# TP n° 11 : Fichiers

Le programme encourage explicitement l'utilisation de fichiers pour stocker durablement des données sous différents formats. L'objectif de ce TP est de découvrir et d'expérimenter la gestion des fichiers sous PYTHON. Dans ce TP, on s'intéresse à la lecture d'un fichier texte et à l'écriture de données textuelles, mais nous verrons plus tard que l'on peut aussi traiter de nombreux types de données : audio, vidéo, images, etc.

## 1 Le module `os`

La bibliothèque `os` propose une interface avec le système d'exploitation qui se veut portable, c'est-à-dire qui devrait fonctionner sous la plupart des systèmes (LINUX, WINDOWS, MAC OS, etc.).

PYTHON
```
import os
```

1. Que font les commandes suivantes ? Ne pas hésiter à vérifier à chaque étape avec `os.getcwd()` et/ou `os.listdir()`.
   - `mydir = os.getcwd()` .....
   - `os.listdir()` .....
   - `os.mkdir("testdir")` .....
   - `os.path.exists("testdir")` .....
   - `os.path.exists(os.path.join(mydir, "testdir"))` .....
   - `os.chdir("testdir")` .....
   - `os.getcwd()` .....
   - `os.chdir(mydir)` .....
   - `os.rename("testdir", "dirtest")` .....
   - `os.path.exists("testdir")` .....
   - `os.rmdir("dirtest")` .....

## 2 Le module `shutil`

Le module `shutil` propose de nombreuses fonctions permettant de manipuler et de copier des fichiers et des répertoires. Par exemple, pour copier dans le répertoire courant le répertoire `donnees` qui se trouve dans le répertoire `/home/classes/mpsi3/informatique/tp11/`, on peut écrire :

PYTHON
```
import shutil

classe_dir = "/home/classes/mpsi3"
donnees_dir = "informatique/tp11/donnees"
source_dir = os.path.join(classe_dir, donnees_dir)
dest_dir = os.path.join(os.getcwd(), "donnees")
shutil.copytree(source_dir, dest_dir)
```

1. Exécuter ce programme et vérifier (avec PYTHON !) que le répertoire a bien été copié dans votre répertoire de travail.

## 3 Gestion des fichiers

On peut assez facilement lire ou écrire dans des fichiers en PYTHON. Pour cela, il faut au préalable créer un *objet-fichier* du fichier dont on connaît le nom (sous la forme d'une chaîne de caractères), avec un mode qui peut être :

- `'r'` pour *read* : fichier ouvert en mode lecture ;
- `'w'` pour *write* : fichier ouvert en mode écriture, écrase l'ancien fichier ou en crée un nouveau ;
- `'a'` pour *append* : fichier ouvert en mode écriture, ajout à la fin du fichier.

Par exemple, pour créer un premier fichier vide :

PYTHON
```
fichier = open("mon_fichier.txt", 'w')
```

Une fois le traitement terminé, il est impératif de refermer l'objet-fichier.

PYTHON
```
fichier.close()
```

2. Créer un fichier vide. Vérifier, avec `os.listdir` ou `os.path.exists` que cela a bien fonctionné.

Il existe une syntaxe particulière qui permet de garantir que l'objet-fichier sera bien fermé, même si une erreur survient avant que l'on ait pu le fermer explicitement :

PYTHON
```
with open(nom_du_fichier, mode) as objet_fichier:
    # Bloc dans lequel
    # le fichier est
    # ouvert
# Ici, l'objet-fichier n'est plus défini
```

Par exemple, écrivons quelques lignes dans le fichier que nous avons créé précédemment, en écrasant son ancien contenu (qui était vide, ce n'est donc pas bien grave) :

PYTHON
```
with open("mon_fichier.txt", "w") as fichier:
    fichier.write("Première ligne")
    fichier.write("... et toujours la même\n")
    fichier.write("Deuxième ligne\n")
fichier.write("Fin\n")  # oops !
```

`fichier.write(chaine)` ajoute les caractères de l'argument à la suite de ce qui a déjà été ajouté (donc sans espaces, saut de ligne, etc.). Rappelons qu'un saut de ligne dans une chaîne de caractères s'obtient avec le caractère spécial `'\n'` (c'est *un* caractère). Pour une tabulation, on peut utiliser le caractère spécial `'\t'`.

Notons que par défaut le répertoire de travail est celui dans lequel le script PYTHON se trouve, mais on peut le modifier avec `os.chdir`.

Pour lire dans un objet-fichier, on peut utiliser :

- `fichier.read()` renvoie la chaîne de tous les caractères du fichier et place le curseur à la toute fin. Lors de l'ouverture d'un fichier le curseur est placé au tout début du fichier. On ne peut donc pas directement lire deux fois le même objet-fichier. Avec un argument entier, `fichier.read(n)` renvoie les `n` caractères suivant la position du curseur et place celui-ci juste après ;
- `fichier.readline()` renvoie la ligne courante à partir du curseur, sous forme d'une chaîne de caractères, et déplace celui-ci au début de la ligne suivante. Cela renvoie une chaîne vide si on atteint la fin du fichier ;
- `fichier.readlines()` renvoie la liste de toutes les lignes (bien pratique pour faire des boucles) ;
- on peut aussi parcourir chaque ligne sans avoir à créer une liste en écrivant directement `for ligne in fichier`. Magique !

Par exemple, le programme suivant permet d'imprimer à l'écran le contenu du fichier `"mon_fichier.txt"`.

PYTHON
```
with open("mon_fichier.txt", "r") as fichier:
    for line in fichier:
        print(line, end='')
```

3. Afficher de même le contenu du fichier en utilisant les méthodes `.read`, `.readline`, `.readlines`, en utilisant la syntaxe avec **with** au moins une fois et la syntaxe avec ouverture et fermeture explicite au moins une fois.
4. Ajouter une troisième ligne de votre choix dans ce fichier, en utilisant le mode `"a"`. Vérifier avec une des méthodes de la question précédente.

### 3.1 Les méthodes `split`, `join` et `strip`

Chaque ligne sauf éventuellement [1] la dernière se termine par le caractère fin de ligne `'\n'`.

5. Si `ligne` est une chaîne de caractères qui se termine par le caractère `'\n'`, comment obtenir une copie de cette ligne sans ce dernier caractère en utilisant des tranches ? .....

On peut aussi écrire `ligne = ligne.strip()`. L'aide en ligne, disponible par exemple en écrivant `str.strip?`, explique ce que réalise cette méthode. En PYTHON, le mot *whitespace* désigne tous les caractères considérés comme des « blancs » :

- `' '` (espace)
- `'\n'` (retour à la ligne)
- `'\r'` (retour chariot)
- `'\t'` (tabulation horizontale)
- `'\f'` (saut de page)
- `'\v'` (tabulation verticale)

1. Mais c'est une mauvaise pratique, même la dernière ligne devrait se terminer par un retour à la ligne.

Dans la suite du TP, on utilisera sans hésiter les méthodes `str.split`, `str.join` et `str.strip`. L'aide en ligne est disponible si vous ne vous souvenez plus ce que réalisent exactement ces fonctions.

## 4 Applications

### 4.1 Admissibles aux mines

Le fichier `mines.tsv` qui se trouve dans le répertoire `donnees/mines` contient les résultats des admissibilités des candidats au concours MINES-PONTS 2015. Il est issu d'un PDF disponible sur le site du concours [2].

Dans ce fichier, sur chaque ligne figurent les champs : n° de candidat, nom et prénom, résultat, série d'oral si admissible. Chaque champ est séparé par une tabulation `'\t '`. On appelle ce format *Tab-separated values* d'où l'extention.

Le fichier étant issu d'un fichier PDF, il a été nettoyé, mais il reste les numéros des pages initiales sur certaines lignes.

6. Afficher, avec PYTHON, le contenu de ce fichier.
7. Écrire une fonction `nettoyage()` créant un fichier dans le même répertoire nommé `mines_propre.tsv` dans lequel ne figurent plus ces numéros de ligne.
8. Écrire une fonction `nb_admissibles()` renvoyant le nombre d'admissibles.
9. Écrire une fonction `separe_series()` créant, toujours dans le bon répertoire, un fichier pour chaque série d'oral : `serie1.tsv`, `serie2.tsv`, `serie3.tsv` et `serie4.tsv`. Chaque fichier contenant le numéro et l'identité des candidats de la série correspondante, au format *Tab-separated values*.

### 4.2 Cent mille milliards de poèmes

En 1961, l'écrivain et poète Raymond QUENEAU proposa un recueil de sonnets potentiels à composer par le lecteur. Un sonnet est composé de 14 vers —

2. https://mines-ponts.fr/

ici des alexandrins — séparés en deux groupes de 4 vers (les quatrains avec une succession de rimes ABAB) et deux groupes de 3 vers (les tercets de rime AAB et CCB). Pour chaque vers QUENEAU propose 10 possibilités. Ces possibilités sont données dans les fichiers `q1v1.txt` (premier quatrain, premier vers), `q1v2.txt` (premier quatrain, deuxième vers), ..., `t2v3` (deuxième tercet, troisième vers), qui se trouvent dans le répertoire `donnees/queneau`.

Par exemple, le premier poème est :

```
Le roi de la pampa retourne sa chemise
pour la mettre à sécher aux cornes des taureaux
le cornédbîf en boîte empeste la remise
et fermentent de même et les cuirs et les peaux

Je me souviens encor de cette heure exeuquise
les gauchos dans la plaine agitaient leurs drapeaux
nous avions aussi froid que nus sur la banquise
lorsque pour nous distraire y plantions nos tréteaux

Du pôle à Rosario fait une belle trotte
aventures on eut qui s'y pique s'y frotte
lorsqu'on boit du maté l'on devient argentin

L'Amérique du Sud séduit les équivoques
exaltent l'espagnol les oreilles baroques
si la cloche se tait et son terlintintin
```

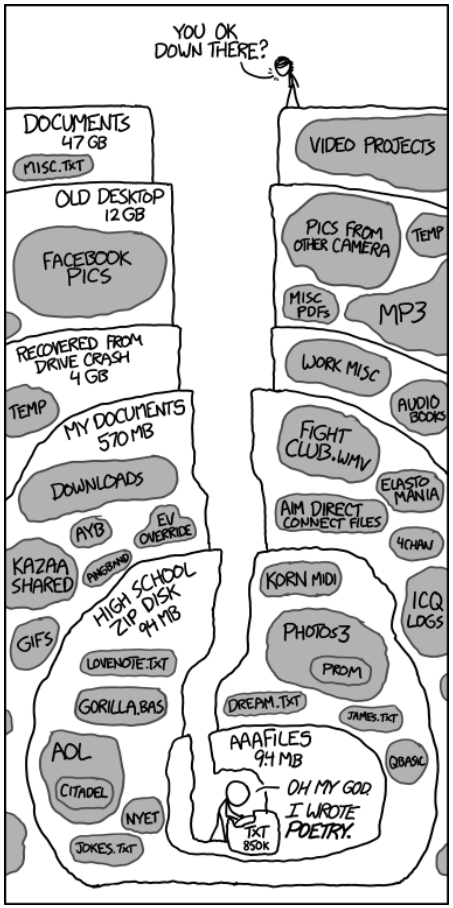
1. Écrire une fonction `premier_poeme()` qui affiche ce premier poème à partir des fichiers sources des vers.
2. Écrire une fonction `compose_poeme()` qui affiche une possibilité de poème choisie aléatoirement parmi les $10^{14}$ poèmes possibles[3]. On utilisera la fonction `random.randint` du module `random`. On pourra commencer par écrire le premier quatrain.

3. QUENEAU ajoute : « En comptant 45 s pour lire un sonnet et 15 s pour changer les volets à 8 heures par jour, 200 jours par an, on a pour plus d'un million de siècles de lecture, et en lisant toute la journée 365 jours par an, pour 190 258 751 années plus quelques plombes et broquilles (sans tenir compte des années bissextiles et autres détails) ».

## 4.3 Nombre de langages informatiques

Le fichier `liste_langages.txt` dans le répertoire `donnees/langages` est tiré de la page correspondante de Wikipedia.

1. Donner le nombre de langages de programmation répertoriés dans ce fichier.




https://xkcd.com/1360/