Informatique
TP4 : Manipulations de fichiers
Manipulations de chaînes et de tableaux
CPP 1A

Djamel Aouane, Frederic Devernay, Matthieu Moy

Mars - avril 2015

1 Manipulations de fichiers

Pour organiser des données sur un disque dur, on utilise généralement des fichiers et des répertoires (parfois appelés « dossiers »).

- Les fichiers (« file » en anglais) contiennent l'information à proprement parler. Un fichier est une suite d'octets (1 octet = 8 bits = 1 nombre de 0 à 255). Par exemple, pour un fichier contenant un programme python, et plus généralement pour tous les fichiers textes, chaque octet correspond à un caractère du programme. Pour d'autres formats de fichier plus évolués (texte mis en forme par LibreOffice ou Word, image JPEG ou PNG...), la suite de caractères n'a pas forcément de sens pour un être humain, mais les logiciels appropriés savent les lire.
- Les répertoires (« directory » en anglais) contiennent des fichiers ou d'autres répertoires.

Lancez le gestionnaire de fichiers en cliquant sur l'icône « Dossier personnel de ... » sur le bureau. Nous utilisons ici Linux, mais l'équivalent existe bien sûr sous Windows (répertoire « Mes documents » par exemple) et OS X. Créez un répertoire `TP4` dans votre répertoire personnel. Vous ferez l'ensemble de ce TP dans ce répertoire.

Vous trouverez sur la page du cours un petit programme Python `make_dirs.py`.

Téléchargez ce programme et placez-le dans le répertoire `TP4` que vous venez de créer (votre navigateur le téléchargera probablement dans un répertoire `Téléchargement`, il faudra donc le déplacer). Vous n'avez pas besoin d'en comprendre le contenu.

Chargez ce programme dans Spyder avec le menu « File », « Open... », puis exécutez-le. Il va créer pour vous quelques répertoires et fichiers.

Si vous revenez à la fenêtre du gestionnaire de fichiers graphique, vous devriez trouver, dans le répertoire `TP4`, des nouveaux répertoires `dir1`, `dir2` et `dir3` et des nouveaux fichiers `file1` et `file2`. Ouvrez le répertoire `dir1` et regardez son contenu. Il contient entre autres un sous-répertoire `subdir1`, ouvrez-le.

Nous allons maintenant parcourir ces répertoires avec un programme Python. Au final, le programme devra afficher la structure des répertoires de la manière suivante :

```
file1
dir1 (repertoire, contient : tutu-3.txt, subdir2, toto-1.py, subdir1, toto-2.txt)
make_dirs.py
```

```
dir3 (repertoire, contient : toto-16.txt, subdir2, subdir1, toto-15.py, tutu-17.txt)
file2
dir2 (repertoire, contient : toto-8.py, toto-9.txt, subdir2, subdir1, tutu-10.txt)
```

1.1 Contenu d'un répertoire

Créez un nouveau programme Python (par exemple, `ls.py`[1]), et écrivez-y le contenu suivant :

```
import os

def ls():
    for e in os.listdir('.'):
        print(e)
```

Exécutez-le et appelez la fonction `ls` (on peut soit appeler `ls()` directement depuis l'interprète interactif, ou bien ajouter un appel `ls()` *en dessous* de sa déclaration dans le fichier). Vous devriez obtenir quelque chose comme (l'ordre peut changer) :

```
ls.py
make_dirs.py
dir1
dir2
dir3
file1
file2
```

Explications :

- `os.listdir` est une fonction Python qui permet de lister les éléments du répertoire passé en paramètre. Il renvoie une liste de chaînes qui sont les noms des fichiers ou répertoires. Notre boucle `for` permet donc de parcourir les fichiers et répertoires, et de les afficher avec `print`.
- Le répertoire courant s'appelle « `.` ». `os.listdir('.')` veut dire « liste le contenu du répertoire courant ».
- Comme d'habitude, `import os` est nécessaire pour utiliser une fonction du module `os` (OS = « Operating System »).

Exercice 1 (Distinction des fichiers et des répertoires) *À l'intérieur de la boucle `for`, distinguez (avec un `if`) le cas des répertoires. La fonction Python `os.path.isdir(nom)` permet de savoir si nom est un répertoire. Si `e` est un répertoire, affichez `(repertoire)` à côté de son nom. Sinon, gardez l'affichage tel qu'il était.*

Vous devriez maintenant obtenir un affichage ressemblant à :

```
ls.py
make_dirs.py
dir1 (repertoire)
dir2 (repertoire)
dir3 (repertoire)
file1
file2
```

1. `ls` est la commande Unix qui permet de lister le contenu d'un répertoire, plus ou moins ce qu'on est en train de faire en Python

Il faut maintenant lister le contenu des répertoires (`dir1`, `dir2` et `dir3`). Pour cet exercice, nous n'irons pas plus loin : nous ne chercherons pas à lister le contenu des sous-répertoires de `dir1`, `dir2` et `dir3`.

1.2 Affichage du contenu des répertoires

Nous allons maintenant compléter notre affichage dans le cas des répertoires. Pour éviter de tout mettre dans la même fonction, créez une fonction `show_dir(d)` qui contient pour l'instant :

```
def show_dir(d):
    print(d + " (repertoire)")
```

Modifiez votre fonction `ls` pour appeler cette fonction correctement. La fonction doit maintenant ressembler à :

```
def ls():
    for e in os.listdir('.'):
        if os.path.isdir(e):
            show_dir(e)
        else:
            print(e)
```

Exercice 2 (Affichage du contenu des répertoires) *Modifiez maintenant la fonction* `show_dir` *pour produire l'affichage « (repertoire, contient : tutu-3.txt subdir2 toto-1.py subdir1 toto-2.txt) ».*

Conseil : créer une variable `content` *de type chaîne, qui vaut initialement* `""` *(chaîne vide). Parcourez le répertoire avec une boucle* `for`*, et à chaque tour de boucle ajoutez l'élément à la chaîne avec :*

```
content = content + " "
content = content + element
```

Terminez l'affichage avec une instruction :

```
print(d + " (repertoire, contient : " + content + ")")
```

Exercice 3 (Affichage d'une liste dont les éléments sont séparés par des virgules) *Reprenez le programme de l'exercice précédent et remplacez l'espace (*`" "`*) qui sépare chaque élément dans la variable* `content` *par une virgule (*`", "`*). Exécutez votre programme : vous devriez avoir un affichage* presque *correct, mais il y a sans doute une virgule en trop au début de la liste. On peut s'en débarrasser en n'exécutant pas* `content = content + ", "` *au premier tour de boucle. Pour cela, ajoutez une variable* `first` *initialisée à* `True` *avant la boucle et positionnée à* `False` *pendant le premier tour. Utiliser un* `if` *autour de* `content = content + ", "`*.*

2 Calcul à partir de données contenues dans un fichier

Nous allons maintenant faire un calcul simple sur des données lues depuis un fichier. On suppose qu'un instrument de mesure a fourni des données dans un fichier texte, avec une valeur par ligne, comme ceci :

```
1
12.3
43
3
10
```

Le but de cette section est de calculer la moyenne de ces valeurs.

2.1 Lecture d'un fichier ligne à ligne

On commence avec le programme suivant :

```
f = open('donnees.txt', 'r')
ligne = f.readline()
while ligne != '':
    print("ligne =", ligne)
    ligne = f.readline()
```

Explications :

- Pour lire dans un fichier, il faut d'abord l'ouvrir. C'est ce que fait la fonction `open`, qui ouvre le fichier `donnes.txt` en lecture (`'r'`, pour « read »).
- La fonction `open` renvoie un objet `f` que l'on peut utiliser avec `f.readline()` qui veut dire « lire la ligne suivante dans le fichier ». Une fois la fin du fichier atteint, `f.readline()` renvoie la chaîne vide.

La variable `ligne` va donc contenir successivement les chaînes `"1"`, `"12.3"`, `"43"`, `"3"` et `"10"` [2]. On va maintenant extraire les nombres contenus dans cette ligne, et en faire la moyenne.

Exercice 4 (Ordre des instructions dans la boucle) *Essayez d'inverser les lignes* `print(ligne)` *et* `ligne = f.readline()` *et exécutez le programme. Cette inversion provoque deux problèmes : la première ligne n'est plus affichée, et une ligne blanche est affichée en fin de programme. Expliquez ces problèmes.*

En pratique, il est donc important d'exécuter `ligne = f.readline()` en dernier : c'est cette instruction qui marque le passage à l'itération suivante, donc c'est la dernière chose qu'on fait avant de revenir en tête de boucle et de tester `ligne != ''` à nouveau.

Exercice 5 (Calcul de moyenne) *En ajoutant quelques lignes au programme ci-dessus, calculez la moyenne des nombres lus. Attention, la fonction* `f.readline()` *renvoie une chaîne de caractère. Pour la convertir en nombre, on peut utiliser* `float(...)`*.*

On pourrait bien sûr généraliser la méthode à des fichiers d'entrée plus compliqués, par exemple avoir plusieurs valeurs par ligne (typiquement séparées par des virgules).

3 Manipulation de chaînes de caractères

La bibliothèque standard de Python contient beaucoup de fonctions de manipulations de chaînes évoluées. Le but ici est de retrouver les algorithmes sans utiliser ces fonctions.

Le but de cette partie est d'écrire une fonction `recherche_mot(m, t)` qui recherche le mot `m` dans le texte `t`. `m` et `t` sont deux chaînes de caractères.

L'algorithme est le suivant : pour chaque position `i` à l'intérieur de la chaîne `t`, on va vérifier si `m` correspond à la sous-chaîne de `t` démarrant à l'indice `i`.

2. plus précisément, la variable `ligne` contient elle-même une fin de ligne, ce qui fait que `print(ligne)` affiche deux retours à la ligne

Exercice 6 *Écrire une fonction* `coincide(t, i, m)` *qui renvoie* `True` *si la sous-chaîne de* `t` *démarrant à l'indice* `i` *et de la même longueur que* `m` *est égale à* `m`*, et* `False` *sinon.*

Par exemple, `coincide("ceci est un test de texte", 12, "test")` *et* `coincide("ceci est un test de texte", 0, "ceci")` *renvoient* `True`*, mais* `coincide("ceci est un test de texte", 11, "test")` *renvoie* `False`*.*

Exercice 7 (Recherche de sous-chaîne) *En utilisant la fonction* `coincide(t, i, m)`*, écrire une fonction* `recherche_mot(m, t)` *qui renvoie l'indice de* `t` *où se trouve le mot* `m` *s'il existe, et la valeur* `None` *sinon.*

Vérifiez sur quelques exemples que la fonction fonctionne comme prévu.

Exercice 8 (Complexité) *Quelle est la complexité de cet algorithme ?*

En pratique, on sait faire beaucoup mieux que cet algorithme, et arriver à un coût linéaire après un pré-traitement de la chaîne à rechercher (algorithme de Knuth-Morris-Pratt par exemple). Les mêmes algorithmes peuvent être utilisés avec autre chose que des chaînes de caractères (exemple : recherche d'un gêne dans une séquence d'ADN).

4 Recherche de valeur dans un tableau

Nous avons vu en cours la recherche linéaire dans un tableau, qui est rappelée ici :

```
def recherche(v, liste):
    for e in liste:
        if v == e:
            return True
    return False
```

Si on veut que notre fonction renvoie l'indice de la valeur quand elle est trouvée, on peut utiliser :

```
def recherche(v, liste):
    for i in range(len(liste)):
        if liste[i] == v:
            return i
    return None
```

Une autre méthode, si le tableau est trié, est la recherche dichotomique : on va utiliser deux variables `gauche` et `droite` pour représenter la portion de tableau dans laquelle la valeur `v` est susceptible de se trouver (l'indice de la valeur est dans l'intervalle [`gauche`, `droite`]). Initialement, `gauche` vaut donc 0 et `droite` vaut `len(t) - 1`. L'idée est de restreindre l'intervalle au fur et à mesure de la recherche. On s'arrête quand on trouve la valeur, ou quand `droite < gauche` (i.e. la portion de tableau dans laquelle la valeur `v` est susceptible de se trouver est vide). À chaque étape, on calcule `m = (gauche + droite) // 2` et on regarde la valeur du tableau se trouvant à cet indice :

- Si la valeur est égale à `v`, on a trouvé la solution.
- Si elle est plus grande que `v`, alors il faut chercher dans la moitié gauche du tableau (entre `gauche` et `m - 1`)
- Si elle est plus petit, c'est l'inverse.

L'intérêt de la recherche dichotomique est qu'elle est beaucoup plus rapide (Si n est la taille du tableau, la recherche dichotomique est en $O(\log(n))$ au lieu de $O(n)$ pour la recherche linéaire).

Exercice 9 (Recherche dichotomique dans un tableau trié) *Écrire la fonction* `recherche(v, liste)` *en utilisant la méthode de recherche dichotomique.*

5 Pour ceux qui n'en ont jamais assez ...

Si vous terminez le TP en avance, vous pouvez vous amuser à améliorer vos programmes, par exemple :

- Permettre à votre fonction `ls` de prendre en paramètre le nom du répertoire à lister.
- (Difficile) Modifier votre fonction `ls` pour qu'elle liste le contenu des sous-répertoire, sous-sous-répertoires, et ainsi de suite. Il faut pour cela que `ls` soit récursive (i.e. s'appelle elle-même).
- Modifiez le programme de l'exercice 5 pour faire le calcul en deux temps : lecture des valeurs depuis le fichier texte vers une liste Python, puis parcours de la liste Python pour calculer la moyenne.

6 Solutions

Section 1 :

```
import os


def show_dir(d):
    content = ""
    first = True  # variable vraie au premier tour de boucle
    for e in os.listdir(d):
        if first:
            first = False
        else:
            content = content + ", "
        content = content + e
    print(d + " (repertoire, contient : " + content + ")")


def ls():
    for e in os.listdir('.'):
        if os.path.isdir(e):
            show_dir(e)
        else:
            print(e)


ls()
```

Note : `if first:` pourrait aussi s'écrire `if first == True:`, mais le `== True` n'est pas nécessaire : `first` est une variable booléenne, on peut l'utiliser directement dans l'argument du `if`. De la même manière en français, on peut écrire « si je suis au CPP alors je fais du Python » ou bien « si la phrase "je suis au CPP" est vraie alors je fais du Python », c'est équivalent.

Exercice 4 : Attention, il y a une ligne `ligne = f.readline()` dans l'initialisation avant de rentrer dans la boucle. Donc, si on fait `print("ligne =", ligne)` après `ligne = f.readline()` dans la boucle, on aura bien exécuté deux fois `ligne = f.readline()` avant le premier `print`, donc la première ligne est lue mais pas affichée. À la dernière exécution, on affiche nécessairement une ligne vide puisque le dernier `print` serait exécuté entre le dernier `readline` et le test de sortie de boucle.

Exercice 5 :

```
f = open('donnees.txt', 'r')
# Première valeur
ligne = f.readline()
nombre = 0
somme = 0
while ligne != '':
    v = float(ligne)
    somme = somme + v
    nombre = nombre + 1
    # Lecture de la valeur suivante
    ligne = f.readline()
print("La moyenne des lignes est :", somme / nombre)
```

Exercice 6 : En n'utilisant que les opérations de base :

```
def coincide(t, i, m):
    for j in range(len(m)):
```

```
        if t[i + j] != m[j]:
            return False  # On a trouvé une différence, pas
                          # la peine de continuer la boucle.
    # Si on arrive ici, c'est que la boucle a terminé et qu'on
    # a comparé tous les caractères. Les chaines coincident.
    return True
```

Une autre solution en utilisant les tranches de tableaux Python :

```
def coincide(t, i, m):
    return t[i:i + len(m)] == m
```

Exercice 7 :

```
def recherche_mot(m, t):
    for i in range(1 + len(t) - len(m)):
        if coincide(t, i, m):
            return i
    return None
```

Exercice 8 : La fonction `coincide` fait `len(m)` itérations dans le pire cas. Elle est appelée `1 + len(t) - len(m)` fois. La complexité est donc `len(m)`$*$`(1 + len(t) - len(m))` (pire cas).

Exercice 9 : L'invariant de boucle est que l'élément recherché est dans l'intervalle [`gauche`, `droite`]. À chaque tour de boucle, on met à jour `gauche` ou `droite` pour maintenir cet invariant.

La ligne **`return`** `m` n'est atteinte que si `t[m] == v`, donc dans ce cas la valeur renvoyée est la bonne. La ligne **`return`** `None` n'est atteinte que si `gauche > droite` donc si [`gauche`, `droite`] est vide, auquel cas l'élément cherché ne peut pas être dans le tableau.

```
def recherche_valeur(v, t):
    gauche = 0
    droite = len(t) - 1
    while gauche <= droite:
        # Décommenter pour voir l'exécution :
        # print(gauche, droite)
        m = (gauche + droite) // 2
        if t[m] == v:
            return m
        elif t[m] < v:
            gauche = m + 1
        else:
            droite = m - 1
    return None

t = [0, 1, 2, 3, 5, 12]
print(recherche_valeur(4, t))
print(recherche_valeur(5, t))
print(recherche_valeur(3, t))
print(recherche_valeur(0, t))
print(recherche_valeur(12, t))
```