TP 2 : recherche séquentielle

Par recherche séquentielle, on entend des algorithmes conduisant à passer en revue tous les termes d'une liste ou d'une séquence pour déterminer la position et/ou la valeur de son plus petit ou plus grand élément, de l'indice où figure une valeur donnée etc.

C'est l'occasion de travailler sur l'objet (mal) nommé `list` en python, les boucles `for` et les `range`... On écrira également quelques fonctions.

Lancez spyder (dans Mathématiques→WinPython), créez un répertoire TP2 dans le répertoire InfoPCSI de votre session (si celui-ci n'existe pas, créez-le), et sauvegardez-y le script de l'éditeur de code (fenêtre de gauche) sous le nom `tp2.py`.

1 range et les boucles for

On utilise le plus souvent la fonction `range` en conjonction avec une boucle `for`, afin de répéter une chaîne d'opérations un certain nombre de fois, pour différentes valeurs, souvent consécutives, d'une variable entière donnée. Voici un premier exemple (à essayer dans la console, si vous n'êtes pas certain de ce qu'il fait) :

```
for i in range(5, 10):
    print(i)
```

Question 1. En reprenant le code précédent, quelles valeurs s'affichent avec `range(a)`, `range(a, b)` et `range(a, b, c)` où `a,b,c` sont des entiers (pas forcément naturels) ? (N'hésitez pas à tester !)

Conjecturez l'affichage obtenu avec `range(2, 8, 2)`, `range(1, 5, -1)`, `range(7, 1, -2)` (et vérifiez alors votre conjecture)

Question 2. Calculer la somme $\sum_{k=1}^{100} k\,2^k$. (Indications : on introduira une variable `s`, initialement nulle, et à laquelle nous ajouterons $1\times 2^1, 2\times 2^2, 3\times 2^3$ et ainsi de suite : et ici bien sûr, on utilise une boucle `for`. On indique que pour élever `x` à la puissance `y`, on écrit en python : `x ** y`.)

Réponse.

```
s = 0
for k in range(1, 101):
    s = s + k * 2**k
# notre somme est contenue dans s
```

2 Listes

Une liste en python est une collection ordonnée d'objets (pas forcément tous de même nature).

Par l'instruction `L = [2, 3.1, "une chaine"]`, on introduit une liste comptant 3 objets : un entier, un nombre flottant, et une chaine de caractères, et on lui donne le nom `L` ce qui permet de s'y référer.

La fonction `len` (abréviation de length qui signifie longueur) permet d'obtenir le nombre d'objets de son argument : ici `len(L)` renvoie la valeur 3. On accède, en lecture comme en écriture, aux trois termes de `L` à l'aide de `L[0]`, `L[1]` et `L[2]` :

```
>>> L[1]
   3.1
>>> L[0] = 3; L
   [3, 3.1, "une␣chaine"]
```

Chercher à lire une valeur en dehors des limites de `L` déclenche une erreur (de l'intérêt donc de connaître a priori le nombre de termes que contient `L`) :

```
>>> L[3]
  ...
  IndexError: list index out of range
```

Non seulement peut-on lire comme modifier la valeur de tel ou tel élément d'une liste donnée, mais on peut aussi rajouter ou retirer des éléments d'une liste. En exécutant `L.append(val)` on ajoute l'élément `val` à la fin de la liste `L`. A l'inverse, en exécutant `L.pop()`, on retire le dernier terme de `L`, et on renvoie cette valeur :

```
>>> L.pop()
```

```
  "une␣chaine"
>>> L.append(4)
>>> L
   [3, 3.1, 4]
```

Remarque 1. Si une liste `L` compte par exemple 5 valeurs, alors dans une boucle telle que `for i in range(5):`, `i` prendra pour valeurs tous les indices permettant de parcourir la liste `L`, à savoir les indices 0, 1, 2, 3 et 4. On utilisera alors souvent la syntaxe : `for i in range(len(L)):` pour obtenir tous les indices des éléments de `L`.

3 Fonctions

On va dans ce TP introduire nos premières fonctions en python. Une fonction est, pour faire simple, un ensemble d'instructions permettant de réaliser un traitement donné, admettant 0, 1 ou plusieurs arguments, et renvoyant une valeur. Définir une fonction permet alors, en invoquant celle-ci, de réaliser un nombre répété de fois ce traitement, avec différents jeux de données, sans devoir réintroduire à chaque fois le code de ladite fonction.

La définition d'une fonction prend la forme suivante :

```
def nomFonction(arg1, ..., argN):
    instruction_1
    ...
    instruction_n
    return val
```

La commande `return`, généralement suivie d'une valeur, interrompt immédiatement l'exécution d'une fonction (même si des instructions figurent encore après ce return) et la valeur qui accompagne `return` est ce qu'on appelle la valeur de retour. La présence d'une instruction `return` n'est pas obligatoire, mais en son absence, une valeur par défaut est renvoyée, à savoir la valeur `None`, ce qui est un peu l'équivalent de « rien ». D'ailleurs, lorsqu'on exécute dans la console une fonction qui renvoie `None`, rien ne s'affiche comme valeur de retour. Essayez (on suppose que `L` désigne une liste) les instructions `L.append(4)` puis `print(L.append(5))`

Introduisez dans le script `tp2.py` (normalement, l'éditeur est la fenêtre de gauche) le code suivant :

```
from random import randrange

def listeAleatoire(n):
    L = []
    for i in range(n):
        L.append(randrange(1000))
    return L
```

La première ligne permet d'importer depuis une bibliothèque externe de fonctions (nommée `random` qui comme son nom le laisse supposer est dédiée à la génération de nombres aléatoires) la fonction nommée `randrange`.

Pour que la fonction ici introduite dans le script soit accessible depuis la console, il faut exécuter le script. Pour ce faire, on peut cliquer sur l'icone verte d'une flèche vers la droite, ou utiliser la touche F5 du clavier. Essayez alors dans la console la fonction ainsi introduite avec, par exemple, `listeAleatoire(10)`. (N.B. : à chaque fois qu'on fait des modifications dans notre script, si on veut qu'elle soient prises en compte, on n'oublie pas de rééxécuter le script)

Question 3. Sachant que l'invocation de `randrange(1000)` renvoie un nombre entier pseudo-aléatoire compris entre 0 et 999, qu'effectue l'invocation `listeAleatoire(100)` ?

Réponse. Une liste est créée, d'abord vide, puis on lui ajoute (à la fin), un par un, 100 nombres entiers choisis aléatoirement entre 0 et 999. Une fois ce travail effectué, la liste ainsi obtenue est renvoyée (ce qui permet de lui donner un nom pour effectuer un traitement plus tard par exemple).

Remarque 2. Dans la fonction ci-dessus, outre l'argument `n` donné à la fonction, lequel devra prendre une valeur entière pour que le code s'exécute correctement, il est fait usage d'une variable de boucle `i`, laquelle va prendre les valeurs de 0 à $n-1$ dans l'exécution de la boucle `for`, et dont on n'a que faire une fois que la fonction a achevé son travail. Bonne nouvelle, c'est par défaut une variable *locale* qui est ici introduite, laquelle est créée au moment de l'exécution de la fonction, et est détruite après celle-ci (la mémoire qu'elle occupait alors est libérée) :

```
>>> L = listeAleatoire(10)
>>> i
```

```
...
   NameError: name 'i' is not defined
>>> i = 10
>>> L = listeAleatoire(5)
>>> i
   10
```

Comme on le voit, il n'y a pas de souci qu'on ait, avant d'invoquer notre fonction, nommé `i` un objet. Ce que désigne `i` ne change pas malgré l'exécution de la fonction `listeAleatoire`.

Remarque 3. Quand on invoque une fonction, lss arguments figurent à l'intérieur de parenthèses. C'est par ces parenthèses que l'on reconnaît qu'on invoque une fonction, et même si aucun argument n'est attendu, un couple de parenthèses sera toujours présent, comme par exemple `L.pop()`.

4 Recherche du minimum, maximum

Dans la liste suivante : `L = [24, 64, 12, 10, 53, 85, 25, 63, 53, 78, 13, 75]` quel est le plus petit terme ? le plus grand ? le second plus grand ? Pour chacune de ces questions, on tâchera d'en donner la réponse en passant en revue, une et une seule fois, l'ensemble des termes de `L`.

Ces questions peuvent paraître stupides, mais réfléchissez au travail que vous avez effectué pour y répondre : c'est tout simplement ce qu'il va falloir automatiser pour qu'une fonction réalise ce travail pour vous. (A noter que pour le maximum et le minimum d'une liste, python a déjà défini deux fonctions, opportunément nommées `min` et `max`, mais il n'est pas question ici d'invoquer l'une ou l'autre…)

Question 4. Recopiez et compléter le code suivant, de recherche du minimum d'une liste :

```
def minimum(L):
    mini = L[0]
    for i in range(...):
        if ...:
            ...
    return ...
```

Bien sûr, on testera, par exemple avec :

```
>>> L = listeAleatoire(100)
>>> minimum(L), min(L)
>>> minimum(list(range(10)), minimum(list(range(10,1,-1)))
```

(Si vous n'êtes pas certain de la liste obtenue avec `list(range(10,1,-1))` n'hésitez pas à tester dans la console !)

et on vérifiera que les valeurs renvoyées sont correctes.

Réponse.

```
def minimum(L):
    mini = L[0]
    for i in range(1, len(L)):
        if L[i] < mini:
            mini = L[i]
    return mini
```

Question 5. En vous inspirant de la question précédente, proposez une fonction d'entête `def maximum(L):` qui renvoie, vous l'avez deviné, le plus grand élément de `L`.

Réponse.

```
def maximum(L):
    maxi = L[0]
    for i in range(1, len(L)):
        if L[i] > maxi:
            maxi = L[i]
    return maxi
```

Question 6. On aimerait désormais non plus connaître la valeur de plus grand terme de `L`, mais l'indice où cette valeur figure. Si le maximum de `L` figure plusieurs fois dans `L`, on aimerait connaître le premier indice où il apparaît.

Ecrire donc une fonction d'entête `def indiceMaximum(L):` qui réalise ce travail. Un exemple d'exécution :

```
>>> indiceMaximum([2, 4, 6, 3, 6, 1])
    2
```

Réponse.

```
def maximum(L):
    imaxi = 0
    for i in range(1, len(L)):
        if L[i] > L[imaxi]:
            imaxi = i
    return imaxi
```

5 Recherche d'un élément

On cherche ici si une valeur donnée figure dans une liste, et si oui, à quel indice.

Question 7. Ecrire une fonction d'entête `def indiceDe(L, val):` qui renvoie le premier indice `i` tel que `L[i]` soit égal à `val` si `val` figure dans `L`, et qui renvoie `None` sinon. (Pas forcément besoin d'un `return` dans ce cas, `None` étant la valeur de retour par défaut)

On s'inspirera des fonctions précédentes, et on pourra tenir compte du fait que l'instruction `return` interrompt aussitôt l'exécution de notre fonction car, en pratique, c'est bien ce que l'on cherche à faire : dès lors que la valeur souhaitée est trouvée, nul besoin de continuer de parcourir les termes de `L` à la recherche de notre valeur…

Réponse.

```
def indiceDe(L, val):
    for i in range(len(L)):
        if L[i] == val:
            return i
```

Question 8. On reprend notre recherche, mais on souhaite désormais connaître non pas le premier indice où figure une valeur donnée dans une liste mais, sous la forme d'une liste, tous les indices où cette valeur apparaît. On écrira pour ce faire une fonction d'entête `def indicesDe(L, val):`

Un exemple d'utilisation :

```
>>> indicesDe([2, 4, 6, 3, 6], 6)
    [2, 4]
>>> indicesDe([2, 4, 6, 3, 6], 1)
    []
```

Réponse.

```
def indicesDe(L, val):
    indices = []
    for i in range(len(L)):
        if L[i] == val:
            indices.append(i)
    return indices
```

6 Travail à faire à la maison et à envoyer par mail

Si vous avez le temps d'aborder en TP les questions suivantes, profitez-en ! Sinon, je vous invite à terminer les questions qui suivent à la maison, et à m'envoyer par mail (`phjondot@gmail.com`), sous la forme d'un fichier python, vos réponses à ces questions. Si vous rencontrez une difficulté, les questions sont les bienvenues, et il n'y a rien de dramatique à devoir s'y reprendre à plusieurs fois avant d'obtenir un code fonctionnel (il n'empêche que je vous invite à tester votre code sur quelques exemples avant que de le soumettre !)

Question 9. D'une liste `L` de nombres, écrivez une fonction d'entête `def longueurMaxProgressionCroissante(L):` qui admet pour unique argument une liste `L` et compte le plus grand nombre de termes en progression croissante (au sens large). Par exemple, pour `[2, 3, 6, 2, 3, 4, 7]` on renverra 4 (trois termes en progression croissante, suivis de 4 termes en progression croissante).

Réponse. Une réponse possible :

```
def plusGrandeLongueur(L):
    if len(L) == 0:
        return 0
    maxLong, longCourante = 1, 1
    for i in range(1, len(L)):
        if L[i] >= L[i-1]:
            longCourante += 1
            if longCourante > maxLong:
                maxLong = longCourante
        else:
            longCourante = 1
    return maxLong
```

En général, les réponses à cette question ont été bonnes. La petite erreur souvent rencontrée étant le recours à une inégalité stricte pour comparer `L[i]` et `L[i+1]`, ce qui ne répond alors pas exactement à la question (croissance au sens large)

Question 10. Reprendre la fonction précédente pour qu'elle renvoie non pas le plus grand nombre de termes en progression coissante, mais une liste de termes consécutifs maximale extraite de `L`. On nommera `sousListeMaxProgressionCroissante` cette nouvelle fonction.

En d'autres termes, `sousListeMaxProgressionCroissante([2, 3, 6, 2, 3, 4, 7])` devra renvoyer `[2, 3, 4, 7]`.

Si plusieurs sous-listes de `L` conviennent, on renverra la première rencontrée dans une lecture de gauche à droite de `L` (par exemple `sousListeMaxProgressionCroissante([2, 4, 6, 1, 3, 5])` renverra `[2, 4, 6]` plutôt que `[1, 3, 5]`.

Réponse. Voici deux réponses parmi celles que j'ai reçues. Dans la première, on garde trace des indices qui délimitent la sous-liste que l'on va extraire :

```
def sousListeMaxProgressionCroissante(L):
    if len(L) == 0:
          return []
    maxLongueur, longCourante = 1, 1
    maxStart = 0
    couranteStart = 0
    for i in range(1,len(L)):
        if L[i-1]<= L[i]:
            longCourante += 1
        else:
            if longCourante > maxLongueur:
                maxLongueur, maxStart = longCourante, couranteStart
            longCourante, couranteStart = 1, 1
    if longCourante > maxLongueur:
        maxLongueur = longCourante
        maxStart = couranteStart
    return L[maxStart:maxStart+maxLongueur]
```

Dans la seconde on construit des sous-listes, en gardant trace de la meilleure :

```
def sousListeProgressionCroissante(L):
    if len(L)==0:
        return []
    sousListeMax= [L[0]]
    sousListeCourante = [L[0]]

    for i in range(1, len(L)):
        if L[i] >= L[i - 1]:
            sousListeCourante = sousListeCourante + [L[i]]  # remarque 1
        else:
            if len(sousListeCourante)>len(sousListeMax):
                sousListeMax = sousListeCourante.copy() # remarque 2
                sousListeCourante = [L[i]]
    if len(sousListeCourante)> len(sousListeMax):
        sousListeMax = sousListeCourante
    return sousListeMax
```

Remarque 1 : pour ajouter un élément à la fin d'une liste, la syntaxe proposée conduit bien à ce que l'on souhaite ici, mais de manière peu efficace, car la somme `sousListeCourante + [L[i]]` conduit à construire une nouvelle liste qui concatène (met bout à bout) les deux listes que sont `sousListeCourante` et `[L[i]]`, et une fois cette nouvelle liste construite, on lui donne le nom de `sousListeCourante` (ce qui signifie que la liste que représentait jusque-là `sousListeCourante` est alors détruite et la place occupée par celle-ci libérée…)

Mieux vaut donc faire appel à la méthode `append` d'une liste dont le rôle est exactement d'ajouter un élément à la fin d'une liste, laquelle sera modifiée. Il se trouve que cet ajout en fin de liste est une opération qui, en temps moyen, est à coût constant et indépendant de la longueur de la liste initiale, contrairement à l'opération présentée précédemment qui présente un coût proportionnel à la longueur de la liste.

Remarque 2 : Réaliser une copie (dite superficielle) d'une liste est également une opération qui n'opère pas en temps constant, mais qui demande un temps proportionnel à la longueur de la liste que l'on recopie. Une autre syntaxe pour réaliser une telle copie serait d'écrire `sousListeMax = sousListeCourante[:]` (choix fait par certains d'entre-vous)

Réaliser une copie aurait du sens si on a besoin de garder trace du contenu exact d'une liste qu'on va être amené à modifier. Dans le cas présent, ce n'est en fait pas utile, car la liste représentée par `sousListeCourante` ne va pas être modifiée dans la suite puisqu'à la ligne suivante `sousListeCourante` ne référence plus la même liste mais est réinitialisée à une autre liste.

La version corrigée qui suit est alors beaucoup plus efficace, puisque agit en temps linéaire selon la longueur de `L` (ce qui indique qu'en doublant la longueur de la liste `L`, le temps nécessaire sera doublé) alors que la fonction précédente agit en temps quadratique (le temps est quadruplé si ou double la longueur de la liste `L`)

```
def sousListeProgressionCroissante(L):
    if len(L)==0:
        return []
    sousListeMax= [L[0]]
    sousListeCourante = [L[0]]

    for i in range(1, len(L)):
        if L[i] >= L[i - 1]:
            sousListeCourante = sousListeCourante.append(L[i])
        else:
            if len(sousListeCourante)>len(sousListeMax):
                sousListeMax = sousListeCourante
                sousListeCourante = [L[i]]
    if len(sousListeCourante)> len(sousListeMax):
        sousListeMax = sousListeCourante
    return sousListeMax
```

On peut tester avec, par exemple `sousListeProgressionCroissante(list(range(10000))` et vous verrez une différence très significative entre les deux fonctions…