

MANIPULATION DE FICHIERS

1. Le principe

But. Jusqu'à présent, dans les programmes Python étudiés, il n'était pas possible de sauvegarder des données afin de les réutiliser. La manipulation de fichiers va être un moyen d'y parvenir.

Type de fichier. Nous nous contenterons ici d'utiliser des fichiers texte tels qu'on peut les écrire avec un éditeur de texte basique (tels que le bloc-notes Windows, *Notepad++* ou *Emacs*).

Types de manipulations. Nos fichiers pourront être utilisés *en lecture* ou *en écriture*.

Nous effectuerons le plus souvent ces opérations *séquentiellement* c'est-à-dire en parcourant le fichier ligne par ligne. Ce chapitre n'est pas exhaustif, seules sont présentées les méthodes les plus simples utilisées en particulier en TP.

2. Ouverture et fermeture d'un fichier

La manipulation d'un fichier passe par 3 étapes :

1. l'ouverture du fichier,
2. la lecture ou l'écriture dans ce fichier,
3. la fermeture de ce fichier.

Les étapes d'ouverture et de fermeture sont essentielles à toute manipulation de fichier. Examinons-les d'un peu plus près.

2.1 Ouverture

On ouvre un fichier avec la fonction `open` prenant en paramètres :

- le nom du fichier, ou plus précisément le chemin menant au fichier.
- le mode d'ouverture :
 - `'r'` : ouverture en lecture seule (read)
 - `'w'` : ouverture en écriture (write)
 - `'a'` : ouverture en ajout (cf infra pour l'intérêt)

Exemple. Le résultat donné par la commande `open` (résultat d'un type bien particulier) sera au passage stocké dans une variable Python :

```
premier_fichier = open("fichier1.txt", 'r')
second_fichier = open("dossier/fichier2.txt", 'w')
```

Remarques. Attention sous pyzo, il faut compiler avec *Exécuter le script* (Ctrl-Maj-E) et non *Exécuter le fichier* (Maj-E), sinon les chemins ne sont pas reconnus !

Évidemment, l'ouverture en lecture d'un fichier qui n'existe pas pose problème :

```
>>> file = open("ce_fichier_n_existe_pas", 'r')
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    file = open("ce_fichier_n_existe_pas", 'r')
FileNotFoundError: [Errno 2] No such file or ✓
    directory: 'ce_fichier_n_existe_pas'
```

En revanche l'ouverture en écriture crée le fichier (vide) s'il n'existe pas et **l'écrase** s'il existait au préalable! Il faut donc être prudent avec l'option **'w'**. L'ouverture en mode ajout permet de limiter ce risque : dans ce cas l'écriture éventuelle ne détruit pas le fichier initial mais se fait à la fin.

2.2 Fermeture

La fermeture doit s'effectuer systématiquement lorsqu'on n'a plus besoin d'accéder au fichier. On utilise pour cela la méthode `close`.

La fermeture d'un fichier est une étape essentielle pour terminer proprement la manipulation du fichier. En effet s'il reste ouvert, cela bloquera tout autre accès au fichier (message d'erreur classique : "Cette action ne peut pas être réalisée car le fichier est déjà utilisé par une autre application").

Remarque. Pour cette raison, il est conseillé d'écrire l'instruction `close` juste après l'instruction `open`.

Les instructions de lecture ou d'écriture proprement dites seront insérées entre les lignes correspondantes

Exemple. L'exemple de code précédent se complète alors en :

```
premier_fichier = open("fichier1.txt", 'r')
#instructions de lecture à insérer
premier_fichier.close()

second_fichier = open("dossier/fichier2.txt", 'w')
#instructions d'écriture à insérer
second_fichier.close()
```

Remarques.

- Le nom choisi pour la variable Python représentant le fichier peut être quelconque (à condition de suivre les restrictions concernant les noms de variable).
- Il est possible d'avoir plusieurs fichiers ouverts simultanément.

3. Lecture séquentielle d'un fichier

3.1 Principe général

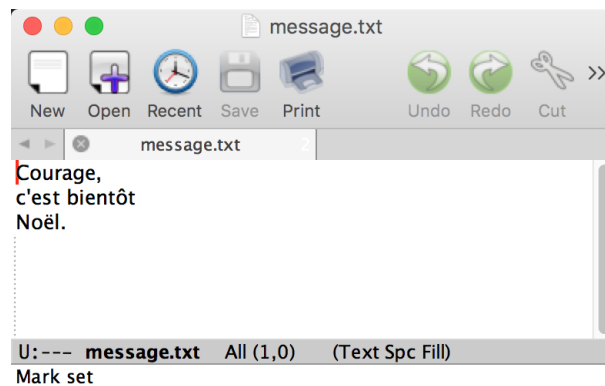
Les fichiers texte possèdent une organisation naturelle par ligne. Python permet d'exploiter cette organisation au moyen d'une simple boucle.

```
mon_fichier = open('le_nom_du_fichier', 'r')

for ligne in mon_fichier:
    #instructions à effectuer sur la ligne
```

```
mon_fichier.close()
```

On illustre le propos sur le fichier `message.txt` suivant :



Ce fichier est constitué de trois lignes de texte que l'on peut afficher dans le shell à l'aide du programme suivant :

```
mon_fichier = open('message.txt', 'r')

for ligne in mon_fichier:
    print(ligne)

mon_fichier.close()
```

Son execution retourne :

```
>>>
Courage,

c'est bientôt

Noël.

>>> |
```

Remarquez le saut de ligne dans le shell.

Remarque. Vous pourrez aussi rencontrer `mon_fichier.readline()` qui retourne une ligne du fichier texte et passe à la ligne suivante. Le programme

```
mon_fichier = open('message.txt', 'r')
a = mon_fichier.readline()
b = mon_fichier.readline()
c = mon_fichier.readline()
print(c)
```

affiche :

```
>>>
Noël.

>>>
```

3.2 Type de la réponse.

La variable `ligne` est une **chaîne de caractères** contenant à chaque itération une nouvelle ligne du fichier texte.

Si l'on souhaite récupérer des nombres, il faut procéder à un nettoyage de chaîne et une conversion comme présenté au chapitre précédent.

4. Écriture dans un fichier

Un fichier ouvert en écriture (mode `'w'`) peut être modifié au moyen de la méthode `write()` qui prend en paramètre la **chaîne de caractères** que l'on souhaite écrire dans le fichier.

Remarques.

- Un fichier ouvert en mode `'w'` sera totalement écrasé au moment de son ouverture.
- L'écriture physique dans le fichier sera en fait effectuée au moment de la fermeture du fichier : une autre raison de ne pas l'oublier !

Exemples. Écriture d'un fichier `message.txt` :

```
mon_fichier = open('message.txt', 'w')
mon_fichier.write("Vive la prépa ! ")
mon_fichier.close()
```

Recopie ligne par ligne d'un fichier (aucun intérêt réel...) :

```
mon_fichier = open('message.txt', 'r')
ma_copie = open('copie.txt', 'w')

for ligne in mon_fichier:
    ma_copie.write(ligne)

mon_fichier.close()
ma_copie.close()
```

5. Exemples de manipulation de fichier

Calculs numériques

On considère un fichier texte dont chaque ligne est constituée de plusieurs valeurs numériques séparées par des virgules.

On veut appliquer une fonction `calcule(liste_de_nombres)` supposée prédéfinie à chaque ligne.

Le fichier de données étant un fichier texte, une phase de conversion est nécessaire avant tout traitement.

```
def traitement(nom_fichier):
    """ traite le fichier <nom_fichier>
        placé dans le répertoire courant
        <nom_fichier> : type str, avec l'extension
    """
```

```
resultats = []

fichier = open(nom_fichier, 'r')

for ligne in fichier: # parcours par ligne

    ## conversion ligne -> liste de flottants
    #####
    ligne_splittee = ligne.split(',')
    # ligne_splittee est une liste de chaînes
    liste_valeurs = []
    for chaine in ligne_splittee:
        # conversion en flottant
        val = float(chaine)
        liste_valeurs.append(val)

    ## calculs sur la liste de flottants
    #####
    res = calcule(liste_valeurs)
    #résultat par ligne
    resultats.append(res)
    # <resultats> contient autant de termes que ✓
    de lignes dans le fichier

fichier.close()

return resultats
```

Représentation graphique

On imagine ici que chaque ligne est constituée d'une abscisse et une ordonnée séparées par un point-virgule.

On veut faire une représentation graphique à partir du fichier.

```
def representation(nom_fichier)
    """ graphique à partir du fichier <nom_fichier> ✓
        placé dans le répertoire courant
        <nom_fichier> : type str, avec l'extension
    """

    abscisses = []
    ordonnees = []

    fichier = open(nom_fichier, 'r')

    for ligne in fichier: # parcours par ligne

        ## récupération abscisse et ordonnée
```

```
#####
ligne_splittee = ligne.split(';')
# ligne_splittee est une liste de 2 chaînes
# on les convertit en flottants
# puis on les adjoint aux listes

absc = float(ligne_splittee[0])
ordo = float(ligne_splittee[1])

abscisses.append(absc)
ordonnees.append(ordo)

fichier.close() # plus besoin du fichier ouvert

## tracé proprement dit
#####

# on suppose l'importation
# import matplotlib.pyplot as plt déjà faite
plt.plot(abscisses, ordonnees, label = 'mon joli ✓
graphique', ...)
...
... # autres commandes graphiques

plt.show()
# l'affichage peut être *sorti* de la ✓
fonction pour plus de modularité
# par exemple pour permettre de juxtaposer ✓
plusieurs graphiques sur une même figure.

return None
```

6. Le cas des fichiers csv

6.1 Présentation

Le format **csv** (comma-separated values) donne un fichier texte avec une structure tabulaire. C'est le format générique pour le traitement statistique de données. Par exemple, le fichier **intensite_tension.csv** contient une série de mesures d'intensités et de tensions électriques. Ses trois premières lignes sont :

```
intensité (/A) ; tension (/V)
0.0031349331371112;6.724317540481
0.006873315009560;14.213994387046
```

On désire stocker les valeurs numériques des intensités et des tensions dans deux listes et rajouter la valeur de la résistance $R = U/I$ dans une troisième colonne d'un nouveau

fichier intensite_tension_resistance.csv.

6.2 Avec readline() et split()

```
# On ouvre le fichier de données en lecture
fichier_entree = open('intensite_tension.csv','r')
# On ouvre le fichier de résultats en écriture ('w').
# Pour l'instant le fichier est vide.
fichier_sortie = open('intensite_tension_resistance.csv', 'w')

# On lit l'entête du fichier de données:
entete = fichier_entree.readline()
# On nettoie la chaîne pour enlever le saut de ligne
entete = entete.strip()

# On complète l'entête (noter la présence du saut de
# ligne en fin de chaîne):
entete = entete + ' ; Resistance (/Ohm) \n'
# On peut alors écrire l'entête du fichier de sortie.
fichier_sortie.write(entete)

# On parcourt le fichier de données ligne à ligne ✓
# après avoir lu l'entête du fichier
for ligne in fichier_entree :
    # On récupère les données d'une ligne et on les
    # convertit en flottant:
    data = ligne.split(';')
    U,I = float(data[1]), float(data[0])

    # On peut alors calculer la donnée à rajouter...
    R = U/I
    # ... et créer la chaîne de caractère à écrire
    # dans le fichier de sortie
    # (noter la présence du saut de ligne \n)
    out = str(I) + ';' + str(U) + ';' + str(R) + '\n'
    fichier_sortie.write(out)

## On ferme les 2 fichiers
fichier_entree.close()
fichier_sortie.close()
```

Voici les premières lignes du fichier obtenu :

```
intensité (/A) ; tension (/V) ; Resistance (/Ohm)
0.0031349331371112 ; 6.724317540481 ; 2144.963623268
0.006873315009560 ; 14.213994387046 ; 2067.996937035
```

6.3 Avec la librairie csv

Python dispose d'une librairie dédiée au traitement des fichiers **csv** dont les fonctions essentielles sont **reader** et **writer**. Elles permettent d'obtenir directement des listes de nombre sans avoir à procéder au traitement ligne par ligne du fichier.

```
import csv

file = open('intensites_tensions.csv','r')
file2 = open('intensites_tensions2.csv','w')

# Lecture et écriture de la première ligne
entete=file.readline()
entete = entete.strip()
entete = entete + ' ; Résistance (/Ohm)'

# Les commandes de lecture et d'écriture
data = csv.reader(file,delimiter=";")
writer = csv.writer(file2,delimiter=";")

# Écriture de l'entête
writer.writerow([entete])

# Lecture et traitement des lignes suivantes de file
for ligne in data :
    # on récupère les chaînes de caractère
    I,U = ligne[0], ligne[1]
    # Calcul de R
    R = float(U) / float(I)
    # Ecriture dans file2
    writer.writerow([I,U,R])

# fermeture des fichiers
file.close()
file2.close()
```