Feuille de TP Python 10

Elements d'analyse textuelle, tris

L'objet de ce TP est de présenter les méthodes de tri du programme en lien avec quelques techniques d'analyse textuelles.
Ce TP est associé à une archive contenant données, canevas d'exercices et corrections.

1 Chaînes de caractères, fichiers en mode texte

1.1 Partout, des chaînes de caractères

Une chaîne de caractères est un objet fondamental en informatique !

Les plus petites unités physico-logiques traitées par processeur d'ordinateur sont les *bits*, des composants électroniques [1] dont l'état se mesure en 0 ou 1. Ces unités sont traditionnellement regroupées en paquets de 8 [2] : les *octets*. Ces octets sont eux-mêmes regroupés –physiquement– en paquets de quelques millions ou milliards (par exemple dans les barettes de mémoire RAM).

Un octet, c'est donc 8 bits pouvant chacun (et indépendammment) valoir 0 ou 1 et il y a donc $2^8 = 256$ configurations d'octets différentes :

$$\texttt{00000000}, \texttt{00000001}, \texttt{00000010}, \ldots, \texttt{11111110}, \texttt{1111111}$$

C'est une quantité idéale pour utiliser les octets pour coder les lettres de l'alphabet latin (majuscules et minuscules), les chiffres, les caractères typographiques usuels (visibles et invisibles, comme par exemple le « retour chariot » créé lorsque l'on frappe la touche ⏎ .

Le codage le plus courant de ces caractères par les entiers de 0 à 255 (et donc l'ensemble des octets) s'appelle le codage ASCII. Chaque octet est donc un caractère, une suite d'octets s'apparente à un texte et, ça tombe bien, un texte peut servir à coder de l'information, que ce soit une suite d'instructions à exécuter (un programme ou script) ou des données : des tables de chiffres aux romans, des noms de fichiers à leur contenu.

1.2 Les chaînes de caractères en Python : déclaration des constantes, assignation

- Un caractère en Python est de type `chr`. On le décrit en plaçant un caractère entre deux `'`, par exemple `'a'` décrit le caractère « a ».
- Une chaîne de caractères, de type `str`, se décrit par un bloc de `chr` encadrés de deux `'`, de deux `"` ou de deux `"""` [3]. Par exemple, sont des spécifications de chaînes de caractères :

  `'abcd'`, `"abcd"`, `"""abcd"""`,

- Evidemment « ' », « " » sont eux-mêmes des caractères et si'on veut les placer dans une chaîne de caractères, il faut les « échapper » à l'aide du caractère « \ ». La chaîne `"bla\"bla"` représente le mot « bla"bla ». De même, les caractère spéciaux (invisibles ou peu visibles) s'écrivent à l'aide de code : `'\n'` est le caractère de « passage à la ligne » et la chaîne `"Bonjour\nTout le monde"` représente le texte « Bonjour
Tout le monde » [4]
- Enfin, le caractère spécial de chez spécial « \ » s'obtient par `'\\'`.

Travail ! Exercice 1.—Copier et exécuter le script suivant, répondre à la question !

```
print("\\Combien y-a-t\'il de caractères \\ imprimés à l\'écran ? \\\n et dans cette chaîne ?")
```

et pour celui-ci ?

```
print(r'\\Combien y-a-t\'il de caractères \\ imprimés à l\'écran ? \\\n et dans cette chaîne ?')
```

1. l'électronique est le support physique le plus courant pour l'informatique mais il en existe d'autres, par exemple des vannes hydrauliques dans certains automates hydrauliques utilisés dans des environnements où l'électricité est proscrite. Ces vannes peuvent être ouvertes ou fermées et le flux électrique est remplacé par le flux d'un fluide.
2. ou 16 ou 32 ou 64
3. cette syntaxe est utilisée dans les `docstring`, elle permet d'avoir de longs textes comportant des passages à la ligne
4. en deux lignes, c'est pour ça qu'on l'a encadré, ce texte.

Exercice 2.—Copier et exécuter le script suivant : quelle gamme de numéros de codes ASCII est montrée ? Expérimentez avec vos caractères préférés pour récupérer leurs codes : Le mien est '€'. Quel est son code ? Pourquoi n'est-il pas dans la gamme habituelle demandant un octet de codage ?

```
print("Table ASCII")
for i in range(10, 153):
    print("code : ",i,", caractère :'", chr(i), "'")
car = 'a'
while len(car) == 1 :
    car = input("entrez un caractère au clavier puis <ENTER> (<ENTER> seul pour stopper):")
    if len(car) == 1 :
        print("car :'",car,"', code :", ord(car))
```

Changer la dernière ligne en

```
        print("car :'{}', code : {}".format(car,ord(car)))
```

Quelle est la différence au niveau de l'affichage ? Décrire l'action *méthode* `.format` sur un objet Python de type `str`.

1.3 Accèder, modifier, concaténér

Une chaîne de caractères Python se comporte comme un t-uple de caractères :

— La longueur de la chaîne (le nombre de caractères) est accessible par `len` ;

— L'égalité de deux chaînes se teste avec ==, leur différen,ce avbec `!=`, leur ordre avec < ,>, >= ou <=.

— L'opération de concaténation [5] se note avec + ;

— du point de vue de l'accès, la syntaxe indicielle et l'extraction de tranches est possible ;

— du point de vue de la modification aucune modification n'est possible, il faut recréer une nouvelle chaîne à partir de fragments :

```
>>> s = 'Le vif zéphyr jubile sur les kumquats du clown gracieux' ; s[7] ='a'

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

Il faut faire

```
s = s[:7] + 'a' + s[8:]
```

Travail ! Exercice 3.—

1. Deviner (calculer !) les chaînes désignées par les identicateurs `a,b,c,d,e` :

```
s = 'Le vif zéphyr jubile sur les kumquats du clown gracieux'
a = s[7] ; b = s[-2] ; c = s[4:8] ; d = s[:len(s)//2] ; e = s[(len(s)//2):] ;
```

2. Donner les valeurs des booléens `d == e`, `d != e`, `d < e`, `d >= e`.

3. Ecrire une boucle qui cherche la chaîne `'kumquats'` dans la chaine `s`, la remplacer une fois par la chaîne `'citrons et kumquats'`

4. Pourquoi le code suivant ne s'arrêtera-t-il pas ? (boucle infinie)

```
ss = 'kumquats' ; sr = 'citrons et kumquats'
ell = 0
while ell < len(s)-len(ss):
    if s[ell:ell + len(ss)] == ss :
        s = s[:ell] + sr + s[ell+len(ss):]
    ell += 1
```

1.4 Eclater une chaîne en liste, concaténer les chaînes d'une liste

Etant donné un texte donner dans une chaîne `s` et un caractère de « séparation » `sep`, on peut l'éclater en une liste de sous-chaînes avec la méthode `.split`, la syntaxe est `L = s.split(sep)`

Réciproquement, étant donnée une liste de chaînes `L`, on peut former une longue chaine en les concaténant avec un séparateur `sep` imposé, la syntaxe est `sep.join(L)`

Travail ! Exercice 4.—Saisir et exécuter (ensuite, isualiser `S`) le code suivant (on suppose que la chaîne `s` est déjà définie) :

```
S = s.split() #Par défaut le séparateur est un espace blanc quelconque
s = "#".join(S)
```

Pourquoi les instructions suivantes permettent-elles de (presque toujours) retrouver la chaîne `s` initiale ?

```
S = s.split('#')
s = " ".join(S)
```

5. « mise bout à bout »

1.5 Charger un texte à partir d'un fichier texte

Un texte long est la plupart du temps sauvegardé dans un fichier sur disque (ou sur le réseau). On peut lire un tel fichier pour importer son contenu dans une chaîne Python.

```
fichier = open(fichier_nom,'r')
s = fichier.read()
fichier.close()
```

On peut aussi lire et traiter les données ligne par ligne

```
fichier = open(fichier_nom,'r')
L = fichier.readlines()
s = ''
for ell in L :
    s = s + ell
    #Ou un autre traitement à partir de la ligne ell
fichier.close()
```

NB : Le nom du fichier (*i.e.* l'emplacement des données sur le disque) est une chaîne de caractères définie préalablement à l'ouverture du fichier.

```
fichier_nom = 'disparition.txt'
```

Travail ! Exercice 5.—Dans le texte du fichier `'disparition.txt'` (disponible dans l'archive), combien y-a-t'il de 'a' ? de 'y' ? de 'e' ? Est-ce normal, docteur ?

On devra utiliser un nouveau script nommé `disparition.py` qui sera complété dans les exercices suivants.

2 Compter avec un dictionnaire

Un aspect fondamental de l'analyse textuelle consiste à dénombrer les mots intervenant dans le texte. La structure de dictionnaire est particulièrement adaptée pour le décompte d'objets dont on ne connait pas le nombre *a priori* .

L'algorithme est le suivant :

Données: $T = (T[k])_{1 \leq k \leq n}$ est une liste de données, $n \geq 1$.
Résultat: Un dictionnaire D, indiquant pour chaque valeur de T en clé, le nombre d'occurrences dans la liste T.
début
  pour $i = 1$ *à* n **faire**
    si $T[i]$ *clé de* D **alors**
      $D[T[i]] += 1$
    sinon
      créer $D[T[i]] = 1$

Travail ! Exercice 6.—En complétant le script `disparition.py`, écrire une fonction `Occurrences(fichier_nom)` qui construit le dictionnaire des occurrences de chaque mot contenu dans le fichier portant le nom `fichier_nom`

Tester et afficher le dictionnaire ainsi construit. Etes vous satisfaite du résultat ? Si oui, on arrête, si non, comment améliorer le résultat ?

3 Trier

3.1 Pourquoi trier ?

Le tri de données et d'informations est une des opérations les plus fondamentales de l'informatique.

On effectue un tri sur un ensemble de données pour pouvoir accéder plus rapidement à certaines caractéristiques de cet ensemble, pour pouvoir sélectionner les données les plus pertinentes ou éliminer les moins pertinentes

1. Une recherche GOOGLE retourne une liste de liens triés par ordre décroissant de pertinence : les difficultés sont multiples dans ce cadre : donner un critère de pertinence, ordonner une liste gigantesque de sites plus ou moins pertinents, etc...
2. En statistiques : ayant une suite de nombres issus par exemple de mesures, on les trie pour accéder à des caractéristiques statistiques importantes de la suite : médiane, quartiles, déciles et éliminer les *outliers*, les données aberrantes.

   C'est ce type d'applications que nous allons un peu aborder dans les §4 et 5.

3. En algorithmique : certains algorithmes nécessitent que les données soient ordonnées en entrée ou incluent des tris à l'intérieur même de l'algorithme.

3.2 Clés de tri

Pour trier en ordre croissant ou décroissant une liste d'informations, il faut disposer d'un critère de comparaison de ces informations, *i.e.* disposer d'un ordre.

Certains types de données disposent d'un ordre naturel (les nombres entiers ou réels), pour trier une liste de nombres, on peut donc se baser sur cet ordre pour effectuer le tri. Dans ce cas, on dispose d'un vecteur (ou suite finie) T de n nombres réels et on cherche à le mettre par ordre croissant (ou décroissant).

Les chaines de caractères possèdent elles aussi un ordre naturel, l'ordre *lexicographique*. Il y a cependant toujours des embrouilles au niveau de la différence majuscule/Minuscule, caractères accentués ou autres caractères spéciaux (!, ?,...). Avant de faire confiance à un tri sur des chaines de caractères, il vaut mieux tester l'ordonnancement réel des caractères.

D'autres types de données n'ont pas d'ordre naturel ou objectif et il faut alors disposer d'un critère pour effectuer l'ordonnancement. Dans ce cas on dispose d'un tableau [6] D ou d'une liste finie de données $D = (d_k)$

Une méthode naturelle (mais pas la seule) est d'associer à chaque donnée d_k un nombre entier ou réel t_k, sa *clé*.

Pour trier D, il suffit alors de trier $T = (t_k)$ et de garder trace de la façon dont les indices ont été déplacés.

De façon alternative, Python permettant l'accouplement des données, on dispose d'une suite TD de couples de la forme (nombre réel,donnée associée) et on cherche à trier cette suite de couples avec comme « ordre » sur les couples

$$(t_1, d_1) \leq (t_2, d_2) \Leftrightarrow t_1 \leq t_2$$

3.3 Python sait naturellement trier

Le langage Python est à base de listes (une liste est formée, au niveau primitif, d'une zone de mémoire « contigüe » contenant des « pointeurs » vers les objet de la liste) et une méthode de tri [7] de ces listes est fournie par le langage.

Cette méthode est *très* optimisée et est plus efficace [8] que toute méthode que vous pourriez implémenter directement dans le langage Python.

On peut utiliser une association, via un dictionnaire, entre objets à trier (en clés du dictionnaire) et clés de tri (en valeurs du dictionnaire).

Dans toutes les applications que vous aurez à construire si vous avez besoin d'un tri, utilisez celui-ci qui est très optimisé.

4 Analyser un texte

Travail ! Exercice 7.—Cet exercice nécessite un script `disparition.py` en bon état de marche.
1. Utiliser le script `analyse-tri-builtin.py`. Analyser ce script : quelles sont les commandes effectuant des tris ? Comment s'effectue un tri avec une clé ?
2. Afficher, dans l'ordre décroissant du nombre d'occurences tous les mots ayant été utilisés plus de 10 fois dans le texte. Quels sont les noms des personnages principaux ?

6. Un point de vocabulaire important : Au niveau de précision où nous nous plaçons dans ce texte, en algorithmique, on peut utiliser indifféremment les termes « tableau » et « liste ». La différence entre les deux concepts apparait lorsque les termes sont complémentés d'adjecifs et/ou de précisions : un tableau bidimensionnel de taille fixe peut-être simulé par une liste de listes et une `list` Python est en fait l'implémentation de la structure de données souvent dénommée « tableau dynamique » en algorithmique standard.
7. `timsort`, *cf.* `http://en.wikipedia.org/wiki/Timsort`
8. Elle est à la fois aussi voire plus efficace que les algorithmes présentés en fin de TP et elle est codée en un langage de bas niveau, beaucoup plus rapide que l'interpréteur Python.

5 Statistiques d'ordre

Travail ! Exercice 8.—Avec un nouveau script `fonction-repartition.py`

1. Lire et comprendre le corps de la fonction `FR(L,x)`. On précisera l'algorithme utilisé en détaillant ce qui est décompté lors de la double boucle while.

Le tri *built-in* est utilisé pour accélérer les calculs.

2. Tracer le graphe de la fonction de répartition d'une liste de `N` = 100000 nombres tirés au sort suivant une loi $\mathcal{N}(0,1)$.

3. Comment utiliser un tri pour déterminer la médiane d'une série numérique `L` ? Ecrire une fonction `Mediane(L)` ?

Le premier quartile ? le dernier quartile ?

6 Différents algorithmes

6.1 Trouver un plus grand/plus petit élément dans une liste

Cette question n'est pas à proprement parler une question de tri. Une erreur fondamentale commise par les débutants pour trouver maximum et minimum d'une liste consiste à trier cette liste par ordre croissant et d'en extraire ensuite premier et dernier élément : il ne faut surout pas faire cela ! ! !

Travail ! Exercice 9.—On se donne une liste T de nombres réels indicée de 0 à $N-1$.

1. Ecrire, en pseudo-code ou en Python, une fonction `Max(T)` retournant l'élément maximal de la liste T.

2. Ecrire, en pseudo-code ou en Python, une fonction `Max2(T)` retournant l'élément maximal de la liste T et le premier indice où celui-ci est atteint.

3. On se donne un tableau T à deux dimensions de nombres réels indicé par les couples de $\{1,\dots,N\}\times\{1,\dots,P\}$. Ecrire, en pseudo-code ou en Python, une fonction `MinMax(T)` retournant le minimum des maxima de chaque ligne.

6.2 Intercaler deux listes triées.

Cette question n'est pas à proprement parler une question de tri (*cf.* ci-dessous le tri fusion) mais utilise le fait que les tableaux en jeu sont déjà triés.

Travail ! Exercice 10.— Dans un script nommé `trifusion.py`, écrire une fonction Python `Fusion0(T,S)` qui fusionne deux tableaux d'entiers *ordonnés* `T` et `S` en un seul tableau ordonné et retourne ce tableau.

6.3 Aspects algorithmiques des tris

L'importance de comprendre les diverses méthodes de tri n'est donc pas « pratique ». Il s'agit pour nous essentiellement

1. de toucher à de l'algorithmique élémentaire,
2. de saisir les contraintes/questions qui se posent lors de l'élaboration d'un algorithme : complexité en temps (nombre d'opérations), complexité en espace (taille mémoire nécessaire à la complétion de l'opération)

Les algorithmes de tri (un *tri* pour simplifier) peuvent avoir (ou pas) un certain nombre de caractéristiques. A titre d'exemples

— Le tri est dit *en place* s'il travaille à l'intérieur de la liste sans besoin de la dupliquer. L'espace mémoire supplémentaire pour stocker les objets reste réduit et l'algorithme est de faible *complexité spatiale*.

— Le tri est dit *stable* s'il préserve l'ordre d'apparition des objets en cas d'égalité des clés. Le tri fourni par Python est stable. On peut faire une cascade de tris portant sur des clés différentes en conservant l'ordre obtenu précédemment en cas d'égalité.

— Enfin, une question importante est celle de la *complexité temporelle* de l'algorithme, *i.e.* une estimation du nombre d'opérations à faire en fonction du nombre n de données. Les tris présentés (Tri à bulles, tris par insertion) en première année sont en $O\left(n^2\right)$. Il existe de meilleurs algorithmes : tri de SHELL en $O\left(n.(\ln n)^2\right)$, *quicksort*, tri par fusion, qui sont en $O\left(n.\ln n\right)$.

6.4 Algorithmes en $O\left(n^2\right)$

On prie la lectrice d'aller relire ce qui a été fait à ce propos en première année ! Il serait bien de connaitre *par coeur* l'algorithme de tri par insertion.

7 Algorithmes en $O(n.\ln n)$. Récursivité

7.1 Une idée de la récursivité

Une fonction est dite *récursive* si sa définition utilise la fonction à définir elle-même, mais sur des données plus « petites ». L'exemple le plus bateau est celui de la fonction factorielle qui pourrait se définir en Python par

```
def factorielle(n) :
    if n==0 :
        return 1
    else :
        return factorielle(n-1)*n
```

Il s'agit d'une définition « mise en abyme[a] ». Il doit être clair que pour que le processus de définition soit valable, cette mise en abyme s'arrête à un moment et donc que, sur les données « minimales », la fonction donne une réponse directe.

a. *cf.* `http://fr.wikipedia.org/wiki/Mise_en_abyme`

Pink Floyd, Pochette de l'album *Ummagumma*, 1969

7.2 Récursivité : `Quicksort`

L'algorithme L'algorithme `Quicksort` est un algorithme qui se décrit simplement de manière récursive.
Soit T notre tableau, indicé[9] de ℓ à h, à trier.

1. La première étape est l'étape de *partition* : On se donne un élément π du tableau : le *pivot* de la méthode et on déplace les éléments du tableau T de sorte que à gauche du pivot se trouvent les éléments inférieurs (ou égaux) à π et qu'à droite se trouvent les éléments supérieurs (strictement) à π. Dans cette nouvelle configuration, l'élement π reçoit l'indice p.
2. La deuxième étape est l'étape de récursion. On trie (par la même méthode) d'une part la partie gauche, située entre les indices ℓ et $p-1$, du tableau T et d'autre part, la partie droite, située entre les indices $p+1$ et h. Si l'une de ces parties est vide ou ne contient qu'un élément, elle n'a pas besoin d'être triée.

Travail ! Exercice 11.—

1. Charger le script `quicksort.py`. Effacer les corps des fonctions `partition0` et `quickSort`
2. Ecrire, dans le script, la fonction Python `partition0(T,l,h)` où T est un tableau qui implémente la première étape entre les indices `l` (attention ! cette lettre est le ℓ !) et `h-1` en prenant comme pivot l'élément `T[h-1]`. Cette fonction doit
   (a) modifier le tableau `T`
   (b) retourner l'indice de pivot `p`
3. Ecrire, en suivant le principe de la deuxième étape, une fonction Python `quickSort(T,l,h)` où `T` est le tableau à trier, entre les indices (au sens large) `l` et `h-1`. Par défaut `l` vaut 0 et `h` vaut `len(T)` de sorte que l'appel `quickSort(T)` trie tout le tableau `T`.
4. Tester
5. Observer le corps de la fonction Python `partition(T,l,h)`. Quel élément est choisi[10] comme pivot ?.

7.3 Récursivité : le tri fusion

L'algorithme Le *tri fusion*, que l'on peut pratiquer à la main lorsque l'on doit classer des fiches par ordre alphabétique est l'algorithme suivant

1. Couper le tas en deux tas de tailles comparables
2. Trier chacun des tas (Etape récursive, on utilise l'algo. lui même pour faire cette étape, sauf si le tas est réduit à une fiche)
3. Fusionner les deux tas ordonnés en les intercalant de manière à obtenir un tas ordonné.

Travail ! Exercice 12.—Dans le script `trifusion.py` commencé lors de l'exercice 10, érire une fonction Python `TriFusion0(T)` qui retourne une version triée du tableau `T` en appliquant l'algorithme de tri fusion.

Tester sur une liste de 1000 nombres tirés au sort.

Une correction est disponible : Utiliser le script `trifusion.py`.

9. ℓ comme *low*, h comme *high*
10. il s'agit de la médiane des trois éléments `T[l]`, `T[c]` `T[h-1]` où `c` est un indice à mi-chemin entre les deux extrémités. Cette correction combat le mauvais comportement de la première version de l'algorithme sur des données couramment rencontrées, par exemple lorsque les données sont déjà triées mais en ordre décroissant