Python - Fichiers

[Mise à jour le : 12/7/2021]

- **Sources**
  - **Documentation** sur Python.org : référence du langage, lecture et écriture de fichiers, formats de fichier, fonctions natives (built-in)
- **Lectures connexes**
  - **Real Python**
    1. Python Interview Problem – Parsing CSV Files
    2. Reading and Writing Files in Python (Guide)
- **Mots-clés** : itérateur, ouverture, lecture, écriture, fermeture, module pickle, context manager.

Les mots ci-dessous sont dits “réservés”. Ils ne peuvent pas être utilisés comme nom de variable. Les mots soulignés sont une nouveauté de Python 3. Les mots en **gras** sont utilisés dans cette page.

and	continue	finally	is	raise
as	def	for	lambda	return
assert	del	from	None	True
async	elif	global	nonlocal	try
await	else	if	not	while
break	except	import	or	**with**
class	False	in	pass	yield

- Fonctions natives **(built-in)**[1] utilisées dans les exemples : **enumerate**(), **flush**(), **open**(), **print**(), **ranges**(), **repr**().
- **Modules** utilisées dans les exemples : **pathlib**, **pickle** et **json**[2]

---

1. Généralités

Un fichier est un **itérateur**. Il est donc **itérable** puisqu'on peut le **lire par une boucle for**. Mais il 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. Ecrire deux boucles for imbriquées sur le même objet fichier ne fonctionnerait pas.

2. Ouvrir un fichier

On utilise la fonction **open**() pour ouvrir un fichier. open renvoie un **objet fichier**.

*.py

```
f = open(chemin\nomfichier, mode, encodage) # f : objet fichier
                                            # l'encodage ne concerne
que les fichiers textuels
```

Cette fonction prend en paramètres :

- le **chemin** (absolu ou relatif) menant au fichier visé *nomfichier*;
- le **mode** d'ouverture
  - **'r'** : ouverture en lecture (**read**) par défaut (si non spécifié)
  - **'w'** : ouverture en écriture (**write**). Le contenu est écrasé. Si le fichier n'existe pas, il est créé.
  - **'a'** : ouverture en écriture en mode ajout (**append**). On écrit à la fin du fichier sans rien écraser.
- l'**encodage** à privilégier est **utf-8**

Par défaut, les fichiers sont ouverts en mode texte. On peut ajouter **b** à **r**, **w** et **a** pour spécifier que le fichier est ouvert en mode **binaire**. On ne doit spécifier l'encodage que pour les fichiers en mode texte.

Exemple : ouverture d'un fichier **en lecture**.

*.py

```
# La fonction open crée un objet de la classe TextIoWrapper.
# On utilise les méthodes de cette classe pour interagir avec le
fichier.
# Lors d'une ouverture en lecture, le chemin et le fichier doivent
exister
# sinon une erreur FileNotFoundError est générée.
f = open(r'data\fichier.txt','r',encoding='utf-8') # on transforme la
chaîne en raw string en
                                                   # plaçant un r devant
pour désactiver tous les '\'
```

Exemple : ouverture d'un fichier **en écriture**.

*.py

```
f = open(r'data\fichier.txt','w',encoding='utf-8') # f : objet fichier
```

Si un fichier **à ouvrir en écriture** n'existe pas, il est automatiquement créé.

3. Fermer un fichier

Un fichier ouvert doit être fermé après son utilisation pour qu'une autre application puisse y accéder. Pour cela, on **doit** utiliser la fonction **close()** ou ouvrir et fermer le fichier avec **with** (cette dernière méthode est à privilégier voir plus loin).

Exemple

.py

```
f.close()
```

4. Ouvrir et fermer automatiquement un fichier avec with

with permet d'ouvrir un fichier en s'assurant qu'il sera automatiquement fermé après sont utilisation qu'une exception se produise ou non. **Cette méthode est a privilégier**.

Exemple

*.py

```
# fichier.txt doit être présent dans le répertoire data sinon une erreur FileNotFoundError est générée.
with open(r'data\fichier.txt','r',encoding='utf-8') as f:
    pass
```

5. Ecrire dans un fichier

On utilise la méthode **write** en lui passant en paramètre la chaîne à écrire dans le fichier. Elle renvoie le nombre de caractères qui ont été écrits.

*.py

```
f.write(chaine) # écrit le contenu de chaine dans le fichier et renvoie le nombre de caractères écrits.
```

Exemples

*.py

```
# Avec un `with' on garantit la fermeture du fichier
# Si fichier.txt n'est pas présent dans le dossier data, il est créé
# Ouverture du fichier en écriture (utf-8)
with open(r'data\fichier.txt','w',encoding='utf-8') as f:
    for i in range(2):
        sortie.write(f"{i}\n") # Résultat : le fichier contient 0 1, un
caractère par ligne

# Ouverture en écriture en mode ajout
# Ajout des valeurs 2 à 9
with open(r'data\foo.txt', 'a', encoding='utf-8') as sortie:
    for i in range(2,10):
        sortie.write(f"{i}\n") # Résultat le fichier contient 0 1 2 3 4
5 6 7 8 9, un caractère par ligne
```

La méthode **write** accepte uniquement des chaînes de caractères. Il faudra donc convertir les autres types en chaîne avant de les écrire dans un fichier texte.

6. Lire le contenu d'un fichier

6.1 Lire l'intégralité d'un fichier

On utilise la méthode **read**() de la classe **TexteIoWrapper** pour lire l'intégralité d'un fichier.

Exemple 1a : ouverture avec **with**, on ne gère pas l'exception *FileNotFoundError* (**à privilégier**)

*.py

```
# foo.txt doit être présent dans le répertoire data sinon une erreur
FileNotFoundError est générée.
# foo.txt contient le texte 0 1 2 3 4 5 6 7 8 9, un caractère par ligne
with open(r'data\foo.txt','r',encoding='utf-8') as f:
    contenu = f.read() # La totalité du fichier est placée dans une
chaîne de caractères
print(contenu)         # résultat : 0 1 2 3 4 5 6 7 8 9, un caractère
par ligne
```

Exemple 1b : ouverture avec **with**, on gère l'exception *FileNotFoundError* (**à privilégier**)

.py

```
# foo.txt doit être présent dans le répertoire data sinon une erreur FileNotFoundError est générée.
# foo.txt contient le texte 0 1 2 3 4 5 6 7 8 9, un caractère par ligne
try:
    with open(r'data\foo.txt', 'r', encoding='utf-8') as f:  # on essaie d'ouvrir le fichier
        contenu = f.read()  # le fichier a été ouvert, on lit le contenu et
    print(contenu)          # résultat : 0 1 2 3 4 5 6 7 8 9, un caractère par ligne
except FileNotFoundError:
    print("Le fichier n'existe pas")
```

Exemple 1c : ouverture avec **open**(), fermeture avec **close**(), on ne gère pas l'exception *FileNotFoundError*

*.py

```
# foo.txt doit être présent dans le répertoire data sinon une erreur FileNotFoundError est générée.
# foo.txt contient le texte 0 1 2 3 4 5 6 7 8 9, un caractère par ligne
f = open(r'data\foo.txt','r',encoding='utf-8')
contenu = f.read() # La totalité du fichier est placée dans une chaîne de caractères
print(contenu)     # résultat : 0 1 2 3 4 5 6 7 8 9, un caractère par ligne
f.close()
```

Exemple 1d : ouverture avec **open**(), fermeture avec **close**(), on gère l'exception *FileNotFoundError*

.py

```
# foo.txt doit être présent dans le répertoire data sinon une erreur FileNotFoundError est générée.
# foo.txt contient le texte 0 1 2 3 4 5 6 7 8 9, un caractère par ligne
try:
    f = open(r'data\foo.txt','r',encoding='utf-8') # on essaie d'ouvrir le fichier
except  FileNotFoundError:
    print("Le fichier n'existe pas")
else:
    contenu = f.read() # le fichier a été ouvert, on lit le contenu et
    print(contenu) # résultat : 0 1 2 3 4 5 6 7 8 9, un caractère par ligne
    f.close()
```

Après ouverture et transfert de son contenu dans une variable de type str, le contenu du fichier peut être traité avec les méthodes de la classe string. Voir “Les séquences - Chaînes de caractères”.

6.2 Lecture du fichier ligne par ligne

readline lit une seule ligne du fichier; un caractère de fin de ligne (**\n**) doit être présent à la fin de la chaîne.

> **\n** n’est omis que sur la dernière ligne du fichier. Si f.readline() renvoie une chaîne vide, c’est que la fin du fichier a été atteinte, alors qu’une ligne vide est représentée par '\n' (une chaîne de caractères ne contenant qu’une fin de ligne).

Exemple

*.py

```
# fichier.txt doit être présent dans le répertoire data sinon une erreur FileNotFoundError est générée.
with open(r'data\fichier.txt','r',encoding='utf-8') as f:
    a = f.readline() # Contenu de fichier.txt
    b = f.readline() # Première ligne
print(a,b)
```

Les fichiers étant des **itérateurs**, ils peuvent être placés dans une boucle **for**. Pour les lire **ligne à ligne**, on peut aussi boucler sur l’objet fichier. C’est plus efficace en termes de gestion mémoire, plus rapide et le code est plus simple.

Exemple

*.py

```
# fichier.txt doit être présent dans le répertoire data sinon une erreur FileNotFoundError est générée.
with open(r'data\fichier.txt','r',encoding='utf-8') as f:
    for line in f:
        print(line, end='')
```

Résultat attendu

> # Contenu de fichier.txt
> Première ligne

> Deuxième ligne
> Troisième ligne

Pour construire une **liste** avec toutes les lignes d'un fichier, il est aussi possible d'utiliser ***list(f)*** ou ***f.readlines()***.

7. Fichier binaire

A faire

8. Enregistrer un objet dans un fichier

- **Problème**

Les données dans un programme Python sont stockées en mémoire (la RAM), sous une forme propice aux calculs. Par exemple un petit entier est fréquemment stocké en binaire dans un mot de 64 bits, qui est prêt à être soumis au processeur pour faire une opération arithmétique.

Ce format ne se prête pas forcément toujours à être transposé tel quel lorsqu'on doit écrire des données sur un support plus pérenne, comme un disque dur, ou encore sur un réseau pour transmission distante

Ainsi par exemple il pourra être plus commode d'écrire notre entier sur disque, ou de le transmettre à un programme distant, sous une forme décimale qui sera plus lisible.

Il convient donc de faire de la traduction dans les deux sens entre représentations d'une part en mémoire, et d'autre part sur disque ou sur réseau (à nouveau, on utilise en général les mêmes formats pour ces deux usages). Les plus utilisés sont **json**, **csv**, **xml** et **pickle**.

Dès que l'on souhaite enregistrer ou transmettre des types de données complexes comme des listes, des dictionnaires ou des instances de classes, le traitement précédent devient vite compliqué. Plutôt que de passer son temps à écrire et déboguer du code permettant de sauvegarder des types de données compliqués, on peut utiliser les modules **pickle** et **json** (JavaScript Object Notation) fournis par Python.

Le module *pickle* est à privilégier si l'on travaille uniquement en Python. Le format JSON étant normalisé, le module *json* est adapté à la sauvegarde de données devant être exploitées par d'autres langages.

Exemple 1 : sauvegarde d'un dictionnaire dans un fichier binaire avec le module *pickle*

- **source** : Pickle in Python: Object Serialization

*.py

```
import pickle

# Création d'un dictionnaire
```

```
scoreW = {
    "joueur 1" : 5,
    "joueur 2" : 10,
    "joueur 3" : 2
}

# Enregistrement
with open(r'data\donnees.pk','wb') as f: # l'extension .pk est
optionnelle, ici choisie arbitrairement
    pck = pickle.Pickler(f) # Création de l'objet Pickler
    pck.dump(scoreW) # Enregistrement

# Lecture de l'objet enregistré
with open(r'data\donnees.pk','rb') as f:
    unpck = pickle.Unpickler(f) # Création de l'objet Unpickler
    scoreR = unpck.load() # Enregistrement

# Affichage du dictionnaire relu après enregistrement
print(scoreR)  # résultat : {'joueur 1': 5, 'joueur 2': 10, 'joueur 3':
2}
               # On retrouve sa structure !
```

Exemple 2 : sauvegarde d'un dictionnaire dans un fichier texte avec le module *json*

- **source** : Working With JSON Data in Python

*.py

```
import json

# Création d'un dictionnaire
scoreW = {
    "joueur 1": 5,
    "joueur 2": 10,
    "joueur 3": 2
}

# Enregistrement
with open(r'data\donnees.json', 'w') as f:
    json.dump(scoreW, f)

# Lecture de l'objet enregistré
with open(r'data\donnees.json', 'r') as f:
    scoreR = json.load(f)

# Affichage du dictionnaire relu après enregistrement
print(scoreR)  # résultat : {'joueur 1': 5, 'joueur 2': 10, 'joueur 3':
2}
               # On retrouve sa structure !
```

Exemple 3 : limite de json pour python

*.py

```
import json

# En partant d'une donnée construite à partir de types de base
data = [
    # des types qui ne posent pas de problème
    [1, 2, 'a', [3.23, 4.32], {'eric': 32, 'jean': 43}],
    # un tuple
    (1, 2, 3),
]

# sauver ceci dans un fichier
with open("s1.json","w", encoding='utf-8') as json_output:
    json.dump(data, json_output)

# et relire le résultat
with open("s1.json", encoding='utf-8') as json_input:
    data2 = json.load(json_input)
    print(data2) # Résultat : [[1, 2, 'a', [3.23, 4.32], {'eric': 32,
'jean': 43}], [1, 2, 3]]
                 # le tuple (1,2,3) s'est transformé en liste [1,2,3]
```

Certains types de base ne sont pas supportés par le format JSON (car ils ne sont pas natifs en JavaScript), c'est le cas notamment pour :
- **tuple**, qui se fait encoder comme une liste ;
- **complex**, **set** et **frozenset**, que l'on ne peut pas encoder du tout (sans étendre la bibliothèque).

9. Les built-in repr() et flush()

- **repr()** permet d'afficher les caractères spéciaux dans une chaîne.

Exemple 1

*.py

```
lines = "abc" + "\n" + "def"  + "\n"
print("abc" + "\n" + "def"  + "\n") # Résultat
                                    # abc
                                    # def
print(repr(lines))                  # Résultat 'abc\ndef\n'
```

Exemple 2 : ouverture d'un fichier texte en binaire pour afficher les codes des caractères

*.py

```
with open(r'data\foo.txt', 'rb') as bytesfile:
    # on lit tout le contenu
    octets = bytesfile.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)}]") # Résultat
partiel
                                                          # 0 → '0'
[0x30]
                                                          # 1 → '\r'
[0xd]
                                                          # 2 → '\n'
[0xa]
```

- **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*.

10. Utilitaires

Depuis la version 3.4, en remplacement de os.path, la bibliothèque pathlib fournit dans une interface orientée objet unique des utilitaires pour **parcourir l'arborescence de répertoires et fichiers**.

Résumé

- On ouvre un fichier en utilisant la fonction **open** prenant en paramètre le chemin vers le fichier, le mode d'ouverture et l'encodage si s'est un fichier texte.
- On lit dans un fichier en utilisant la méthode **read** et on écrit avec la méthode **write**.
- Un fichier doit être refermé après usage avec la méthode **close** ou automatiquement avec la syntaxe de **with**. Cette deuxième méthode est recommandée en Python moderne.
- Le module **pickle** est utilisé pour enregistrer des objets Python dans des fichiers et les recharger ensuite.

Quiz

- Reading and Writing CSV Files in Python
- Reading and Writing Files in Python

Pour aller plus loin ...

- Context Managers and Python's with Statement
- Python Practice Problems: Parsing CSV Files
- Working With JSON Data in Python
- Python 3's pathlib Module: Taming the File System
- Python mmap: Improved File I/O With Memory Mapping

1)
Fonctions toujours disponibles.
2)
JavaScript Object Notation