Lycée Louis-Le-Grand, Paris
MPSI 4 – Informatique commune
G. Dewaele, A. Troesch

2015/2016

TP n° 5 : Manipulations de listes

Pour tout le TP, on rappelle qu'on accède à l'élément d'indice `i` d'une liste `l` par `l[i]`. On rappelle également qu'on peut extraire des tranches : `l[i:j]` extrait la liste des éléments d'indice i (inclus) à j (exclus). On peut rajouter un pas en 3e paramètre : `l[i:j:p]`. On peut aussi se servir d'une indexation négative permettant d'accèder d'abord aux derniers éléments. Par exemple `l[-1]` est le dernier élément de la liste.

Correction de l'exercice 1 – Manipulations élémentaires de listes

1. Obéissons sagement :

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

   Cela nous renvoie le 5e terme de la liste et non le 4e. Cela résulte du fait qu'en Python, toutes les numérotations commencent à 0.

2. On peut utiliser un terme d'une liste comme une variable, en particulier faire des appels et des affectations :

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

3. La somme de deux listes est bien définie, mais ne correspond pas, comme on pourrait s'y attendre dans une premier temps, à la somme terme à terme, mais à la concaténation des listes. Le produit de deux liste n'est pas défini. Le produit d'une liste par un entier naturel est bien défini, de façon cohérente avec la somme : il s'agit de la concaténation répétée de la liste avec elle-même :

```
>>> l + l2
[1, 2, 3, 4, 7, 4, 3, 2, 1, 0]
>>> l * l2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can't␣multiply␣sequence␣by␣non-int␣of␣type␣'list'
```

   Le produit d'une liste par un entier naturel est bien défini, de façon cohérente avec la somme : il s'agit de la concaténation répétée de la liste avec elle-même :

```
>>> 4 * l
[1, 2, 3, 4, 7, 1, 2, 3, 4, 7, 1, 2, 3, 4, 7, 1, 2, 3, 4, 7]
>>> l * 4
[1, 2, 3, 4, 7, 1, 2, 3, 4, 7, 1, 2, 3, 4, 7, 1, 2, 3, 4, 7]
```

4. Les 3 opérations `l = l + [1]`, `l += [1]` et `l.append(1)` semblent équivalentes. Cependant, à y regarder de plus près, leur effet n'est pas le même sur les adresses mémoires. Les deux dernières conservent la même adresse (autrement dit, il s'agit d'une modification de l'objet existant), la première s'effectue avec changement d'adresse (autrement dit, on définit un nouvel objet).

```
>>> id(l)
140627996952984
>>> l = l + [1]
>>> id(l)
140627991252792
>>> l += [1]
```

```
>>> id(l)
140627991252792
>>> l.append(1)
>>> id(l)
140627991252792
```

5. Après avoir importé le module `random`, on peut consulter l'aide des deux fonctions en tapant `help(random.randint)` et `help(random.sample)` :

```
Help on method randint in module random:

randint(self, a, b) method of random.Random instance
    Return random integer in range [a, b], including both end points.

Help on method sample in module random:

sample(self, population, k) method of random.Random instance
    Chooses k unique random elements from a population sequence or set.

    Returns a new list containing elements from the population while
    leaving the original population unchanged. The resulting list is
    in selection order so that all sub-slices will also be valid random
    samples. This allows raffle winners (the sample) to be partitioned
    into grand prize and second place winners (the subslices).

    Members of the population need not be hashable or unique. If the
    population contains repeats, then each occurrence is a possible
    selection in the sample.

    To choose a sample in a range of integers, use range as an argument.
    This is especially fast and space efficient for sampling from a
    large population: sample(range(10000000), 60)
```

Vous pouvez aussi obtenir l'ensemble des fonctions disponibles dans le module `random` en tapant `help(random)`. On définit alors notre liste simplement par :

```
>>> li = random.sample(range(1,10001), random.randint(10, 10000))
```

La fonction `range` est un itérateur, égrénant les entiers de 1 jusqu'à 10000 (1 de moins que l'entier donné). Si un seul paramètre est donné, l'énumération commence à 0. Si 3 paramètres sont donnés, le troisième paramètre est le pas de l'énumération (voir l'aide).

6. On récupère le dernier terme de la liste en utilisant des indices négatifs :

```
>>> li[-1]
1607
```

7. `help(list)` nous renvoie la liste des fonctions et méthodes disponibles pour la classe `list` On y apprend notamment comment trouver la longueur d'une liste :

```
>>> len(li)
1251
```

8. La première occurrence d'une valeur (et ici la seule), s'obtient ainsi :

```
>>> li.index(1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: 1 is not in list
```

Pas de chance, 1 n'était pas dans ma liste. Après quelques essais, j'ai obtenu une liste contenant 1 :

```
>>> li.index(1)
3658
```

9. La méthode `pop` semble adaptée : elle retourne la valeur en cet place et la supprime de la liste.

```
>>> x = li.pop(5)
>>> x
7312
```

10. On utilise la méthode `sort`, avec le paramètre `reverse = True` :

```
li.sort(reverse = True)
```

Cette méthode admet un autre paramètre `Key`. La clé d'un tri est une fonction f, appliquée à chaque valeur de la liste, le tri se faisant alors sur les valeurs obtenues. Ainsi :

```
li.sort(key = math.sin)
```

trie les valeurs du tableau suivant les valeurs décroissantes de leur sinus. Pour trier suivant une fonction qu'on décrit soi-même, soit on définit cette fonction avec la syntaxe usuelle au préalable, soit on utilise la description `lambda` :

```
li.sort(key = lambda x: x ** 2 - x, reverse = True)
```

11. Assez intuitivement (on peut consulter l'aide pour plus d'information, on y apprend notamment qu'on peut débuter la somme avec une valeur initiale quelconque) :

```
>>> sum(li)
30121192
```

Pour sommer les carrés, on peut écrire : `sum(x ** 2 for x in li)`

Correction de l'exercice 2 – Techniques de slicing sur les listes

On rappelle (voir le cours) qu'étant donné une liste `li`, `li[m:n]` *renvoie un tableau contitué des termes d'indices* m *à* $n-1$ *du tableau* `li`. *Les questions suivantes sont à faire dans l'interpréteur Python.*

1.

```
>>> liste = [2,4,1,8,9,0,2,8,1,10]
>>> len(liste)
10
>>> sousliste = liste[2:6]
>>> sousliste
[1, 8, 9, 0]
>>> id(liste)
140628019470136
>>> id(sousliste)
140627996953560
>>> sousliste.sort(reverse = True)
>>> liste[2:6] = sousliste
>>> liste
[2, 4, 9, 8, 1, 0, 2, 8, 1, 10]
>>> id(liste)
140628019470136
```

La modification d'une tranche n'a pas affecté l'adresse de la liste.

2. On peut insérer la liste comme élément d'elle-même par la méthode `insert`, puis supprimer le terme ajouté avec la fonction `del` :

```
>>> liste.insert(3,liste)
>>> liste
[2, 4, 9, [...], 8, 1, 0, 2, 8, 1, 10]
>>> liste[3]
[2, 4, 9, [...], 8, 1, 0, 2, 8, 1, 10]
>>> del(liste[3])
```

On remarque qu'il s'agit d'une définition récursive : l'indice 3 est égal à la globalité.

Une deuxième méthode consiste à utiliser des slicings, en remplaçant la tranche [3 :3] par une liste contituée d'un seul terme égal à la liste elle-même. On peut alors supprimer ce terme par une technique similaire.

```
>>> liste[3:3] = [liste]
>>> liste
[2, 4, 9, [...], 8, 1, 0, 2, 8, 1, 10]
>>> liste[3:4] = []
>>> liste
[2, 4, 9, 8, 1, 0, 2, 8, 1, 10]
```

Là encore, il s'agit d'une insertion récursive.

Si on n'était intéressé que par une insertion de l'ancienne liste, non récursivement, il fallait d'abord faire une copie de cette liste (par exemple par un slicing) :

```
>>> liste.insert(3,liste[:])
>>> liste
[2, 4, 9, [2, 4, 9, 8, 1, 0, 2, 8, 1, 10], 8, 1, 0, 2, 8, 1, 10]
>>> del(liste[3])
>>> liste[3:3] = [liste[:]]
>>> liste
[2, 4, 9, [2, 4, 9, 8, 1, 0, 2, 8, 1, 10], 8, 1, 0, 2, 8, 1, 10]
```

3. On peut le faire de deux façons différentes que je vous laisse comprendre sur l'exemple ci-dessous :

```
>>> li1 = [2,3,1,4]
>>> li2 = [7,8,9,6]
>>> li3 = li2[:2] + li1 + li2[2:]
>>> li3
[7, 8, 2, 3, 1, 4, 9, 6]
>>> li4 = li2[:]
>>> li4[2:2] = li1
>>> li4
[7, 8, 2, 3, 1, 4, 9, 6]
```

Correction de l'exercice 3 – Mutabilité et copies

Cet exercice est l'occasion de mieux comprendre les notions d'objets mutables, et non mutables (ou immuables). Rappelons qu'une variable est une liaison vers un emplacement mémoire (donc la donnée d'une adresse). L'instruction `y = x` a pour effet de créer pour `y` une liaison vers la même adresse. Si x est réel, et qu'on modifie x, par exemple par l'instruction `x += 1`, on attribue un nouvel emplacement en mémoire pour la variable `x`. En revanche, `y` pointe toujours vers la première adresse dont le contenu n'a pas été modifié. Ainsi, la modification faite sur x n'affecte pas y Dans le cas des listes, il en est autrement :

```
>>> liste1 = [1,2,3,4]
>>> liste2 = liste1
>>> id(liste1)
140628019469920
>>> id(liste2)
140628019469920
```

On a bien les mêmes identifiants, comme dans la situation ci-dessus. En revanche, modifions un des attributs de la liste :

```
>>> liste1[2] += 2
>>> liste1
[1, 2, 5, 4]
>>> liste2
[1, 2, 5, 4]
```

La modification faite sur `liste1` affecte aussi `liste2`. La raison en est qu'une liste est constituée d'une série d'adresses pour chacun des attributs. La variable `liste1` et la variable `liste2` réalisent une liaison vers un même bloc mémoire dans contenant les différentes adresses des attributs. Lorsqu'on modifie un attribut, on peut être amené à modifier son adresse, mais cette nouvelle adresse remplacera l'ancienne dans le bloc mémoire vers lequel pointent `liste1` et `liste2`. Après cette opération, les deux listes sont donc toujours égales.
Pour contourner le problème, il faut faire une copie du bloc d'adresses, de sorte que l'adresse de `liste1` et de `liste2` soient distinctes. Cela peut se faire avec un slicing, ou avec la fonction `copy`

```
>>> liste3 = liste1[:]
>>> id(liste3)
140628019469632
>>> liste2 = liste1.copy()
>>> id(liste2)
140628019469776
>>> liste1[1] = 6
>>> liste1
[1, 6, 5, 4]
>>> liste2
[1, 2, 5, 4]
>>> liste3
[1, 2, 5, 4]
```

En effet, la modification d'un attribut de `liste1` entraîne la modification de l'adresse de cet attribut dans le bloc mémoire associé à la variable `liste1`, mais pas dans les autres blocs associés à `liste2` et `liste3`. Ainsi, les attributs correspondants de ces deux listes ont toujours l'ancienne adresse dont le contenu n'a pas été modifié.
Attention, les choses se corsent si les attributs eux-mêmes sont mutables :

```
>>> liste4= [[1,2,3],4,5]
>>> liste5 = liste4[:]
>>> id(liste4)
140628019469272
>>> id(liste5)
140628019469344
>>> id(liste4[0])
140628019469200
>>> id(liste5[0])
140628019469200
>>> liste4[0][1]= 6
>>> liste4
[[1, 6, 3], 4, 5]
>>> liste5
[[1, 6, 3], 4, 5]
>>> id(liste4[0])
140628019469200
```

En effet, comme on l'a vu plus haut, la modification d'un attribut d'une liste se fait à adresse globale contante : on n'a donc pas modifié l'adresse de l'attribut `liste4[0]`. Or, `liste5[0]` (à partir d'un autre bloc de mémoires) pointe vers la même adresse, donc toutes les modifications faites sur `liste4[0]` le sont aussi sur `liste5[0]`.
Voici une conséquence amusante. Un `tuple` est un objet non mutable. En particulier, on ne peut pas définir de façon isolée la valeur d'un de ses attributs (car on ne peut pas attribuer de nouvelle adresse à cet attribut) :

```
>>> couple = ([1,2],3)
>>> couple[0] = 4
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

Cependant, l'objet `couple[0]` étant une liste, on peut modifier cette liste, sans changement d'adresse global. Cela modifie logiquement le couple :

```
>>> couple[0][1] = 4
>>> couple
([1, 4], 3)
```

Ce qui est immuable dans un `tuple`, c'est le nombre et l'adresse des attributs. Toute modification sur les attributs à adresse constante est valide, et modifie le `tuple`.

Correction de l'exercice 4 – De façon naïve, on définit une fonction de création aléatoire de tableau, et une fonction calculant le nombre d'absents de la façon suivante :

```
def creeliste(n,p):
    """Crée une liste de longueur n constituée d'entiers aléatoires entre 0 et p-1"""
    return [npr.randint(p) for i in range(n)]

def nbr_abs(li,p):
    """Nombre d'absents parmi les entiers de 0 à p-1 dans la liste li"""
    s = 0
    for k in range(p):
        if k not in li:
            s += 1
    return s
```

On pourrait améliorer la première fonction, en remarquant grâce à l'aide disponible que la fonction `randint` du module `numpy.random` admet un paramètre permettant de créer directement un tableau à la taille voulue, dont les entrées sont aléatoires. On peut surtout remarquer que la recherche d'un élément dans un tableau étant linéaire, la seconde fonction a un coût quadratique.
On peut diminuer ce coût de 2 façons :

- soit en travaillant avec des ensembles au liue de listes, l'appartenance d'un élément à un ensemble se faisant en coût constant. Cela nécessite une conversion de type, mais la recherche des absents est particulièrement simple à faire en utilisant les opérations ensemblistes disponibles (c'est le troisième algorithme proposé) ;
- soit en créant une liste de longueur p repérant les absents (l'attribut d'indice i prendra la valeur 1 si i est absent de la liste, 0 sinon). Au départ on l'initialise en mettant des 1 partout. On parcourt alors la liste une seule fois, en mettant à 0 les attributs correspondants de la liste de repères. Il reste alors à sommer les 1 de la liste de repères pour obtenir le nombre d'absent (ce qui se fait aussi en temps linéaire).

```
def nbr_abs2(li,p):
    """Comme nbr_abs"""
    repere = [1 for i in range(p)]
    for j in li:
        repere[j] = 0
    return sum(repere)

def nbr_abs3(li,p):
    """Comme nbr_abs"""
    return len({i for i in range(p)}.difference(set(li)))
```

On peut faire une étude comparative du temps de réponse, en utilisant la fonction `time` du module `time`, calculant la durée écoulée depuis un temps de référence. Pour avoir des temps d'exécution qui se ressentent, je fais le test pour $n = p = 10000$:

```
li = creeliste(10000,10000)
debut = time.time()
print(nbr_abs(li,10000))
fin = time.time()
print('Durée par la méthode 1: {}'.format(fin - debut))
debut = time.time()
print(nbr_abs2(li,10000))
fin = time.time()
print('Durée par la méthode 2: {}'.format(fin - debut))
debut = time.time()
print(nbr_abs3(li,10000))
fin = time.time()
print('Durée par la méthode 3: {}'.format(fin - debut))
```

On obtient :

```
3740
Durée par la méthode 1: 2.1569442749023438
3740
Durée par la méthode 2: 0.0011718273162841797
3740
Durée par la méthode 3: 0.0023987293243408203
```

Pour $n = p = 20000$, on obtient :

```
7324
Durée par la méthode 1: 8.575148344039917
7324
Durée par la méthode 2: 0.002469301223754883
7324
Durée par la méthode 3: 0.004538059234619141
```

Cela illustre bien le coût quadratique de la première méthode (une multiplication par 2 de la donnée engendre une multiplication par 2^2 du temps d'exécution) et le coût linéaire des deux autres.
Nous répétons maintenant l'expérience afin d'obtenit la moyenne du nombre d'absents.

```
def moyenne(N,n,p):
    """Moyenne sur N répétitions du nombre d'absents parmi 1..p-1 dans un tab de lg n"""
    S = 0
    for i in range(N):
        S += nbr_abs2(creeliste(n,p),p)
    return S / N
```

Pour $N = 10000$, $n = p = 100$, on trouve quasi-instantanément : `36.6147`.
La valeur théorique est l'espérance de la variable aléatoire comptant le nombre d'absents. Soit X cette variable aléatoire (pour un choix de n entiers dans $[\![0, p-1]\!]$. En notant, pour $i \in [\![0, p-1]\!]$, X_i la variable aléatoire prenant la valeur 0 si i est dans l'ensemble obtenu, et 1 sinon, on a :

$$X = \sum_{i=0}^{p-1} X_i.$$

Or, les choix étant indépendants,

$$P(X_i = 1) = \left(\frac{p-1}{p}\right)^n.$$

Ainsi, X_i suit une loi de Bernoulli de paramètre $\left(\frac{p-1}{p}\right)^n$, et par conséquent :

$$E(X_i) = \left(\frac{p-1}{p}\right)^n$$

Par linéarité de l'espérance, on obtient :

$$E(X) = n\left(\frac{p-1}{p}\right)^n.$$

Pour $n = p = 100$, on obtient :

```
>>> 100 * ((99 / 100) ** 100)
36.60323412732292
```

Ce résultat est proche de la moyenne trouvée plus haut.

Correction de l'exercice 5 – In and Out Shuffle

Pour les out shuffle, on obtient :

```
def out_shuffle(li):
    return li[0::2] + li[1::2]

def nombre_out_shuffle(li):
    li_init = li.copy()
    i = 0
    while (i == 0) or (li != li_init):
        li = out_shuffle(li)
        i += 1
    return i
```

Pour essayer, on définit une liste, on affiche un out shuffle, puis le nombre nécessaires.

```
li = [i for i in range(52)]
print(out_shuffle(li))
print(nombre_out_shuffle(li))
```

On obtient :

```
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40, 42, 44,
46, 48, 50, 1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29, 31, 33, 35, 37, 39, 41,
43, 45, 47, 49, 51]
8
```

Ainsi, il faut 8 out shuffles pour qu'un jeu de 52 cartes retrouve sa position initiale.
Pour les in shuffle, il n'y a quasiment rien à changer, à part l'ordre dans lequel on recolle les deux demi-paquets :

```
def in_shuffle(li):
    return li[1::2] + li[0::2]

def nombre_in_shuffle(li):
    li_init = li.copy()
    i = 0
    while (i == 0) or (li != li_init):
        li = in_shuffle(li)
        i += 1
    return i
```

Cette fois-ci, il faut répéter 52 in shuffle pour retomber sur la configuration initiale.

Correction de l'exercice 6 –

Nous proposons 4 fonctions permettant de répondre au problème par des méthodes différentes, ou implémentées différemment :

- La première méthode consiste à enlever du tableau les suicidés successifs, grâce à la fonction `del`. On trouve le prochain suicidé en avançant de p dans le tableau (mais comme on vient de supprimer une personne, l'un de ces pas se fait de façon automatique, raison pour laquelle on n'ajoute que $p-1$). On doit réduire modulo la longueur du tableau, pour recommencer au début du tableau lorsqu'on arrive au bout (disposition en cercle). Comme on veut k survivants, on itère $n-k$ fois ce procédé.

- La deuxième méthode consiste à faire tourner la liste, de sorte à ce que la personne suicidée soit toujours en début de liste (cela revient à changer à chaque étape le point initial du cercle). Cela se fait en faisant passer un morceau initial du tableau à la fin.
- La troisième méthode est une variante de la première, mais on ne supprime pas les personnes suicidées du tableau, on les met à 0. A chaque étape, on doit doit trouver la prochaine personne vivante en testant si le nombre qui lui est associé est nul ou non.
- Enfin, on propose une implémentation récursive de la première méthode. La profondeur de récursion étant par défaut limitée à 1000 en Python, pour les tests sur des grands tableaux, il nous faut d'abord changer le paramètre correspondant à la profondeur maximale de récursion. N'abusez pas de ce type de méthodes. Ce n'est d'ailleurs pas dans ce type de situations que les méthodes récursives sont les plus intéressantes.

```
import time
import sys
sys.setrecursionlimit(100000)

# Problème de Josephus

def josephus1(n,k,p):
    """Donne les positions des k derniers survivants dans un
    problème de Josephus sur n personne, de pas p"""
    groupe = [i+1 for i in range(n)] # création du groupe initial
    j=0
    for i in range (n-k):
        del(groupe[j])
        j = (j+p-1)%len(groupe) # on a supprimé une personne, ce qui
                                # crée un décalage de 1 sur les indices
    return groupe

def josephus2(n,k,p):
    """Donne les positions des k derniers survivants dans un
    problème de Josephus sur n personne, de pas p"""
    groupe = [i+1 for i in range(n)] # création du groupe initial
    for i in range (n-k):
        groupe = groupe[1:]
        for i in range(p-1):
            groupe.append(groupe.pop(0))
    return(groupe)

def josephus3(n,k,p):
    """Donne les positions des k derniers survivants dans un
    problème de Josephus sur n personne, de pas p"""
    groupe = [i+1 for i in range(n)]
    j=0
    for i in range(n-k):
        groupe[j]=0
        for k in range(p):
            j = (j+1)%n
            while groupe[j]==0:
               j=(j+1)%n
    survivants = [i for i in groupe if i != 0]
    return(survivants)

def josephus4rec(groupe,k,p,i0):
    if len(groupe)==k:
        return groupe
    else:
```

```
        del(groupe[i0])
        i0 = (i0+p-1)% len(groupe)
        return josephus4rec(groupe,k,p,i0)

def josephus4(n,k,p):
    groupe = [i+1 for i in range(n)]
    return josephus4rec(groupe,k,p,0)


print(josephus1(41,2,3))
print(josephus2(41,2,3))
print(josephus3(41,2,3))
print(josephus4(41,2,3))

for n in [100,1000,10000,100000]:
    print("\nTemps␣d'exécution␣pour␣n␣={},␣k=2,␣p=3:\n".format(n))
    for jos in [josephus1,josephus2,josephus3,josephus4]:
        debut=time.time()
        jos(n,2,3)
        fin=time.time()
        print(fin-debut)
```

Nous obtenons par nos 4 algorithmes la réponse au problème de Josephus, pour un groupe de 41 personnes, 2 survivants et un pas égal à 3 : $[14, 29]$.
Les tests de temps donnent les résultats suivants :

```
Temps d'exécution␣pour␣n␣=100,␣k=2,␣p=3:

5.793571472167969e-05
0.0002079010009765625
0.0002498626708984375
0.00010704994201660156

Temps␣d'exécution pour n =1000, k=2, p=3:

0.0003960132598876953
0.0027399063110351562
0.0030150413513183594
0.0026650428771972656

Temps d'exécution␣pour␣n␣=10000,␣k=2,␣p=3:

0.010905981063842773
0.2707388401031494
0.0377650260925293
0.02138519287109375

Temps␣d'exécution pour n =100000, k=2, p=3:

1.115144968032837
56.63474106788635
0.43638110160827637
Segmentation fault: 11
```

Au passage, remarquez la possibilité d'itérer sur une liste quelconque, y compris une liste de fonctions.
Ces résultats indiquent que la première méthode est la plus efficace. La version récursive l'est un peu moins, à cause des tests supplémentaires qu'elle induit, et la mémoire qu'elle nécessite (d'ailleurs, la mémoire est insuffisante pour

$n = 100000$, ce qu'indique l'erreur retournée pour cette valeur). La troisième méthode est encore raisonnable, mais les tests effectués pour trouver les personnes encore vivantes ralentissent un peu les calculs. En revanche, la deuxième méthode est très mauvaise, du fait des manipulations lourdes sur les listes qu'elle nécessite.

Correction de l'exercice 7 –
On commence par construire la liste des entiers de 2 à n. On selectionne la première entrée de la liste (à savoir 2), et on barre (ici, on supprime) les multiples de cette valeur encore présents dans le tableau. On recommence avec la valeur suivante restant encore dans le tableau, etc.

```
def crible1(n):
    lst = list(range(2, n+1))
    i = 0
    while i < len(lst):
        for k in range(2, n//lst[i]+1):
            if k*lst[i] in lst:
                lst.remove(k*lst[i])
        i += 1
    return lst
```

On peut se rendre compte que le test d'appartenance à `lst` étant linéaire, ainsi que l'opération de suppression de la valeur (pour ce faire, il commence par rechercher la première occurrence), il y a un doublement du temps d'exécution par rapport à ce qu'on pourrait faire. Le test est là essentiellement pour éviter l'erreur qui se produit si la valeur n'est pas dans la liste. On peut éviter ce test en utilisant la structure `try... except...` qui essaye de faire quelque chose, et en cas d'erreur, se reporte à la consigne donnée après le mot `except`. De la sorte, on n'effectue qu'une fois la recherche de l'élément dans la liste, ce qui devrait diminuer de moitié le temps de réponse :

```
def crible2(n):
    lst = list(range(2,n+1))
    i = 0
    while i < len(lst):
        for k in range(2, n//lst[i]+1):
            try:
                lst.remove(k*lst[i])
            except:
                pass
        i += 1
    return lst
```

L'instruction `pass` permet de définir un bloc vide, lorsque la présence de ce bloc dans la structure est imposée (comme ici avec la structure `try... except...`
On peut complètement se dispenser de la recherche dans la liste des multiples en ne supprimant pas les multiples au fur et à mesure, mais en les marquant d'une façon ou d'une autre (en les remplaçant par 0 par exemple). Ainsi, à tout moment, on sait à quels indices se situent les multiples recherchés, ce qui nous permet d'avoir un accès à ces valeurs en temps constant :

```
def crible3(n):
    lst = list(range(2,n+1))
    p = 0
    for i in range(2,n+1):
        if lst[i-2] != 0:
            for k in range(2, n//i + 1):
                lst[k * i - 2] = 0
    return [j for j in lst if j != 0]
```

En utilisant la fonction `time` du module `time`, on peut calculer le temps d'exécution de ces 3 fonctions :

```
debut1 = time.time()
crible1(10000)
fin1 = time.time()
```

```
debut2 = time.time()
crible2(10000)
fin2 = time.time()
debut3 = time.time()
crible3(10000)
fin3 = time.time()
```

Pour $n = 10000$, on trouve :

```
Méthode 1 : 2.0605762004852295
Méthode 2 : 1.5177109241485596
Méthode 3 : 0.006280660629272461
```

Pour $n = 20000$, on trouve :

```
Méthode 1 : 8.277140855789185
Méthode 2 : 6.073685646057129
Méthode 3 : 0.012461662292480469
```

Correction de l'exercice 8 – Dans la définition des diviseurs, il faut rajouter 1 à la liste `entiers`, sinon, il nous manque le diviseur 1.

```
>>> entiers = list(range(2,101))
>>> diviseurs = [ (n,[d for d in [1] + entiers if n % d == 0]) for n in entiers]
```

Les nombres premiers sont alors les nombres ayant exactement 2 diviseurs (1 et eux-même) :

```
>>> premiers = [p[0] for p in diviseurs if len(p[1])==2]
>>> premiers
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97]
```

La recherche des diviseurs par la méthode ci-dessus est quadratique (en considérant le coût de la réducation modulo d comme constant). Les autres instructions étant linéaires, globalement, on obtient un algorithme quadratique, moins bon que ce qu'on peut obtenir par le crible d'Eratosthène (si on le programme correctement).