Les fichiers

Complément - niveau basique

Voici quelques utilisations habituelles du type `file` en python

Avec un *context manager*

Nous avons vu dans la vidéo les mécanismes de base sur les fichiers. Nous avons vu notamment qu'il est important de bien fermer un fichier après usage.

Dans la pratique, il est recommandé de **toujours** utiliser l'instruction `with`, que nous approfondirons en semaine 6. Dans l'immédiat notez qu'au lieu de faire

```
# la méthode "de bas niveau" n'est plus recommandée
# depuis l'introduction de l'instruction 'with'
sortie = open("s1.txt", "w")
for i in range(2):
    sortie.write("{}\n".format(i))
sortie.close()
```

On fera plutôt

```
# de cette manière, on garantit la fermeture du fichier
with open("s1.txt", "w") as sortie:
    for i in range(2):
        sortie.write("{}\n".format(i))
```

Depuis son introduction dans python-2.5, cette forme est devenue très populaire car elle présente le gros avantage de **garantir** que le fichier sera **bien fermé**, et cela même si une exception devait être levée à l'intérieur du bloc `with`. Et marginalement le code est plus lisible dans cette deuxième forme.

Les modes d'ouverture

Les modes d'ouverture les plus utilisés sont

- `'r'` (la chaîne contenant l'unique caractère `r`) pour ouvrir un fichier en lecture seulement;
- `'w'` en écriture seulement; le contenu précédent du fichier, s'il existait, est perdu;
- `'a'` en écriture seulement, mais pour ajouter du contenu à la fin de fichier.

Voici par exemple comment on pourrait ajouter du texte dans le fichier `s1.txt` qui devrait, à ce

stade contenir 2 entiers

```
# on ouvre le fichier en mode 'a' comme append - ou ajouter
with open("s1.txt", "a") as sortie:
    for i in range(100, 102):
        sortie.write("{}\n".format(i))

# maintenant on regarde ce que contient le fichier
with open("s1.txt") as entree: # remarquez que sans 'mode', on ouvre en lecture se
ule
    for line in entree:
        # comme line contient deja la fin de ligne
        # on ajoute une virgule pour eviter une deuxieme fin de ligne
        print line,
```

Il existe de nombreuses variantes au mode d'ouverture, pour par exemple:

- ouvrir le fichier en lecture et en écriture,
- ouvrir le fichier en mode binaire,
- utiliser le mode dit ***universal newlines*** qui permet de s'affranchir des différences de fin de ligne entre les fichiers produits sur, d'une part linux et MacOS, et d'autre part Windows.

Ces variantes sont décrites dans la section sur la fonction built-in `open` (https://docs.python.org/2/library/functions.html#open) dans la documentation python.

Comment lire un contenu (haut niveau)

Les fichiers textuels classiques se lisent en général, comme on vient d'ailleurs de le faire, avec une simple boucle `for` sur l'objet fichier, qui itère sur les lignes du fichier. Cette méthode est **recommandée** car elle est **efficace**, et n'implique pas notamment de charger l'intégralité du fichier en mémoire.

On trouve aussi, dans du code plus ancien, l'appel à la méthode `readlines` qui renvoie une liste de lignes:

```
# il faut éviter cette forme qu'on peut trouver dans du code ancien
with open("s1.txt") as entree:
    for line in entree.readlines():
        print line,
```

Vous remarquerez cependant que cette méthode implique de **charger l'intégralité** du fichier en mémoire. Similairement à la discussion que nous avons eue en fin de semaine 3 sur les itérateurs et la performance, ici aussi l'utilisation du fichier comme un itérateur est de loin préférable.

Complément - niveau intermédiaire

Un fichier est un itérateur

Pendant que l'on parle d'itérateur, remarquons qu'un fichier - qui donc **est itérable** puisqu'on peut le lire par une boucle `for` - est aussi **son propre itérateur**:

```
# un fichier est son propre itérateur
```

```
with open("s1.txt") as entree:
    print entree.__iter__() is entree
```

Dans le prolongement de la discussion sur les itérateurs et les itérables en fin de semaine 3, nous vous rappelons que tous les autres types de base que nous avons vus jusqu'ici ont leur itérateurs implémentés comme des objets séparés. Nous avions à l'époque souligné que ce choix permettait notamment de réaliser deux boucles imbriquées sur la même liste.

```
# deux boucles imbriquées sur la même liste fonctionnent comme attendu
liste = [1, 2]
for i in liste:
    for j in liste:
        print i, "x", j
```

Par contre, écrire deux boucles `for` imbriquées sur **le même objet fichier** ne **fonctionnerait pas** comme on pourrait s'y attendre :

```
# Si on essaie d'écrire deux boucles imbriquées
# sur le même objet fichier, le résultat est inattendu
with open("s1.txt") as entree:
    for l1 in entree:
        # on enleve les fins de ligne
        l1 = l1.strip()
        for l2 in entree:
            # on enleve les fins de ligne
            l2 = l2.strip()
            print l1, "x", l2
```

Autres méthodes

Vous pouvez également accéder à des fonctions de beaucoup plus bas niveau, notamment celle fournies directement par le système d'exploitation; nous allons en décrire deux parmi les plus utiles.

Lire un contenu - bas niveau

La méthode `read()` permet de lire dans le fichier un buffer d'une certaine taille:

```
# lire dans le fichier deux blocs de 4 caractères
with open("s1.txt") as entree:
    for bloc in xrange(2):
        print "Bloc {} >>{}<<".format(bloc, entree.read(4))
```

Le premier bloc contient bien 4 caractères si on compte les deux sauts de ligne

```
Bloc1 = "0\n1\n"
```

Le second bloc contient quant à lui

```
Bloc2 = "100\n"
```

Avec la méthode `read` mais sans argument, on peut lire tout le fichier d'un coup - mais là encore prenez garde à l'utilisation de la mémoire

```
with open("s1.txt") as entree:
    contenu = entree.read()
    print "Dans un contenu de longueur {} " \
          "on a trouvé {} occurrences de 0" \
          .format(len(contenu), contenu.count('0'))
```

La méthode `flush`

Les entrées-sortie sur fichier sont bien souvent ***bufferisées*** par le système d'exploitation. Cela signifie qu'un appel à `write` ne provoque pas forcément une écriture immédiate, car pour des raisons de performance on attend d'avoir suffisamment de matière avant d'écrire sur le disque.

Il y a des cas où ce comportement peut s'avérer gênant, et où on a besoin d'écrire immédiatement (et donc de vider le ***buffer***), et c'est le propos de la méthode `flush`().

Pour en savoir plus

Pour une description plus exhaustive vous pouvez vous reporter à la page sur le type `file` (https://docs.python.org/2.7/library/stdtypes.html?highlight=file%20 object#file-objects) dans la documentation python.