Compte rendu du T.P. 3, suite

4 Comment fabriquer une liste de nombres aléatoires pour nos tests

a)Savoir importer la fonction et consulter l'aide :

```
from random import randint
help(randint)
Help on method randint in module random:

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

On comprend que : `randint(a,b)` fabrique un nombre entier (pseudo)-aléatoire dans $[\![a, b]\!]$.

b) Fabriquer alors une fonction `liste_alea` qui prend un entier `n` comme entrée et qui fabrique une liste de `n` nombres entiers aléatoires :

La méthode est la même qu'au §2 pour fabriquer une liste de `n` éléments : (si `n` est bien défini) :

```
L=[]
for i in range(n):
    L.append(randint(1,1000))
```

Mais ici on demande en plus de *fabriquer une fonction*, donc on met ce code dans la définition d'une fonction, ce qui donne :

```
def liste_alea(n): # le  n est l'argument de la fonction,
# il sera rentré par l'utilisateur à chaque appel de la fonction
    L=[]
    for i in range(n):
        L.append(randint(1,1000))
    return L
```

5 Maximum dans une liste ou un tuple :

Le but est ici de *comprendre comment trouver le max. d'une liste en parcourant la liste.* C'est notre premier programme de manipulation sur les listes. Il est totalement hors-sujet d'y utiliser des outils plus forts sur les listes comme le tri. En fait dans tout ce qui suit, quand on demande de programmer une opération sur les listes, les seuls outils permis seront les commandes d'extractions $L[i]$ ou $L[i:j]$ et les outils que vous avez déjà développés.

a) Fonction `mon_max` :

Première méthode : en parcourant la liste à l'aide d'une boucle `for` sur les *indices*

```
def mon_max(L):
    "Renvoie le max. d'une liste ou d'un tuple"
    M=L[0]
    for i in range(len(L)):
        if L[i]>M:
            M=L[i]
    return M
```

Deuxième méthode : en parcourant à liste à l'aide d'une boucle `for` sur les *valeurs*

```
def mon_max(L):
    "Renvoie le max. d'une liste ou d'un tuple"
    M=L[0]
    for a in L: # le compteur a parcourt les valeurs dans la liste L
        if a>M:
            M=a
    return M
```

b) Amélioration

```
def max_mieux(L):
    "renvoie la valeur et un indice du max. d'une liste ou d'un tuple"
    M=L[0]
    i_max=0
    for i in range(len(L)):
        if L[i]>M:
            M=L[i]
            i_max=i
    return (M,i_max)
```

Cet algorithme d'obtention de max et de imax est **INCONTOURNABLE** : vous le retrouverez dans la plupart des épreuves écrites de concours sous une forme ou une autre...

6 Programmer le `del`

a) On a parlé en cours de la commande `del` : Si L=[4,3,5,4], que fait `del(L[2])` ?

Réponse : elle *modifie* la liste `L` en enlevant la seconde entrée donc après `del(L[2])`, on aura dans `L` la liste `[4,3,4]`.

b) A l'aide de la technique vue en cours pour extraire une partie d'une liste (*slicing*) fabriquer votre propre fonction `efface` qui fait la même chose que `del`, en prenant en entrée deux arguments `L` et `i`.

Une méthode qui « marche presque » :

```
def mon_del(L,i):
    L=L[0:i]+L[i+1:]
    return L
```

Pourquoi ne fait-elle pas *exactement* **ce que fait le `del` ?** car si la valeur de retour est bien correcte, la liste `L` sur laquelle on appelle la fonction ne sera pas modifiée !

Voir par exemple :

```
>>>L=[0,1,2,3,4]
>>>mon_del(L,2)
[0,1,3,4]
>>>L
[0,1,2,3,4]
```

On comprendra cela en cours avec le cours sur les listes (chap 5) .!

Une méthode qui « marche vraiment » :

```
def mon_delvrai(L,i):
        L[i:i+1]=[] # cette fois pas de L= en local, mais une modification du L global..
        # on verra cela.
```

7 Davantage de méthodes sur les liste :

7.1 Commentaires sur les aides données dans l'énoncé du T.P.

: • `index` : `ma_liste.index(45)` retourne le premier indice où apparaît le nombre 45 dans `ma_liste`, s'il apparaît, sinon donne une erreur. Ce que l'aide met entre crochets est des arguments optionnels qu'on peut rajouter : pour dire qu'on ne regarde que les indices entre .. et ...

Ces crochets ne doivent pas être entrés quand on appelle `index`. Par exemple si on veut chercher l'indice de la valeur 45 pour les indices entre 10 et 17 dans `ma_liste` on écrira `ma_liste.index(45,10,17)`

• `remove` : `ma_liste.remove(45)` enlève la première entrée de la liste où apparaît le nombre 45.

• `pop` : `ma_liste.pop(2)` : fait deux choses. Elle *retourne* l'entrée `ma_liste[2]` et modifie `ma_liste` en enlevant cette entrée.

Ainsi par exemple si `L=[2,3,4,5]`. Après `a=L.pop(2)`, on a `a=4` et `L=[2,3,5]`

• `count` : `ma_liste.count(34)` renvoie le nombre de fois qu'apparaît la *valeur* 34 dans `ma_liste`

• `reverse` : `ma_liste.reverse()` modifie `ma_liste` en la transformant en la liste où les valeurs sont dans l'ordre inverse.

Si `L=[1,2,3]` et qu'on fait `L.reverse()`, on obtient `L=[3,2,1]`

Rappel : les *méthodes* associées à une classe (ici la classe `list`) sont des fonctions particulières qui ne s'appliquent qu'aux objets de la classe, ici donc qu'à des listes. Leur syntaxe est particulière : `nom_de_notre_liste.nom_de_la_methode(argument_suppl)`. Comme une liste est un objet mutable, elles peuvent modifier la liste à laquelle elles s'appliquent.

b) Version maison des méthodes sur les listes :

7.2 Versions maisons de index

En ne tenant pas compte des arguments optionnels donnés par l'aide, une première version que j'ai vue en T.P. est :

```
def mon_index(L,a):
    "renvoie le premier indice où apparaît la valeur a dans L \
et dit 'y'a pas'  sinon "
    for i in range(len(L)):
        if L[i]==a:
            return i
    return "y'a pas"
    # bien comprendre que return fait terminer la fonction....
    #donc la derniere ligne
    # ne s'execute que si on n'a pas trouvé d'indice
    # ce rôle du return a été souligné au chapitre 3, I sur les fonctions.
```

Noter que la réponse *"y'a pas"* n'est pas exactement ce que fait la vraie commande `index`. La vraie commande `index` provoque une erreur.

Pour faire détecter une erreur : `raise ValueError('Message d'erreur')`

D'où le deuxième script :

```
def mon_index(L,a):
    "renvoie le premier indice où apparaît la valeur a dans L \
    et lève une erreur sinon"
    for i in range(len(L)):
        if L[i]==a:
            return i
    raise ValueError("indice non trouvé")
```

La connaissance de ces mots clefs `ValueError` n'est PAS exigible pour les concours.
Dans les épreuves de concours, vous n'aurez pas à « gérer les erreurs ».

7.3 Version maison de `index` avec une boucle `while`

```
def monindice(L,a):
    i=0
    while i<len(L) and L[i]!=a:  # attention à l'ordre des deux conditions.
        i=i+1
    return i
```

Cette fonction renvoie le premier indice `i` tel que `L[i]==a` soit vrai, et s'il n'en existe pas va renvoyer la dernière valeur prise par `i` c'est à dire `len(L)`.

Comparons à ce qui se passe pour le script suivant :

```
def monindicebis(L,a):
    i=0
    while L[i]!=a and i<len(L):
        i=i+1
    return i
```

Si on lance `monindice(L,a)` avec une valeur `a` qui n'apparaît pas dans `L`, la fonction retournera :

Si on lance `monindicebis(L,a)` avec une valeur `a` qui n'apparaît pas dans `L` on aura une `index error`.

Pourquoi ? Et pourquoi cela ne se produisait pas pour `monindice` ci-dessus ?

L'ordre des deux conditions du `while` est important grâce au caractère *paresseux* du `and`

On peut donc rajouter une condition à la fin pour qu'elle fasse la même chose la précédente et que `index` :

```
def monindice(L,a):
    i=0
    while i<len(L) and L[i]!=a:  # attention à l'ordre des deux conditions.
        i=i+1
    if i==len(L) :
     raise ValueError('valeur non trouvée')
    else :
     return i
```

7.4 Version maison de `remove` :

```
def mon_remove(L,a):
    "efface la première entrée de L où apparaît la valeur a, \
    lève une erreur sinon"
    i=mon_index(L,a)
    return  mon_del(L,i)# appelle la fonction mon_del déjà créée.
```

7.5 Version maison de `pop` :

```
def mon_pop(L,i):
    x=L[i]
    mon_del(L,i)
    return x
```

7.6 Version maison de `count` :

```
def mon_count(L,a):
    compteur=0
    for i in range(len(L)):
        if L[i]==a:
            compteur=compteur+1
    return compteur
```

7.7 Version maison de `reverse`

Une solution est la suivante :

```
def mon_reverse(L):
    temp=L[:]# cree une shallow copy de L,
    for i in range(len(L)):
        L[i]=temp[len(L)-i-1]
```

7.8 Version maison de `sort`

Les algo. de tri seront vus plus tard... (2ème année en IPT). **Commentaires sur ce script :**

- La variable `temp` est `temporaire` : cette copie de `L` va permettre de modifier `L` en gardant jusqu'au bout le souvenir de ce qu'il y a au départ de `L` grâce à `temp`.
- Dans la boucle `for`, le compteur `i` parcourt *tous* les indices de la liste : ces indices vont de `0` à `n-1` si on appelle `n=len(L)`.
- Dans cette boucle `for`, le `len(L)-i-1` parcourt les entiers de `n-1` à `0`, donc `temp[len(L)-i-1]` parcourt les valeurs de la liste de l'entrée `n-1` à `0`.
- Avez-vous compris pourquoi on a besoin de `temp` : que se passerait-il si on avait écrit en dernière ligne : `L[i]=L[len(L)-i-1]` ?

Remarque : on peut aussi faire un `reverse` qui n'utilise pas de liste auxiliaire en échangeant $L[i]$ avec $L[n-1-i]$ grâce à une troisième variable tampon... (moins cher en mémoire qu'une liste).

8 Comparaison de la rapidité de deux fonctions : le module `time`

Commentaire du script donné dans le sujet.

```
from time import clock # la fonction clock renvoie un temps d'horloge
def duree(fonction,n=100): # n prendra par défaut la valeur 100
    debut=clock() # on stocke un temps d'horloge dans debut
    fonction(n) # on appelle la fonction avec l'argument n
     #ce qui prend un certain temps d'exécution
    fin=clock() # on stocke le temps d'horloge dans fin
    return fin-debut # la différence représente le temps d'exécution
```

Remarque : le second argument de `duree` est un *argument optionnel* : si on ne l'entre pas i.e. qu'on entre seulement `duree(f)` où `f` est une fonction, alors `n=100` par défaut. En revanche `duree(f,1000)` donnera `n=1000`.

b) Retour sur le2) : y-a-t-il une différence entre les techniques du 2.b), du 2.c) et du 2.d) pour fabriquer des listes.

Tester la durée des deux méthodes pour constituer la liste des `sin(i)` pour $i \in [\![0, 1000]\!]$.

On va l'utiliser pour comparer la rapidité de plusieurs fonctions qui fabriquent la liste des $\sin(i)$ pour $i \in [\![1, n]\!]$.

Par exemple :

```
def fa(n):
    liste=[]
    for i in range(n):
        liste=liste+[sin(i)]
    return liste
```

```
def fc(n):
    liste=[]
    for i in range(n):
        liste.append(sin(i))
    return liste
```

Ensuite, on exécute `test(fa,1000)` et `test(fc,1000)` : résultat ?

Le `append` est bien plus rapide que le `+`.