MANIPULATION DES FICHIERS PAR PYTHON

1 MOTIVATION

Un programme sera souvent amené à traiter un grand nombre de données et à en générer.
Il ne serait pas réaliste de vouloir tout rentrer à chaque fois à la main avec des commandes `input`. Nous n'allons pas non plus écrire ces données directement dans le programme, car il faudrait modifier notre script pour chaque changement de données, au risque d'introduire de nombreuses erreurs. Nous utiliserons donc des fichiers externes que Python lira et éventuellement modifiera.
Ceci permet aussi de pouvoir sauvegarder des données générées par Python : par exemple des scores pour un jeu, ou bien des données statistiques sur une étude... L'avantage de ce fichier et qu'il restera en mémoire même après la fermeture de Python, contrairement aux variables qui sont effacées lorsque l'on ferme Python. De la sorte, vous pourrez par exemple traiter un fichier de données généré par un autre logiciel pour en extraire les informations qui vous intéressent, les enregistrer dans un fichier `.csv`, puis les ouvrir ensuite avec un tableur.

2 OUVERTURE, FERMETURE ET PARCOURS D'UN FICHIER TEXTE

L'ouverture d'un fichier se fait avec la commande `open`.
Qui dit ouverture, dit parfois *création* lorsque le fichier n'existe pas encore.

Bien sûr, tout ceci nécessitera d'avoir les bons droits d'accès au fichier en lecture et/ou en écriture comme nous l'avons vu en début d'année.

Par exemple, pour créer un fichier[1] `test.txt` dans le dossier `C:/MonDossier/` :

```
#création du fichier et ouverture en mode écriture.
f=open("C:/MonDossier/test.txt","w")
f.close()                           #fermeture

f=open("C:/MonDossier/test.txt",mode="w") #synonyme de la première instruction.
f.close()                           #fermeture
```

Python 1: Ouverture/Fermeture d'un fichier

- Le premier argument correspond au chemin d'accès au fichier. Notez bien l'utilisation des "/" et non des "\" pour indiquer les dossiers, quelque soit l'OS.
- Le deuxième argument indique le mode d'ouverture. Les principaux modes possibles sont expliqués dans le tableau 1.
- on peut aussi rajouter un argument pour spécifier l'encodage du texte (voir la section 4), et quelques autres options...

mode	comportement
r	Ouvre en mode lecture seule. Le flux est positionné au début du fichier.
r+	Comme **r**, mais permet aussi l'écriture.
w	Crée un nouveau fichier et l'ouvre en écriture. S'il existait déjà un fichier du même nom, il l'écrase.
x	Comme **w**, mais renvoie une erreur s'il existe déjà un fichier du même nom. Évite d'écraser un fichier par mégarde[2].
w+	Comme **w**, mais permet aussi la lecture.
a	Ouvre le fichier en écriture, crée le fichier s'il n'existe pas. Le flux est positionné en **fin** de fichier. L'écriture est toujours ajoutée en fin de fichier.
a+	Comme **a**, mais permet aussi la lecture.

Table 1: Principaux modes pour l'ouverture d'un fichier en mode texte.

[1] S'il n'existe pas encore
[2] Cette fonctionnalité n'existe que depuis Python 3.3. Notez qu'un fichier écrasé est définitivement perdu, il n'est pas envoyé à la corbeille.

Attention : Un fichier est considéré comme un flux (*stream*) que l'on parcourt. Les opérations sont faites directement à l'endroit où l'on se trouve. Lorsque l'on ouvre, en fonction du mode choisi, on est soit au début, soit à la fin du fichier, et les commandes de lecture et d'écriture n'auront pas le même effet.
Si vous lisez le fichier alors que vous êtes placé au début, vous aurez l'intégralité du fichier et vous retrouverez à la fin.
Si vous voulez lire à nouveau, vous n'aurez plus rien devant vous : il faut déjà revenir au début pour recommencer à lire.

La métaphore du flux se veut assez parlante. Vous êtes comme une embarcation sur une rivière : vous n'avez pas la vision globale de la rivière, mais uniquement des quelques mètres devant vous.
Si vous voulez parcourir la rivière, il faut d'abord aller à la source et puis tout descendre. Tant que vous n'êtes pas arrivé au bout, vous ne savez pas du tout s'il vous reste beaucoup ou très peu à parcourir.
Une fois arrivé au bout, si vous voulez revoir toute la rivière, il faut déjà repartir à la source avant de recommencer la descente.
La méthode `seek` permet d'avancer à une certaine position. Avec la même métaphore, vous pouvez demander à aller directement au kilomètre 25 sur la rivière. Aller à la source, c'est revenir au kilomètre 0.

Voici quelques commandes que vous pouvez essayer (changez l'endroit du fichier `C:/MonDossier/` en fonction de votre arborescence personnelle). Ces scripts sont à écrire les uns à la suite des autres.

⚠ si vous chargez la bibliothèque `os`, alors il utilisera la fonction `open` de cette bibliothèque qui a un comportement différent de celui présenté ici.

```
#*****************************************************************************
# Crée un fichier et écrit un texte dedans. Si le fichier existe, il est écrasé.

f = open("C:/MonDossier/test.txt","w")
f.write("Python est le nom scientifique d'un genre de serpents ")
f.write("de la famille des Pythonidae.")
f.close()                              #fermeture

#*****************************************************************************
# Remplace les premiers caractères du fichier par le mot 'Ecrase'

f = open("C:/MonDossier/test.txt","r+") #flux au début du fichier.
f.write("Ecrase")
f.close()                              #fermeture

#*****************************************************************************
# Différents parcours de fichier avec lectures

f = open("C:/MonDossier/test.txt","r+") #flux au début du fichier.
f.read() #flux à la fin du fichier
f.read()
f.seek(0) #place le flux à l'indice 0, au début
contenu = f.read()
type(contenu)
print(contenu)
f.write("\nC'est la fin")

#*****************************************************************************
# Lit globalement par lignes

f.seek(0)
lignes = f.readlines() #tableau de toutes les lignes du fichier
f.close()
for s in lignes:
  print(s,end = "") #empêche print de mettre un entrée a la fin.

# résultat similaire
for s in open("C:/MonDossier/test.txt","r+"):
  print(s,end = "")
```

Python 2: Exemples de manipulations de fichiers

La commande `read` peut prendre un argument qui indique le nombre (maximum) de caractères à lire.
Le caractère "`\n`" utilisé avec la méthode `write` indique un passage à la ligne.

> **Un objet et ses méthodes**
> Vous observez que dans l'exemple précédent, on ne travaille pas directement sur le fichier, mais sur un objet obtenu par la fonction `open`. On a donné à cet objet le nom `f` qui n'a rien à voir avec le nom du fichier.
> Cet objet propose de nombreuses *méthodes* : qui sont comme des fonctions embarquées, ou attachées à cet objet.
> On fait appel à une méthode en faisant suivre l'objet d'un point "." et du nom de la méthode avec ses arguments.
> par exemple, pour appeler la méthode de fermeture du fichier, on écrit `f.close()`.
>
> Nous n'insistons pas là dessus, mais sachez que toute la programmation actuelle s'appuie sur ce concept : on parle de *programmation objet* : on crée des objets qui ont des attributs et des méthodes.
> Nous avons déjà utilisé certaines méthodes avec les listes par exemple. On dit que chaque liste que vous créez, est une *instanciation* de la classe "`list`". Elle en hérite les méthodes `append`, `remove`...

Exemple

Comptez le nombre de lignes d'un fichier.

```
#source est le nom du fichier avec le chemin d'accès.
fs = open(source, 'r')
nb = 0
for line in fs:
  nb += 1
print (nb) #écrit le nombre de lignes du fichier
```

Python 3: Nombre de lignes d'un fichier

Exemple

1) Programmez une fonction `copieFichier` qui copie la `source` vers la `destination`.

2) Programmez une fonction `copieFichier2` qui fait la même chose, mais en utilisant la méthode `readlines`.

3) Faites de même, une fonction `copieFichier3` avec la méthode `readline`.

Solution :

```
def copieFichier(source, destination):
  '''copie intégrale d'un fichier source vers un fichier destination'''
  fs = open(source, 'r')
  fd = open(destination, 'w')
  fd.write(fs.read())
  fs.close()
  fd.close()

def copieFichier2(source, destination):
  '''copie intégrale d'un fichier source vers un fichier destination
  ligne à ligne'''
  fs = open(source, 'r')
  fd = open(destination, 'w')
  for ligne in fs.readlines():
    fd.write(ligne)
  fs.close()
  fd.close()

def copieFichier3(source, destination):
  fs = open(source, 'r')
  fd = open(destination, 'w')
  test = True
  while test:
    ligne = fs.readline()
    if ligne == '':
      test = False
    else:
```

```
28          fd.write(ligne)
29      fs.close()
30      fd.close()
```

Python 4: Copie d'un fichier

3 Où suis-je ?

Principe du chemin dans l'arborescence :
Pour indiquer un emplacement, on utilise un chemin dans l'arborescence. Par exemple, pour trouver `C:/MonDossier/test.txt`, on va dans le disque `C:`, puis dans le dossier `MonDossier`, dans lequel on trouve le fichier `text.txt`.

Caractère / :
Windows utilise habituellement le caractère "\" pour séparer les étapes successives du chemin dans l'arborescence. Unix (et donc Mac, Linux...) utilisent le caractère "/". C'est ce dernier caractère qu'utilise Python, même si vous êtes sous Windows (car le caractère \ est un caractère d'échappement qui a une autre signification).

Chemin absolu versus chemin relatif :
Le **chemin absolu** est un chemin qui part de la *racine* de l'arborescence (par exemple le nom du disque sous Windows, ou simplement le dossier racine sous Unix), pour descendre jusqu'à l'emplacement souhaité.
Le **chemin relatif** est un chemin à partir du dossier de travail (par exemple le dossier Mes documents, ou le dossier home).
Pour donner une analogie géographique, donner le chemin absolu pour trouver une ville dans un pays, c'est par exemple donner le chemin pour y aller depuis la capitale. En revanche, si je sais que je suis à Saint Maur et si je souhaite aller à Nogent-sur-Marne, il peut être plus judicieux de donner la route à suivre depuis mon emplacement (Saint-Maur) et non depuis la capitale.
L'avantage du chemin absolu est qu'il est valable même si je ne sais pas bien où je suis (et j'arrive toujours à revenir facilement au dossier racine).
L'avantage du chemin relatif est que si je travaille uniquement dans un dossier (et ses sous-dossiers) et que je déplace tout ce dossier sur une nouvelle machine, mon chemin relatif restera valable alors que le chemin absolu devra être modifié.
En général, on privilégie le chemin relatif quand c'est possible.

Le dossier courant :
Utiliser le chemin relatif nécessite de savoir où l'on est. On l'obtient avec la commande `pwd` en Python.
Une fois que l'on connaît le dossier de travail de Python, on peut se déplacer dans l'arborescence facilement.

Se déplacer dans l'arborescence :
Le point "." indique le dossier courant (ou dossier de travail).
Les double points ".." indiquent que l'on monte au dossier parent.
Le slash en début de chemin "/" indique le dossier racine.
Le tilde " ~ " indique le dossier *home*.

Par exemple,

- pour aller dans le sous dossier `info` y chercher le document `text.txt`, on écrit le chemin `./info/text.txt`.
- pour aller remonter d'un dossier pour chercher le document `test.py`, on écrit le chemin `../test.py`.

La bibliothèque **os** dans python permet de faire des manipulations comme dans le shell (sous réserve de disposer des droits).

4 Encodage

Nous n'insistons pas particulièrement sur l'encodage ici, car les programmes que vous rédigerez seront généralement réservés à un usage très local.
L'encodage correspond à la manière de coder chaque caractère en "langage machine", et cela diffère en fonction des besoins des utilisateurs. Ainsi, les francophones ont besoin d'accents, les russes utilisent l'alphabet cyrillique...
Cela fait qu'au cours de l'histoire (informatique), en fonction des besoins des utilisateurs, différentes façon de coder les

caractères ont été utilisées. Votre OS utilise un encodage par défaut, et on s'arrange souvent pour que les logiciels en héritent tous. C'est la raison pour laquelle, Python devrait utiliser le même encodage que tous les logiciels sur un même PC (et plus généralement dans une région) et que vous n'aurez pas de problèmes à ce niveau.
Mais si vous récupérez des fichiers de l'extérieur, le problème peut arriver et il faut y penser. Il vous est sûrement déjà arrivé d'ouvrir un fichier ou un mail où les lettres accentuées devenaient des points d'interrogation ou des caractères bizarres. C'est simplement qu'il était ouvert avec le mauvais encodage (même si la plupart des logiciels intègrent des systèmes pour deviner le bon encodage).
Lors de l'ouverture du fichier, en cas de doute, vous pourrez donc préciser l'encodage du fichier pour éviter des erreurs. Mais cela demande bien sûr de savoir quel est cet encodage !
De même, pour indiquer à Python dans quel encodage est écrit le fichier source, il est conseillé d'écrire en première ligne du fichier qui contient votre programme (ou en deuxième) :

- si vous travaillez en encodage Latin-1 aussi appelé ISO-8859-1 (en général Windows) :

```
1 # -*- coding:Latin-1 -*-
```

- si vous travaillez en utf-8

```
1 # -*- coding:Utf-8 -*-
```

Si vous voulez en savoir plus, n'hésitez pas à lire l'excellent article (très simple) :
http://www.joelonsoftware.com/articles/Unicode.html traduit ici.