liste2  
[9, 8, 7, 6]  
>>> lis.extend(liste2)  
>>> lis  
[1, 2, 3, 4, 5, 6, 7, 8, 9, 9, 8, 7, 6]  
>>> liste2  
[9, 8, 7, 6]
```

Il y a d'autres méthodes..... voir la doc!!!



# Deux fonctions un peu spéciales : split

```
>>> ch="python est un langage formidable"
```

```
>>> ch.split()
```

```
['python', 'est', 'un', 'langage', 'formidable']
```

```
>>> ch="1223034505640457"
```

```
>>> ch.split('0')
```

```
['1223', '345', '564', '457']
```

`ch.split(c)` “explode” la string `ch` en morceaux par rapport au paramètre `c` (qui est “ ” par défaut) et renvoie la liste correspondante

## Deux fonctions un peu spéciales : join

```
>>> l=["bonjour", "à" , "tous"]
```

```
>>> " ".join(l)
```

```
'bonjour à tous'
```

```
>>> l
```

```
['bonjour', 'à', 'tous']
```

```
>>> "*".join(l)
```

```
'bonjour*à*tous'
```

Join transforme une liste de chaînes en chaîne

# Attention: del en boucle

Question: Il s'agit, étant donné une liste d'éléments `lis` et un élément `e`, de supprimer tous les éléments égaux à `e` dans la liste `lis`.

On écrira une telle fonction de plusieurs façons

→ Écrire la fonction `supp1(e,l)` qui renvoie une nouvelle liste dans laquelle on a recopié tous les éléments de `lis` s'ils ne sont pas égaux à `e`. On pourra écrire la boucle en une seule ligne.

→ . Écrire une fonction qui modifie la liste `lis` en supprimant tous les `e` en utilisant `del l[i]` pour toutes les cases qui contiennent `e`.

--> Écrire une fonction qui modifie la liste `lis` en supprimant tous les `e` en utilisant `lis.remove` autant de fois que possible.

# Version “recopier”

```
def supprime(x,l):  
    return [y for y in l if y!=x]
```

```
>>> supprime(2,l)  
[1, 1, 1, 1]
```

```
def supprime2(x,l):  
    res=[]  
    for y in l:  
        if y!=x:  
            res.append(y)  
    return res
```

```
>>> supprime2(2,l)  
[1, 1, 1, 1]
```

## Version boucle de del

```
def delsupprimefaux(x,l):  
    for i in range(len(l)):  
        if l[i]==x:  
            del l[i]
```

```
>>> delsupprimefaux(2,l)
```

Traceback (most recent call last):

File "<pyshell#2>", line 1, in <module>

delsupprimefaux(2,l)

File "/export/home/lambert/l1-introduction-programmation/delremove.py",  
line 14, in delsupprimefaux

if l[i]==x:

IndexError: list index out of range



## Version boucle de del

```
def delsupprimejuste(x,l):
```

```
    i=0
```

```
    n=len(l)
```

```
    while i < n:
```

```
        if l[i]==x:
```

```
            del l[i]
```

```
            n=n-1
```

```
        else:
```

```
            i=i+1
```

```
>>> l=[2,1,2,2,2,1,1,1,2]
```

```
>>> delsupprimejuste(2,l)
```

```
>>> l
```

```
[1, 1, 1, 1]
```

```
>>>
```

# Version boucle de remove

```
def removesupprime(x,l):
```

```
    while x in l:
```

```
        l.remove(x)
```

```
>>> removesupprime(2,l)
```

```
>>> l
```

```
[1, 1, 1, 1]
```

```
l=[2,1,2,2,2,1,1,1,2]
```

# les “compréhensions”

Avec for, on peut faire fabriquer des listes très facilement c'est le “for en compréhension”:

```
def carre(l):  
    return [x**2 for x in l]
```

```
>>> l=[5, 8, 2, 1, 6, 9, 3]
```

```
>>> carre(l)
```

```
[25, 64, 4, 1, 36, 81, 9]
```

```
>>> l
```

```
>>> [5, 8, 2, 1, 6, 9, 3]
```

remarque: avec cette méthode, on crée automatiquement une nouvelle liste

# On peut ainsi créer des listes du genre

liste des truc(x) pour x dans une séquence et tels que machin(x)

Dans ce cas le programmeur n'a plus besoin d'écrire la boucle, elle se fait "seule" (mais bien entendu, il y a une boucle). Cela peut être dans une fonction où n'importe où dans un programme.

Exemple: extraire d'une liste la liste des entiers divisibles par 3

```
def extraire3(l):  
    return [ y for y in l if y%3==0]
```

```
>>> extraire3([5,6,2,4,9,23,5,7,1,9],6)  
[6,9,9]
```

→ liste des cubes des entiers de 5 et 15 (inclus):

```
>>> liste=[x**3 for x in range(5,16)]  
>>> liste  
[125, 216, 343, 512, 729, 1000, 1331, 1728, 2197, 2744, 3375]
```

→ fabriquer la liste ["x1", "x2", "x3", "x4",....., "x23"]

```
>>> liste=['x'+ str(i) for i in range(1,24) ]  
>>> liste  
['x1', 'x2', 'x3', 'x4', 'x5', 'x6', 'x7', 'x8', 'x9', 'x10', 'x11', 'x12', 'x13',  
'x14', 'x15', 'x16', 'x17', 'x18', 'x19', 'x20', 'x21', 'x22', 'x23']
```





Remarque: le for en compréhension ne marche pas pour tout!!

En particulier, pas d'utilisation possible pour renvoyer autre chose qu'une liste (par exemple pas possible de calculer une somme directement)

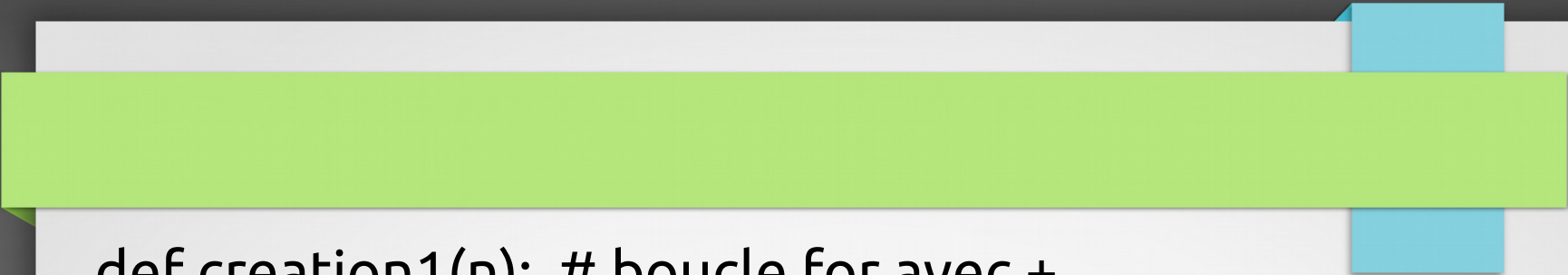
# comparaisons

On dispose de 3 moyens pour fabriquer une grande liste:

- une boucle de +
- une boucle de append
- un for "en compréhension"

On va faire des tests de comparaison des 3 méthodes

Pour cela on va, pour chacune des méthodes, fabriquer une liste de taille  $n$  contenant  $n$  nombres aléatoires.



```
def creation1(n): # boucle for avec +  
    l=[]  
    for i in range(n):  
        l+= [randint(1,1000)]  
    return l
```

```
def creation2(n): # for en compréhension  
    return [randint(1,1000) for i in range(n)]
```

```
def creation3(n): # boucle for avec append  
    l=[]  
    for i in range(n):  
        l.append(randint(1,1000))  
    return l
```

## Et la fonction de test:

```
from time import time
from random import randint

def test(n): # fonction de test
    print("pour n = ", n)
    t0=time()
    creation1(n)
    t1=time()
    creation2(n)
    t2=time()
    creation3(n)
    t3=time()
    print("avec +", t1-t0)
    print("avec for en compréhension", t2-t1)
    print("avec append", t3-t2)
```

## Les résultats: append plus efficace que +

```
>>> test(100)
pour n = 100
avec + 0.0006330013275146484
avec for en compréhension 0.0005497932434082031
avec append 0.0005671977996826172
```

```
>>> test(1000)
pour n = 1000
avec + 0.0061490535736083984
avec for en compréhension 0.002485990524291992
avec append 0.001455068588256836
```

```
>>> test(5000)
pour n = 5000
avec + 0.019189834594726562
avec for en compréhension 0.007050991058349609
avec append 0.0073909759521484375
```



## Compréhensions : doubles ou triples boucles

→ on peut écrire la liste des couples formés avec un élément de chacune de deux listes.

```
>>>liste1=[1,2,3,4,5,6]
>>>liste2=["a","b","c"]
```

```
def prod(l1,l2):
    return [[x,y] for x in l1 for y in l2]
```

```
>>> prod(liste1,liste2)
[[1, 'a'], [1, 'b'], [1, 'c'], [2, 'a'], [2, 'b'], [2, 'c'], [3, 'a'], [3, 'b'], [3, 'c'], [4, 'a'], [4, 'b'], [4, 'c'], [5, 'a'], [5, 'b'], [5, 'c'], [6, 'a'], [6, 'b'], [6, 'c']]
```

remarque: attention à la syntaxe pour les deux variables  
(pas de virgule entre les 2 for)

→ on peut créer une liste de dominos:  
on représente un domino par une liste à deux éléments  
et on suppose les dominos numérotés de 0 à 6:

```
>>> cree_dom = [ [x,y] for x in range(0,7) for y in range  
(0,7) if x<=y]  
# on met x<=y pour ne pas avoir [1,6] et [6,1]
```

```
>>> cree_dom  
[[0, 0], [0, 1], [0, 2], [0, 3], [0, 4], [0, 5], [0, 6], [1, 1], [1, 2],  
[1, 3], [1, 4], [1, 5], [1, 6], [2, 2], [2, 3], [2, 4], [2, 5], [2, 6],  
[3, 3], [3, 4], [3, 5], [3, 6], [4, 4], [4, 5], [4, 6], [5, 5], [5, 6],  
[6, 6]]
```

→ on voudrait obtenir la liste représentant tous les lancers de trois dés tels que la somme des trois dés fasse 6:

```
>>> des= [[x,y,z] for x in range(1,7) for y in range(1,7) for z in range(1,7)
if x+y+z==6]
>>> des
[[1, 1, 4], [1, 2, 3], [1, 3, 2], [1, 4, 1], [2, 1, 3], [2, 2, 2], [2, 3, 1], [3, 1, 2], [3, 2, 1], [4, 1, 1]]
```

→ Si on ne veut pas obtenir à la fois [2, 3, 1] et [3, 1, 2] par exemple, on peut rajouter un test:

```
>>> des= [[x,y,z] for x in range(1,7) for y in range(1,7) for z in range(1,7)
if x+y+z==6 and x<=y<=z]
>>> des
[[1, 1, 4], [1, 2, 3], [2, 2, 2]]
```

# listes de listes

Python gère très bien les listes de listes (comme les tuples de tuples). Quelques exemples:

```
>>> jeucarte = [["valet", "coeur"], ["valet", "pique"],  
                ["dame", "carreau"], ["as", "trefle"]]
```

```
>>> listeEtudiants= [{"dupont", "pierre", [12, 3, 1987],  
                    "info", "0231675423", 12, 4, 15}, {"durant", "julien",  
                    [1, 9, 1987], "info", "0231274239", 2, 14, 8}, {"martin", "anne",  
                    [24, 12, 1988], "info", "0234575478", 12, 14, 13},  
                    {"marie", "lea", [7, 7, 1987], "info", "0223785645", 8, 9, 10}]
```



→ Sur des listes de listes on peut aussi faire un for :  
exemple si on veut faire la somme des dominos d'une liste de dominos (représentant les dominos restant à la fin d'une partie par exemple): on travaille sur une liste style: `[[0,1],[3,4],[5,6]]`

```
def somme1(liste_dominos):  
    res=0  
    for d in liste_dominos: # d=[3,4] par ex  
        res+= d[0]+ d[1]  
    return res
```

Mais on peut aussi écrire le programme avec un parcours des couples de la liste:

```
def somme2(liste_dominos):  
    res=0  
    for [x,y] in liste_dominos:  
        res+=x+y  
    return res
```