

# TP 25

## Fichiers texte

### I Quelques notions sur le système de fichiers

#### I.1 L'arborescence des fichiers

Les fichiers sur l'ordinateur permettent de stocker des informations et de les récupérer quand on en a besoin. Dans un programme Python, il est courant de vouloir accéder à un fichier pour en lire le contenu, ou bien pour écrire dedans et sauvegarder des données.

Ils ont tous un nom et sont rangés dans des dossiers. Chaque dossier peut donc contenir plusieurs fichiers mais aussi d'autres dossiers. Cela se représente naturellement par un arbre. Chaque fichier possède un **chemin d'accès** dans lequel le caractère / indique que ce qui suit se trouve dans le dossier de ce qui précède. Le chemin d'accès est en général donné relativement à un dossier de base dans lequel on travaille.

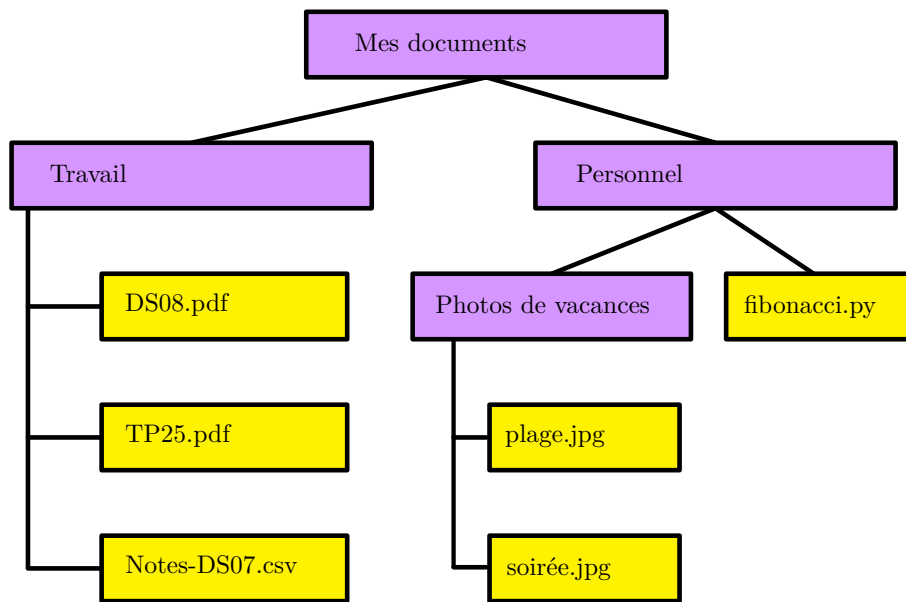


Figure 1. – Ici, travaillant à partir du dossier `Mes documents`, le fichier `plage.jpg` est à l'adresse relative `Personnel/Photos de vacances/plage.jpg`. Le dossier `Travail` contient trois fichiers ; le dossier `Personnel` contient un dossier (contenant lui-même deux fichiers) et un fichier.

#### Avertissement

Pour bien démarrer le TP, il est impératif de bien vérifier que le programme Python comprenne, que les chemins d'accès au fichier sont donnés relativement au dossier dans lequel se trouve le TP. Ce n'est malheureusement pas automatique.

Le dossier est en général bien configuré après avoir exécuté une première fois l'option sous Pyzo « Démarrer le script ». Sinon le clic droit côté interactif puis « Définir le répertoire courant en accord avec avec le fichier ouvert dans l'éditeur ». La commande en mode interactif `cd` permet d'afficher le dossier dans lequel le programme pense être ; la commande `ls` permet d'afficher le contenu du dossier. Sous Spyder ou Jupyter, tester aussi la commande `pwd` (*print working directory*).

Si les petits tests dans le programme s'exécutent correctement, le TP peut démarrer.

#### I.2 Les droits d'accès

Les fichiers sur les systèmes Linux et leurs dérivés ont tous des **droits**, qui déterminent qui peut accéder au fichier et comment. Ces droits ont des noms :

- **r** (*read*) : le droit de lire le contenu du fichier.
- **w** (*write*) : le droit d'écrire du contenu dans le fichier.
- **x** (*execute*) : quand le fichier est un programme, le droit d'exécuter le programme.

Chaque fichier possède, ou non, indépendamment chacune de ces 3 autorisations, soit un total de  $2^3 = 8$  combinaisons possibles. Par exemple si un fichier possède le droit de lecture mais pas d'écriture, il est impossible d'en modifier le contenu.

En fait, Linux étant à la base même un système multi-utilisateurs, chacune de ces trois autorisations peut être donnée indépendamment pour l'utilisateur, ou son groupe d'utilisateurs, ou pour tous les autres utilisateurs, soit un total de  $8^3 = 512$  combinaisons. Sur les ordinateurs de la salle, un clic droit sur un fichier puis sur « Propriétés » permet d'afficher les droits du fichier.

### I.3 Le processus de lecture et écriture de fichier

Bien qu'un fichier texte soit constitué, comme son nom l'indique, de texte, on ne peut pas le lire exactement comme les chaînes de caractères en accédant à tout moment au  $i$ -ème caractère. Les problèmes suivants se posent :

- Le fichier peut être stocké sur le disque dur, ou sur une clé USB, ou sur un ordinateur auquel on accède à travers le réseau ; dans tous ces cas chaque accès au fichier demande un certain effort au matériel, et il n'est pas raisonnable d'écrire une boucle qui demande à lire chaque caractère un par un. Cela serait beaucoup trop lent et userait le matériel.
- On ne peut pas non plus toujours le stocker en mémoire d'un seul coup avant de le lire. Penser par exemple à une vidéo YouTube de plusieurs heures, dont le fichier original occupe plusieurs giga-octets : on n'a pas besoin de la télécharger en entier, mais pendant qu'on lit un morceau alors la suite se télécharge en arrière-plan.

En fait, lors de la lecture d'un fichier, un mécanisme automatique et largement caché à l'utilisateur se met en place et le fichier est chargé en mémoire par blocs, comme dans l'exemple décrit de vidéo. Le terme technique est *mise en mémoire tampon* (en anglais *buffer*), de la mémoire temporaire pour stocker un bout de fichier.

En écriture aussi, il serait très inefficace de demander au disque dur d'écrire uns par uns les caractères. Les données à écrire sont chargées dans une « file d'attente » et celle-ci est automatiquement vidée quand elle est pleine. Le terme technique est *flush* comme dans... tirer la chasse d'eau. On peut retenir que l'utilisateur n'a pas toujours la garantie que sa commande d'écriture a effectivement bien écrit les données sur le disque dur, même si elle s'est correctement terminée.

### I.4 Ouverture et fermeture

Pour lire ou écrire dans un fichier, il faut donc **l'ouvrir** ce qui a pour effet notamment de préparer ce processus de mise en mémoire tampon.

À n'importe quel moment le système d'exploitation sait si un fichier est ouvert quelque part ou non. C'est par exemple cela qui va provoquer des avertissements ou des erreurs si on débranche une clé USB alors qu'un fichier dessus est encore considéré comme ouvert... Cela sert aussi à interdire à un autre programme de modifier le fichier pendant qu'on est en train de le lire (ce serait embêtant !).

Le contraire est donc la **fermeture** du fichier. Cela a notamment pour effet de vider complètement la mémoire tampon, et donc d'écrire effectivement sur le disque et de sauvegarder le fichier. Après la fermeture seulement, on peut être confiant que le programme a bien réussi et que le fichier contient bien ce qu'on y a écrit.

## II Les fichiers en Python

### II.1 L'ouverture

La fonction `open(nom, mode)` sert à ouvrir un fichier en donnant son nom ou son chemin d'accès. Elle renvoie une variable de type `file` qui représente un fichier ouvert. La fonction prend deux arguments : le nom du fichier (une chaîne de caractères), et le mode d'ouverture. Les modes possibles sont listés ci-dessous :

- **"r"** (*read*) : Lecture seule. Permet de lire du contenu dans le fichier.
- **"w"** (*write*) : Écriture dans le fichier. Si le fichier n'existe pas, il est créé comme un fichier vide. Sinon, il est d'abord effacé, pour être remplacé par ce qu'on va écrire. **Ce mode est dangereux.**
- **"a"** (*append*) : Écriture dans le fichier mais, s'il existe déjà, tout ce qui va être écrit va s'ajouter à la fin du fichier.

Une fois que le fichier a été ouvert avec `f = open(nom, mode)`, la méthode `f.close()` ferme le fichier. Il n'est alors plus possible de lire ou d'écrire dedans.

### Avertissement

Certains modes sont **dangereux** car ils peuvent **effacer un fichier**. Avant d'utiliser des fichiers en écriture, vérifiez bien :

- Que le fichier n'existe pas déjà, ou bien qu'il existe mais que vous êtes sûr de vouloir l'effacer,
- Que le code Python a le bon chemin d'accès au fichier,
- Que la variable représentant le fichier représente bien le fichier qu'on veut.

*Remarque.* Il est important d'utiliser un mode qui soit en accord avec les droits du fichier. Typiquement de nombreux fichiers de configuration du système sont en lecture seule (pour l'utilisateur, mais pas pour l'administrateur) et l'ouverture en mode écriture échouera, même si on ne demande pas à écrire dedans. On ouvre donc dans le mode qui donne *le moins* de droits possible par rapport à ses besoins.

### Exercice 1

Tester et observer le fichier de démarrage. Exécutez-le plusieurs fois et observez les fichiers produits.

## II.2 La lecture

Une fois qu'un fichier a été ouvert en mode lecture `"r"` et représenté par une variable `f`, les méthodes suivantes sont disponibles. Chacune déplace une sorte de « curseur », de sorte que les lectures suivantes continuent à avancer dans le fichier.

- `f.read()` : lit **tout** le fichier d'un seul coup et renvoie **une** chaîne de caractères.

```
>>> f = open("test.txt", "r")
>>> f.read()
'Première ligne du fichier de test.\nDeuxième ligne.\nTroisième et dernière ligne.\n'
```

Observez les caractères `"\n"` : ceux-ci indiquent un saut de ligne (`n` pour *new line*). Les sauts de lignes sont des caractères à part entière, tout comme les espaces ou la ponctuation.

- `f.readline()` : lit **une ligne** du fichier. Renvoie donc une variable de type chaîne de caractère.

```
>>> f = open("test.txt", "r")
>>> f.readline()
'Première ligne du fichier de test.\n'
>>> f.readline()
'Deuxième ligne.\n'
```

Le caractère `"\n"` est encore présent à la fin de la ligne, et appeler la fonction plusieurs fois continue d'avancer dans le fichier.

- `f.readlines()` : lit **toutes les lignes** du fichier, jusqu'à la fin. Renvoie une **liste de chaînes de caractères**, chaque élément de la liste représentant une ligne du fichier.

```
>>> f = open("test.txt", "r")
>>> f.readlines()
['Première ligne du fichier de test.\n', 'Deuxième ligne.\n', 'Troisième et dernière\nligne.\n']
```

- `for s in f` : boucle qui fournit les **lignes unes par unes**, toujours avec leur caractère de fin de ligne.

```
>>> for s in f: s
'Première ligne du fichier de test.\n'
'Deuxième ligne.\n'
'Troisième et dernière ligne.\n'
```

- `f.seek(0)` : remet le curseur au début du fichier, on peut alors le relire depuis le début.

La présence du caractère de fin de ligne est souvent ennuyeuse, de même que la présence éventuelle de caractères espaces en trop en fin de ligne : cela empêchera de faire des comparaisons exactes entre chaînes de caractères ! La méthode `s.strip()` renvoie une nouvelle chaîne sans le `"\n"` final, et supprime les espaces en trop en début et en fin de ligne, pour ne garder que le contenu intéressant :

```
>>> s = "      une ligne avec trop d'espaces  \n"
>>> s
'      une ligne avec trop d'espaces  \n'
>>> s.strip()
'une ligne avec trop d'espaces'
```

En résumé, dans ces exercices on traitera les fichiers de la façon suivante :

```
f = open(nom, "r")
for s in f:
    ...
    ... s.strip() ...
    ...
...
```

### Exercice 2

Le fichier joint `noms.txt` contient la liste des prénoms des élèves de la classe ; chaque prénom est sur une seule ligne.

1. D'abord, écrire un programme qui ouvre et ferme le fichier et contient une boucle qui affiche uns par uns les noms.
2. Améliorer le programme pour qu'il affiche `Bonjour [nom] et bienvenue en BCPST1B !` (où `[nom]` sera remplacé par le nom de l'élève).

### Exercice 3

Le fichier joint `dictionnaire.txt` contient un dictionnaire de plus de 600 000 mots pour la correction orthographique.

Écrire une fonction `cherche(m)` qui prend en argument un mot `m` et qui renvoie `True` si le mot `m` apparaît dans le fichier, `False` sinon.

### Exercice 4

Le fichier joint `pi.txt` contient un million de décimales du nombre  $\pi$ , formatées en lignes de 100 décimales.

Écrire un programme qui compte combien de fois chaque chiffre apparaît, avec une liste `L` de taille 10 initialement remplie de zéros, où `L[i]` contiendra le nombre de fois que le chiffre `i` apparaît dans les décimales.

Attention, étant donné un caractère `x` de la chaîne (qui représente bien un nombre, mais est de type `str`), on a besoin de la fonction `int(x)` pour le convertir en un nombre.

## II.3 Écriture

Une fois qu'un fichier a été ouvert dans un mode permettant l'écriture, et représenté par une variable `f`, il y a deux façons principales d'écrire dedans :

- `f.write(s)` : écrit la chaîne de caractères `s` dans le fichier. Pour sauter des lignes, il suffit d'écrire dans `s` des caractères `"\n"` : des appels successifs à `write` ne passent pas automatiquement à la ligne. Comme la fonction prend uniquement en argument une chaîne de caractère, si on veut écrire des nombres il faut d'abord formater les chaînes comme dans l'annexe III.1.

- `print(s, file=f)` : la bonne vieille fonction `print` à laquelle on dit d'écrire dans le fichier `f` plutôt que sur l'écran. On peut alors écrire tout ce qu'on écrit d'habitude avec `print` (des chaînes de caractères, des nombres, des listes, etc). Un autre argument optionnel `end=""` permet d'enlever le saut de ligne automatique à chaque appel de `print` (« remplacer le caractère `"\n"` automatiquement ajouté à la fin par la chaîne vide `" "` »).

*Remarque.* Pourquoi toutes ces complications ? Remarquez que `f.write(s)` est exactement l'opération inverse de `s = f.read()`, si on lit le fichier y compris avec ses sauts de lignes alors on le ré-écrit tel quel. La commande `print`, elle, a été conçue pour être très pratique pour être utilisée en mode interactif, mais pour écrire dans un fichier texte ce n'est pas toujours le comportement voulu : on veut en général formater les chaînes plus précisément.

### Exercice 5

Il est courant d'avoir un programme qui fait un calcul et de vouloir enregistrer le résultat pour ne pas le perdre. Choisissons par exemple, au hasard, un programme qui calcule les termes successifs d'une suite de récurrence d'ordre 2... Écrire une fonction `fibonacci(n)` qui ouvre un fichier nommé `fibonacci.txt` pour écriture, et écrit dedans un par un les termes de la suite de Fibonacci.

À la fin on doit obtenir un fichier qui ressemble à ceci :

```
0
1
1
2
3
5
8
13
...
```

N'hésitez pas à tester avec  $n = 100$  et vous pouvez enregistrer le fichier et le garder en souvenir !

### Exercice 6

On veut étudier des séries de 10 lancers de dés, et conserver les résultats obtenus. Écrire un programme qui ouvre un fichier nommé `dés.txt` en mode `"a"`, simule 10 fois de suite un lancer de dé, et écrit le résultat dans le fichier. Idéalement, les résultats sont séparés par un espace, et il faut ajouter une nouvelle ligne une seule fois à la fin.

Si on lance plusieurs fois le programme et que tout se passe bien, les séries de lancers s'accumulent. Le fichier devra ressembler à :

```
2 6 5 3 6 6 1 1 3 5
6 3 2 3 2 4 6 1 3 3
1 3 1 4 6 5 6 6 5 1
...
```

## II.4 Lire d'un côté et écrire de l'autre

**Texte simple** On propose la situation suivante. La procédure Parcoursup se termine et la direction du lycée Hoche reçoit la liste des élèves qui ont été admis et ont accepté de venir. Elle souhaite donc envoyer un message de bienvenue personnalisé à chacun. Mais il n'est pas question de copier-coller des messages, ni de tout ré-écrire à la main. À la place, on veut écrire un programme Python qui, d'un côté lit la liste des élèves, de l'autre produit des fichiers prêts à envoyer...

### Exercice 7

Écrire une fonction `bienvenue(n, nom)` qui prend en argument un nombre et un prénom, qui crée une variable contenant le nom du fichier sous la forme `bienvenue-[n].txt` (en remplaçant `[n]` par le nombre) puis ouvre ce fichier en mode `"w"` et écrit dedans un texte de bienvenue personnalisé. On pourra choisir librement le texte, par exemple

```
Bonjour [nom],  
Félicitations pour votre admission au lycée Hoche. Nous vous souhaitons la bienvenue dans  
la classe de BCPST1B et espérons que vous vous y plairez.  
Le Lycée Hoche
```

Pour tester la fonction il faut par exemple essayer `bienvenue(1, "L.-C. Lefèvre")` et observer si un fichier a bien été créé. Si c'est le cas on peut passer à la deuxième partie.

Comme plusieurs personnes de la classe peuvent porter le même prénom, on ne peut malheureusement pas utiliser directement le prénom pour créer le nom du fichier, c'est pour cela qu'il faudra les numéroté.

Idéalement, les fichiers produits devraient tous se trouver dans un même dossier `lettres` et les numéros devraient tous être alignés sur deux caractères (en complétant avec un zéro), voir l'annexe III.1. Mais on peut s'en passer pour l'instant.

### Exercice 8

Écrire une programme qui ouvre en lecture le fichier `noms.txt` contenant la liste de tous les élèves de la classe, qui itère sur les lignes pour récupérer les prénoms uns par uns (en comptant au passage les prénoms dans une variable `n`) et appelle la fonction précédente pour générer le fichier de bienvenue.

Puis lancer le programme et observer. Si tout se passe bien, de nombreux fichiers doivent être créés dans le dossier `lettres`.

**Générer du PDF** Pour produire un beau fichier PDF, il faudrait passer par un langage de programmation permettant de mettre en forme un document tout en donnant du contenu. C'est bien trop compliqué pour ce TP...

Un programme `lettres.py` est fourni dans le matériel du TP. Le programme Python écrit un fichier dans un autre langage appelé LaTeX, qui est très utilisé pour produire des documents scientifiques de haute qualité (les feuilles d'exercices, sujets de DS et DM, mais pas ce TP). Tester le programme en exécutant tout le fichier d'un coup, attendre, et admirer le résultat.

**Avec un fichier CSV** Une autre grosse source d'amélioration consiste à charger non pas une simple liste des élèves mais tout un fichier CSV contenant plus d'informations sur chaque élève. Un fichier CSV s'ouvre au départ comme un fichier texte `f = open(nom, mode)` mais ensuite il faut créer un objet `g` « lecteur » pour itérer dessus en fournissant unes à unes les lignes avec la syntaxe `for x in g`, et il y a deux façons naturelles de faire, en utilisant le module `csv` (à importer bien entendu) :

- `g = csv.reader(f)` : l'itération `for x in g` fournira unes par unes les entrées (lignes) comme des listes, ainsi `x[0]` sera la première colonne de l'entrée, `x[1]` la deuxième colonne, etc. Le fichier CSV n'a alors pas besoin de ligne d'en-tête, et s'il y en a une elle apparaît en tant qu'entrée comme les autres.
- `g = csv.DictReader(f)` : l'itération `for x in g` fournira des dictionnaires, dont les clés sont déterminées en utilisant la ligne d'en-tête (qui doit être présente) du fichier CSV. Avec le fichier fourni de liste des élèves, on aura donc accès à `x["nom"]`, `x["prenom"]`, etc.

On le traite de la façon suivante :

```
import csv  
f = open("liste.csv", "r")  
g = csv.DictReader(f)  
for x in g:  
    ...  
    ... x["nom"], x["prenom"], x["etablissement_origine"], ...  
    ...  
f.close()
```

### Exercice 9

Reprendre le programme précédent mais avec un fichier CSV :

- La fonction `bienvenue(n, ...)` prend par exemple en argument des noms, prénoms, établissements d'origine,
- Le programme utilise le fichier CSV de liste des élèves `liste.csv` (et lit les lignes comme des dictionnaires) et appelle la fonction `bienvenue(n, ...)`. Il écrit dans le dossier `lettres-CSV`.

### III Annexes

#### III.1 Le formatage des chaines de caractère

Le problème du formatage consiste à insérer dans une chaîne de caractère des mots, des nombres, et parfois de régler certains paramètres comme le nombre de caractères affichés ou le nombre de chiffres significatifs.

Une syntaxe moderne en Python consiste à précéder la chaîne de caractère directement par la lettre `f` (on parle de *f-string* : chaîne formatée). Ensuite, lorsque la chaîne va contenir la syntaxe `{nom:code}` la variable nommée `nom` va être remplacée par sa valeur, éventuellement formatées avec le code de formatage `code`. Quelques exemples :

```
>>> nom = "L.-C. Lefèvre"
>>> print(f"Bonjour {nom}")
Bonjour L.-C. Lefèvre
>>> x = 15
>>> print(f"x = {x:4d}")
x =    15
>>> print(f"x = {x:04d}")
x = 0015
```

Pour les nombres entiers, le code `d` permet de l'afficher comme un nombre décimal ; précédé d'un nombre, il indique la place en nombre de caractères que va occuper l'écriture du nombre. Précédé aussi d'un zéro, il force l'affichage d'un zéro pour remplir l'espace. On peut aussi précéder encore d'un `+` pour forcer l'affichage du symbole `+` pour les nombres positifs ; cela est pratique pour bien aligner en colonne des nombres positifs et négatifs : observer l'alignement

```
>>> for x in (-123, -8, 13, 256): print(f"{x:+04d}")
-123
-008
+013
+256
```

Pour les nombres à virgule flottante, il existe diverses options pour régler le nombre de chiffres significatifs, l'affichage en décimal ou bien en notation exponentielle, etc. C'est un mini-langage en soi. Se référer à une documentation.

#### III.2 Le mot-clé `with`

Lorsqu'on manipule des fichiers on trouve souvent le code suivant :

```
with open(nom, mode) as f:
    ...
    # bloc d'instructions
    # opérations sur le fichiers
    ...
...
```

La première ligne est à peu près équivalente à notre `f = open(nom, mode)`. Mais les différences avec la syntaxe précédente sont :

- Ce qui suit est un *bloc d'instructions*, exactement comme avec les `if`, les boucles, etc.
- Le fichier est automatiquement fermé à la fin du bloc d'instructions...
- ... même si une erreur pendant l'exécution se produit.

Au contraire sans le mot-clé `with`, si une erreur se produit après avoir ouvert le fichier alors le programme va s'arrêter et donc risque de ne pas aller jusqu'à l'instruction de fermeture du fichier !

On parle de **gestionnaire de contexte** — le mot-clé déclenche une sorte de veille sur le bon déroulement des opérations qui sont bien délimitées dans le bloc d'instructions. Il semble que le mot-clé `with` soit une bonne pratique recommandée en Python, mais pas adoptée en BCPST...

### III.3 Le module `os`

Le module `os` (*Operating System*, système d'exploitation) contient de nombreuses fonctions pour gérer tout ce que l'on pourrait faire manuellement en se baladant dans l'explorateur de fichiers, et d'obtenir des informations sur le système. Quelques exemples :

- `os.getcwd()` : donne le nom du dossier dans lequel on travaille.
- `os.listdir()` : renvoie la *liste* de tous les dossiers et fichiers situés dans le dossier dans lequel on travaille.
- `os.mkdir(nom)` : crée un dossier avec le nom donné.
- `os.remove(nom)` : supprime le fichier de nom donné.
- `os.rename(nom_avant, nom_après)` : renomme un fichier.

Pour faire des tests dans ce TP, cela peut être intéressant d'écrire dans le programme le code qui efface les fichiers créés, pour pouvoir facilement recommencer. On peut aussi vouloir écrire un programme qui traite automatiquement tous les fichiers dans un dossier donné.