w3-s1-c1-fichiers

October 18, 2018

1 Les fichiers

1.1 Complément - niveau basique

Voici quelques utilisations habituelles du type fichier en Python.

1.1.1 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. On a vu aussi qu'il est recommandé de **toujours** utiliser l'instruction `with` et de contrôler son encodage. Il est donc recommandé de faire :

```
In [1]: # avec un `with' on garantit la fermeture du fichier
        with open("foo.txt", "w", encoding='utf-8') as sortie:
            for i in range(2):
                sortie.write(f"{i}\n")
```

1.1.2 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 en fin de fichier.

Voici par exemple comment on pourrait ajouter deux lignes de texte dans le fichier `foo.txt` qui contient, à ce stade du notebook, deux entiers :

```
In [2]: # on ouvre le fichier en mode 'a' comme append (= ajouter)
        with open("foo.txt", "a", encoding='utf-8') as sortie:
            for i in range(100, 102):
                sortie.write(f"{i}\n")
```

```
In [3]: # maintenant on regarde ce que contient le fichier
        with open("foo.txt", encoding='utf-8') as entree: # remarquez que sans 'mode', on ouvr
            for line in entree:
                # line contient déjà un retour à la ligne
                print(line, end='')
```

```
0
1
100
101
```

Il existe de nombreuses variantes au mode d'ouverture, pour par exemple : * ouvrir le fichier en lecture *et* en écriture (mode +) ; * ouvrir le fichier en mode binaire (mode `b`).

Ces variantes sont décrites dans la section sur la fonction built-in `open` dans la documentation Python.

1.2 Complément - niveau intermédiaire

1.2.1 Un fichier est un itérateur

Nous reparlerons des notions d'itérable et d'itérateur dans les semaines suivantes. Pour l'instant, on peut dire qu'un fichier - qui donc **est itérable** puisqu'on peut le lire par une boucle `for` - est aussi **son propre itérateur**. Cela implique que l'on ne peut le parcourir qu'une fois dans une boucle `for`. Pour le reparcourir, il faut le fermer et l'ouvrir de nouveau.

```
In [ ]: # un fichier est son propre itérateur
```

```
In [ ]: with open("foo.txt", encoding='utf-8') as entree:
            print(entree.__iter__() is entree)
```

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

```
In [4]: # Si l'on essaie d'écrire deux boucles imbriquées
        # sur le même objet fichier, le résultat est inattendu
        with open("foo.txt", encoding='utf-8') as entree:
            for l1 in entree:
                # on enlève les fins de ligne
                l1 = l1.strip()
                for l2 in entree:
                    # on enlève les fins de ligne
                    l2 = l2.strip()
                    print(l1, "x", l2)
```

```
0 x 1
0 x 100
0 x 101
```

1.3 Complément - niveau avancé

1.3.1 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.

Digression - `repr()` Comme nous allons utiliser maintenant des outils d'assez bas niveau pour lire du texte, pour examiner ce texte nous allons utiliser la fonction `repr()`, et voici pourquoi :

```
In [5]: # construisons à la main une chaîne qui contient deux lignes
        lines = "abc" + "\n" + "def"  + "\n"

In [6]: # si on l'imprime on voit bien les retours à la ligne
        # d'ailleurs on sait qu'il n'est pas utile
        # d'ajouter un retour à la ligne à la fin
        print(lines, end="")
```

```
abc
def
```

```
In [7]: # vérifions que repr() nous permet de bien
        # voir le contenu de cette chaine
        print(repr(lines))
```

```
'abc\ndef\n'
```

Lire un contenu - bas niveau Revenons aux fichiers ; la méthode `read()` permet de lire dans le fichier un buffer d'une certaine taille :

```
In [8]: # read() retourne TOUT le contenu
        # ne pas utiliser avec de très gros fichiers bien sûr

        # une autre façon de montrer tout le contenu du fichier
        with open("foo.txt", encoding='utf-8') as entree:
            full_contents = entree.read()
            print(f"Contenu complet\n{full_contents}", end="")
```

```
Contenu complet
0
1
100
101
```

```
In [9]: # lire dans le fichier deux blocs de quatre caractères
        with open("foo.txt", encoding='utf-8') as entree:
            for bloc in range(2):
                print(f"Bloc {bloc} >>{repr(entree.read(4))}<<")
```

```
Bloc 0 >>'0\n1\n'<<
Bloc 1 >>'100\n'<<
```

On voit donc que chaque bloc contient bien quatre caractères en comptant les sauts de ligne :

bloc #	contenu
0	un `0`, un *newline*, un `1`, un *newline*
1	un `1`, deux `0`, un *newline*

La méthode `flush` Les entrées-sorties 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`.

1.3.2 Fichiers textuels et fichiers binaires

De la même façon que le langage propose les deux types `str` et `bytes`, il est possible d'ouvrir un fichier en mode *textuel* ou en mode *binaire*.

Les fichiers que nous avons vus jusqu'ici étaient ouverts en mode *textuel* (c'est le défaut), et c'est pourquoi nous avons interagi avec eux avec des objets de type `str` :

```
In [10]: # un fichier ouvert en mode textuel nous donne des str
         with open('foo.txt', encoding='utf-8') as input:
             for line in input:
                 print("on a lu un objet de type", type(line))

on a lu un objet de type <class 'str'>
on a lu un objet de type <class 'str'>
on a lu un objet de type <class 'str'>
on a lu un objet de type <class 'str'>
```

Lorsque ce n'est pas le comportement souhaité, on peut : * ouvrir le fichier en mode *binaire* - pour cela on ajoute le caractère `b` au mode d'ouverture ; * et on peut alors interagir avec le fichier avec des objets de type `bytes`

Pour illustrer ce trait, nous allons : 0. créer un fichier en mode texte, et y insérer du texte en UTF-8 ; 0. relire le fichier en mode binaire, et retrouver le codage des différents caractères.

```
In [11]: # phase 1 : on écrit un fichier avec du texte en UTF-8
         # on ouvre donc le fichier en mode texte
         # en toute rigueur il faut préciser l'encodage,
         # si on ne le fait pas il sera déterminé
         # à partir de vos réglages système
         with open('strbytes', 'w', encoding='utf-8') as output:
             output.write("déjà l'été\n")

In [12]: # phase 2: on ouvre le fichier en mode binaire
         with open('strbytes', 'rb') as rawinput:
             # on lit tout le contenu
             octets = rawinput.read()
```

```
            # qui est de type bytes
            print("on a lu un objet de type", type(octets))
            # si on regarde chaque octet un par un
            for i, octet in enumerate(octets):
                print(f"{i}  {repr(chr(octet))} [{hex(octet)}]")
```

```
on a lu un objet de type <class 'bytes'>
0  'd' [0x64]
1  'Ã' [0xc3]
2  '©' [0xa9]
3  'j' [0x6a]
4  'Ã' [0xc3]
5  '\xa0' [0xa0]
6  ' ' [0x20]
7  'l' [0x6c]
8  "'" [0x27]
9  'Ã' [0xc3]
10  '©' [0xa9]
11  't' [0x74]
12  'Ã' [0xc3]
13  '©' [0xa9]
14  '\n' [0xa]
```

Vous retrouvez ainsi le fait que l'unique caractère Unicode `é` a été encodé par UTF-8 sous la forme de deux octets de code hexadécimal `0xc3` et `0xa9`.

Vous pouvez également consulter ce site qui visualise l'encodage UTF-8, avec notre séquence d'entrée :

https://mothereff.in/utf-8#d%C3%A9j%C3%A0%20l%27%C3%A9t%C3%A9%0A

```
In [ ]: # on peut comparer le nombre d'octets et le nombre de caractères
        with open('strbytes', encoding='utf-8') as textfile:
            print(f"en mode texte, {len(textfile.read())} caractères")
        with open('strbytes', 'rb') as binfile:
            print(f"en mode binaire, {len(binfile.read())} octets")
```

Ce qui correspond au fait que nos quatre caractères non-ASCII (3 x `é` et 1 x `à`) sont tous encodés par UTF-8 comme deux octets, comme vous pouvez vous en assurer ici pour `é` et là pour `à`.

1.3.3 Pour en savoir plus

Pour une description exhaustive vous pouvez vous reporter : * au glossaire sur la notion de `object file`, * et aussi et surtout au module `io` qui décrit plus en détail les fonctionnalités disponibles.