Informatique Pour Tous

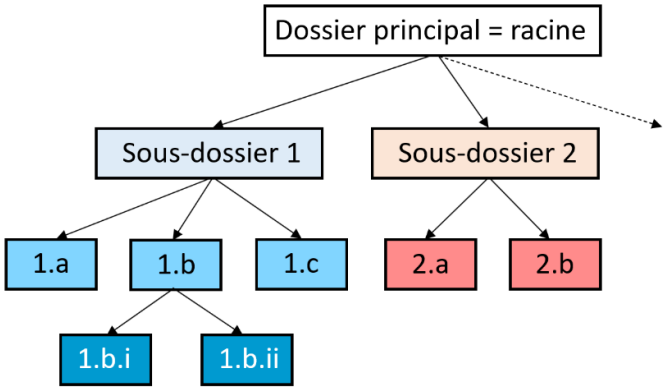
Cours 13 – Lecture et écriture des fichiers textes

Jusqu'ici, tous les scripts que nous avons écrits se contentaient d'afficher un résultat (que ce soit un résultat numérique ou un graphique). Mais dans certains cas, ceci n'est pas suffisant et les résultats doivent être enregistrés dans un fichier. Par exemple :
→ Si *Python* est utilisé pour communiquer avec d'autres logiciels, ou d'autres périphériques, alors l'export des données calculées par *Python* peut être nécessaire pour la communication des données.
→ Si l'algorithme utilisé est un algorithme itératif très long ou instable, il est intéressant de stocker les données à chaque itération. En effet, tant qu'elles ne sont pas sauvegardées sur la mémoire morte (ROM) de l'ordinateur, toutes les variables du script restent stockées sur la mémoire vive (RAM) de l'ordinateur, et sont donc supprimées et définitivement perdues en cas de défaillance de l'ordinateur.

A / Pré-requis

A.1 / Chemin d'accès vers un fichier

Sur les systèmes d'exploitation traditionnels (*Unix* – Windows ou Mac – , *Linux, GNU*) le rangement des dossiers sur la mémoire morte (ROM) est pensé sous la forme d'une **arborescence**, par exemple :



- Le dossier principal est appelé la « racine », c'est en général le nom du disque dur (ou de la clé USB, du périphérique mémoire …)
- Ce dossier se décompose en sous-dossiers, eux-mêmes décomposés en sous-sous-dossiers, etc. À un niveau quelconque de l'arborescence, le dossier du « dessus » est appelé le dossier parent et les dossiers du « dessous » sont appelés les dossiers enfants.
- Dans chaque dossier (y compris la racine) on peut trouver des fichiers. Chaque fichier a un nom et une extension (séparés par un point). Cette dernière indique au système d'exploitation le type du document et donc avec quel logiciel il doit l'ouvrir par défaut : exemples "*My_song.mp3*" est une bande son, "*Harry_potter.avi*" est une vidéo, "*devoir.pdf*" est un fichier pdf, etc. Dans un dossier donné, le nom et l'extension permettent de désigner un unique fichier.
- **Le chemin d'accès** d'un fichier permet de savoir où trouver un fichier spécifique. Il peut être indiqué *via*
  - **Le chemin absolu** (aussi appelé **adresse** sous *Windows*) permet de retracer tout le chemin permettant d'accéder au fichier depuis la racine. Exemple : "*C:\Users\Documents\fichier.txt*"
  - **Le chemin relatif** permet de se situer par rapport au dossier actuellement ouvert.

Ainsi, si le chemin absolu d'un fichier est "*C:\Users\Documents\fichier.txt*" alors son chemin relatif est :
- Si le dossier *Users* est ouvert : ".\Documents\fichier.txt"
- Si le dossier *Documents* est ouvert : ".\fichier.txt" ou encore plus simplement "*fichier.txt*".

Remarque: sous *Linux* ou *Mac* les séparateurs sont des " / " alors que sous *Windows* ce sont des " \ "

Allons plus loin : quand on écrit un script *Python* qui doit pouvoir être exécuté depuis n'importe quel ordinateur, alors :

- Il y a de grandes chances pour que les noms des dossiers ne soient pas exactement les mêmes d'un ordinateur à l'autre : suivant le système d'exploitation, les noms des différents dossiers, etc, l'arborescence change. On ne peut donc pas utiliser le chemin absolu ! De même, si vous utilisez des adresses absolues, mais que vous déplacez le script et les fichiers vers un autre dossier, il ne les retrouvera plus.

- En pratique, on va donc utiliser les chemins relatifs en rangeant le script *Python* dans le même dossier que les fichiers qu'il doit lire ou écrire.

A.2 / Ouverture et droits d'accès à un fichier

Lorsqu'on ouvre un fichier avec un logiciel, on peut accorder à ce logiciel des droits :

- **En lecture** (*read* ou lecture seule) : le logiciel peut lire le fichier mais ne peut pas l'éditer
- **En écriture** (*write* – ceci présuppose le droit en lecture, donc on parle aussi de droit en « lecture-écriture») :

  Le logiciel peut lire et éditer le fichier, donc le modifier

Remarque: *Python* ajoute un droit intermédiaire **ajout** (*add*) qui signifie qu'on peut ajouter des lignes au document, mais rien supprimer.

A.3 / Import et export de données sous forme de texte

En ingénierie, on travaille souvent avec des données sous forme de texte, car c'est leur forme la plus compacte en termes de stockage et de vitesse de lecture/écriture. Les objets communicants, comme un oscilloscope par exemple, exportent généralement leurs données sous un format ".csv" ou ".txt", et l'on trouve aussi parfois des ".dat".

Tous ces formats sont des formats de texte : les données sont rangées suivant le modèle d'une table, par exemple :

Temps (s)	Donnée 1 (binaire /8 bits)	Donnée 2 (signal en Volt)
0.00	56	- 2.15
0.01	79	- 1.05
0.02	137	- 0.41
0.03	174	+ 0.52
0.04	191	+ 1.46

Pour interpréter cette table comme texte, il faut fixer deux ou trois standards :

- Comment **sépare**-t-on les colonnes ?

Là, vous allez certainement vous dire : « Mais enfin, sur un fichier texte, quand je tape sur Entrée, ça va à la ligne tout seul, ça ne m'écrit pas un caractère… », mais en pratique c'est faux ! Un saut de ligne correspond à un code, c'est juste que les logiciels de traitement de texte ne vous affichent pas ce caractère, et vous représentent le saut de ligne graphiquement ! Mais au fond, un fichier texte, c'est une longue suite de caractère qui se suivent, c'est l'afficheur qui l'interprète au fur et à mesure que vous le tapez (**ex :** `print("Vive\tLa\nPTSI\tPT")` affiche : `Vive  La`
`PTSI  PT`)

Ainsi, quand vous allez « lire » un fichier texte avec *Python*, il va transformer l'intégralité du fichier texte en une seule variable, qui contient **l'intégralité du texte** comme une seule chaîne de caractères ! Il faut donc lui indiquer comment est codé le saut de ligne, pour qu'il puisse l'interpréter.

- On y revient donc : comment **sépare**-t-on les **lignes** ? Par convention, le **séparateur** est "\n".
- Comment **sépare**-t-on les **colonnes** d'une même ligne ? Par convention, les **séparateurs** de colonnes les plus classiques sont :

  un pt virgule ";" ou bien un espace " " ou bien une tabulation ("\t" en *Python*)
- On arrive alors au contenu d'une « case » (une ligne et une colonne donnée). Il reste alors à préciser son **type**.

  Exemple : dans la table précédente *Python* devra interpréter
  - Le format de la 1ère ligne comme : format *str*
  - Le format de la 1ère et de la dernière colonne : format *float*
  - Le format de la colonne du milieu : format *int*

Exemples : avec un séparateur de colonnes de type point-virgule, la table précédente ressemblerait à ceci :
"Temps;Donnée 1 ;Donnée2\n0.00 ;56 ;-2.15\n0.01;79;-1.05\n0.02;137;-0.41\n0.03;174;0.52\n0.04;191;1.46"

B / Découvrons au travers d'exemples

B.1 / Lecture d'un fichier avec un petit nombre de colonnes

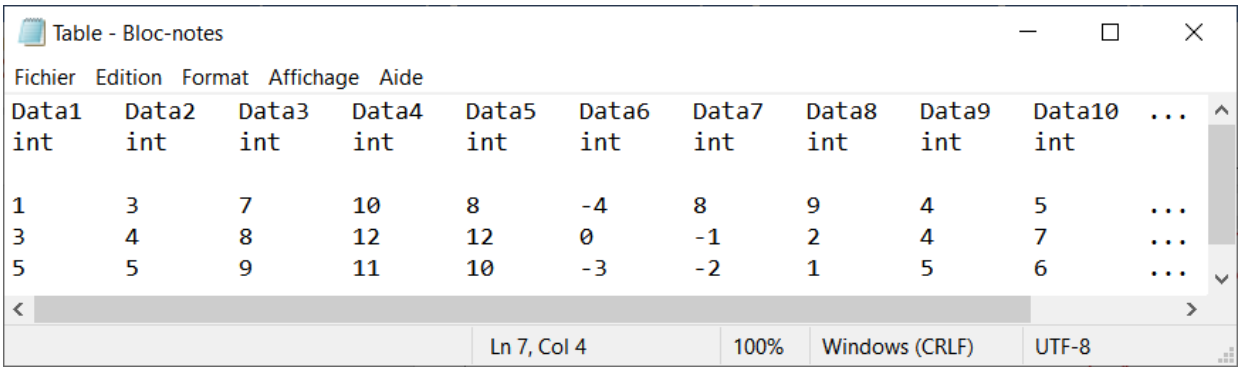
À titre d'exemple, repartons du fichier *csv* (comma separated values) "*fichier.csv*" précédent où les colonnes sont séparées par un point-virgule.

Temps (s)	Donnée 1 (binaire /8 bits)	Donnée 2 (signal en Volt)
0.00	56	- 2.15
0.01	79	- 1.05
0.02	137	- 0.41
0.03	174	+ 0.52
0.04	191	+ 1.46

```
document = open("fichier.csv","r")      # le fichier doit être dans le même dossier
                                        # que le script python (.py)
L_temps, L_data1, L_data2 = [], [], []      # initialisation des listes à lire
Liste_lignes = document.readlines()         # liste des lignes (séparateur "\n")
     # ["Temps;Donnée 1 ;Donnée2\n", "0.00 ;56 ;-2.15\n", "0.01;79;-1.05\n", …]


for i in range(1, len(Liste_lignes)) :
    lignei = Liste_lignes[i].strip("\n")

    L_ligne_i = lignei.split(";")

    L_temps.append( float(L_ligne_i[0])

    L_data1.append( int(L_ligne_i[1]) )

    L_data2.append( float(L_ligne_i[2]) )
```

B. 2 / Lecture d'un fichier avec un grand nombre de colonnes

Supposons cette fois un fichier "Table.txt" qui a un nombre élevé de colonnes. Pour balayer **chaque** ligne, il faut donc envisager une boucle *for*. On suppose ici que toutes les données sont de même type, et en ouvrant avec un éditeur de texte (pour *Windows* : *Bloc-notes* ou *Notepad++* par ex), on voit que le séparateur de colonnes est une **tabulation.**



On veut ici écrire la table dans une variable globale `M`, implémentée sous forme de liste de listes.

```
document = open("fichier.csv","r")      # idem
M = []      # initialisation de la matrice (liste de listes)

L_lignes = document.readlines()       # idem liste des lignes (séparateur "\n")

for i in range(3, len(L_lignes)) :          # ou, idem :
    ligne = L_lignes[i]                     for ligne in L_lignes[3:] :
    ligne = ligne.strip("\n")    # Rmq 1 : en fait cette ligne n'est pas nécessaire
              # Rmq 2 : la commande ligne.strip("\n") ne modifie pas la variable ligne
              # en place (contrairement, par exemple, à L.pop(-1) pour une liste L)

    L_col_lignei = ligne.split("\t")

    Lignei_matM = []

    for j in range(len(L_col_lignei)) :

        Lignei_matM.append( int(L_col_lignei[j]) )

    M.append( Lignei_matM )
```

B. 3 / Écriture d'un fichier avec un petit nombre de colonnes

Imaginons maintenant l'inverse du cas **B. 2** : on part d'une matrice `M` connue, et on veut écrire son contenu dans un fichier texte "Table.txt", avec une tabulation en guise de séparateur de colonnes.

```
document = open("fichier.csv","w")    # attention, quand on accède en écriture "w"
                        # toujours bien penser à fermer le fichier en fin de script.

txt_a_ecrire = ""   # str vide, ou si besoin, lignes d'en-tête !

for i in range(len(M)) :       # nombre de lignes

    for j in range(len(M[i])) :      # nombre de colonnes de la ligne i

        txt_a_ecrire += str(M[i][j]) + "\t"        # Rmq : pas de append pour les str

    txt_a_ecrire = txt_a_ecrire[:-2] + "\n"       # Suppression du dernier "\t" (en
                                                  # trop) et saut de ligne
document.write( txt_a_ecrire )    # on écrit dans le fichier

document.close()        # on le ferme (évite des conflits si on l'ouvre par ailleurs)
```

C / Rappels sur les chaînes de caractères en *Python*

C.1 / Caractère et chaîne de caractères

Contrairement à beaucoup d'autres langages de programmation, *Python* ne fait pas de différence entre un **caractère** et une **chaîne de caractères** (*string*, « liste » de plusieurs caractères). Pour lui, il n'existe que le type *str* pour coder des listes de caractères, qui peuvent éventuellement ne contenir qu'un seul élément.

- Une variable de type *str* est un peu comme une **liste**. Ainsi, si `texte = "Vive le semestre 1"` alors :
  - `len(texte)` renverra la valeur : 18 (les espaces comptent comme 1 élément)
  - `texte[0]` renverra : `"V"` et `texte[-1]` renverra : `"1"`

    Remarque: attention, ce n'est pas le chiffre 1 (qui serait un *int*), mais le caractère `"1"` (type *str*) ! Si on veut demander à *Python* d'interpréter de quel semestre il s'agit, il faut lui demander : `int(texte[-1])`
  - Rappel, comme pour une extraction de sous-liste `L[a:b]` avec `a` et `b` valeurs entières, avec `a < b` (fonctionne comme *range*), on peut extraire un *str* d'une chaîne plus grande :

    Ainsi `texte[:4]` renverra `"Vive"` et `"semestre 1"` correspond à l'extraction : `texte[-10:]` ou `texte[8:]`
- Si on veut faire interpréter à *Python* un texte comme un nombre, il faut en préciser le type. Exemples :
  - Je lis : `x = "24"` mais je voudrais l'interpréter comme un entier. Je tape alors : `x2 = int(x)`
  - Je lis : `y = "3.14"` mais je veux l'interpréter comme *float*. Je tape alors : `y2 = int(y)`
- Attention, l'opération « + » entre deux *str* (comme avec les *list*) n'est pas la somme mais la **concaténation**.
  - `x,y = "24","3.14" # double affectation`

    `x+y` renvoie `"243.14"` donc `float(x+y)` renvoie `243.14`

    Alors que `float(x) + float(y)` aurait renvoyé `27.14` (car `float(x)` et `float(y)` sont d'abord interprétés comme des nombres, et donc le « + » est compris comme une addition).

C.2 / Méthodes utiles pour les *str*

Je rappelle qu'une **méthode**, en programmation orientée objet, c'est un peu comme une **fonction** mais spécifique à un type d'objet. En *Python,* une fonction s'écrit `fonction(arguments…)` alors qu'une méthode s'écrit `objet.methode(…)` . Par exemple, pour des listes, nous avions vu la méthode `L.append(…)` .

Remarque: Nous avons vu que pour le type *list* où les méthodes sont « procédurales » (elles affectent la liste en argument **en place** sans renvoyer quoi que ce soit) : `L = [1,2,3]`

```
L.append(4)    # L vaut maintenant : [1,2,3,4]
L = L.append(5)    # L vaut maintenant : None
                   # (car pas de return !)
```

Pour le type *str* en revanche, les méthodes **renvoient** la valeur du *str* modifié, mais ne le modifient pas en place.

```
T = "ABCD"
T.strip("D")      # renvoie "ABC", mais T vaut "ABCD"  (non modifié)
T = T.strip("D")  # cette fois T est modifié par l'affectation et vaut "ABC"
```

- **La méthode `.strip()`** permet de supprimer un «morceau» (texte en argument) d'une chaîne de caractères.

→ `texte = "ABCD"` alors
  - `texte.strip("D")` renvoie `"ABC"`
  - `texte.strip("B")` renvoie `"ACD"`
  - `texte.strip("CD")` renvoie `"AB"`

Par exemple, si à l'issue de la lecture d'une ligne, on a `ligne = "Texte_de_la_ligne\n"` alors on peut supprimer le `"\n"` de fin de ligne en tapant : `ligne = ligne[:-2]`

En effet la commande `ligne[:-2]` (ou la commande équivalente `ligne[0:-2]`) extrait tous les éléments de `ligne` sauf les 2 derniers (ce qui vaut donc `"Texte_de_la_ligne"`) et la 1ère partie de la commande `ligne = …` s'interprète donc comme `ligne = "Texte_de_la_ligne"`, ce qui remplace l'ancienne valeur de `ligne`.

Cette commande est équivalente à taper `ligne = ligne.strip("\n")`

- **La méthode `.replace()`** permet de modifier un «morceau» d'une chaîne de caractères.

Si l'on travaille avec un logiciel (comme *Excel*) qui enregistre ses formats *float* avec une virgule et pas un point, on a des problèmes lors de l'interprétation des nombres. Par exemple, pour *Excel*, ¼ donne 0,25 (et pas 0.25 ! notez la différence de standard d'écriture du *float*). Alors si j'exporte cette variable comme texte depuis *Excel* vers *Python* :

```
Donnee_exportee = "0,25 "
```

Alors si je tape `float(Donnee_exportee)`, *Python* renverra une erreur. Il faudrait donc que je transforme la virgule en point. On peut le faire manuellement, mais si *Excel* a exporté en réalité un tableau de milliers de valeurs… il faut une méthode automatique.

```
Donnee_exportee.replace(",",".")
```

Cette méthode cherche toutes les occurrences du 1er argument (qui doit être du texte), et les remplace par le 2ème.

- **La méthode `.split()`** permet de récupérer les données d'une chaîne de caractères pour les stocker dans une liste. Elle est alors très utile pour extraire une série de données chiffrées.

Ligne	Commande	Résultat
`Ligne = "10 20 30"`	`Ligne.split( )`	`['10', '20', '30']`
`Ligne = "10;20;30"`	`Ligne.split(";")`	

Sans arguments, il y a un découpage chaque fois qu'il y a un ou plusieurs espaces, ou des tabulations (la tabulation possède la notation : `"\t"`). C'est pour cela qu'à la 1ère ligne, l'argument n'était pas nécessaire. En revanche, avec argument, il y a découpage chaque fois que l'argument mis entre guillemets est rencontré.

→ Nous obtenons alors une liste dont chaque élément est une chaîne de caractères qui peut être convertie en entier (int) ou flottant (float) avec les commandes associées.

Exemples : supposons qu'à la lecture d'une ligne on lise : `ligne = "1,1 1,2 1,3 1,4 1,5\n"`

1. Pour enlever le `"\n"` à la fin on tape la commande : `ligne = ligne.strip("\n")`
2. Pour transformer les points en virgule, on tape : `ligne = ligne.replace(",",".")`
3. Pour créer la liste des valeurs de cette liste, on tape : `LISTE = ligne.split()`
4. (pas besoin d'argument car les séparateurs de colonnes sont des espaces : par défaut).

À l'issue des 3 étapes précédentes, nous avons obtenu successivement :

1. `ligne = "1,1 1,2 1,3 1,4 1,5"`
2. `ligne = "1.1 1.2 1.3 1.4 1.5"`
3. `LISTE = ["1.1","1.2","1.3","1.4","1.5"]` `# Mais ce n'est pas une liste de nombres ! C'est une liste de str`

D / Ouverture, lecture, écriture et fermeture de fichier

D.1 / Ouverture et fermeture d'un fichier

- **Ouverture** avec la fonction `open()`

  La fonction `open("chemin","droit")` prend deux arguments, tous les deux sous la forme *str* :

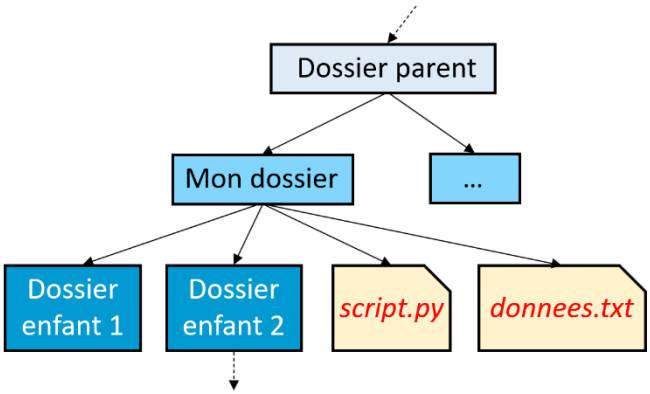
  - Le 1er argument est le `"chemin"` d'accès au document. Ce dernier peut être écrit sous la forme :
    - `"C:/Users/Info/TP/donnees.txt"` donc d'un chemin absolu
      Remarque: comme précisé en fin de **A.1** nous éviterons cette utilisation.
    - `"donnees.txt"` donc d'un chemin relatif ce qui suppose que le script et le fichier à ouvrir se trouvent dans le même dossier, au même niveau.

      Rappel : les extensions classiques pour les fichiers de données sont `".txt"`, `".dat"`, `".csv"`
  - Le 2ème argument est le droit donné à *Python* sur ce fichier, ce dernier peut être :
    - `"w"` pour *write* : droit en lecture et écriture
    - `"a"` pour *add* : droit en lecture et ajout de données (rare)
    - `"r"` pour *read* : droit en lecture seule

Exemples : imaginons l'arborescence ci-contre. Dans un même dossier sont rangés le fichier *donnees.txt* et le script *Python* que l'on est entrain d'éditer. Alors, pour ouvrir le fichier des données en lecture seule, on tapera :

```
data = open("donnees.txt","r")
```

→ Ceci crée alors une variable `data` de type *str* qui contient l'intégralité du document texte `"donnees.txt"`.



Remarque: si jamais le document `donnees.txt` n'existait pas, la commande `data = open("donnees.txt","r")` ne renverra pas une erreur, elle créera un fichier texte `donnees.txt` vierge. C'est ce qu'on fait lorsque l'on veut créer un document que l'on va écrire par la suite avec *Python* (notamment quand on veut sauvegarder le résultat d'un long calcul).

- **Fermeture** avec la fonction `close()`

  Attention ! il faut toujours fermer un fichier après l'avoir ouvert (surtout avec droits d'ajout et d'écriture), sinon il sera irrémédiablement considéré *corrompu* par l'ordinateur (perte de toutes les données !).

  Donc, à la fin de tout script où vous avez ouvert des fichiers, pensez à les refermer. **Exemples** de code type :

```
data = open("donnees.txt","r") # début du script, on ouvre le fichier
#### CORPS DU SCRIPT ####
#
# Tous le corps où vous faites le traitement des données (intégration, dérivation,
# moyenne glissante, calculs de moyennes ou écarts types, etc).
#
# Si besoin → enregistrement des données retouchées
#
#### FIN DU CORPS DU SCRIPT ####
data.close()  # en fin de script, on ferme le fichier proprement.
```

Remarque: notez que lors de l'ouverture, on a donné un nom de variable au document ouvert, par exemple *fichier*, et lors de la fermeture du fichier la commande doit appeler ce nom de variable. Par exemple : `fichier.close()`.

D.2 / Lecture et écriture d'un fichier

Dans le « corps » du script, donc entre l'ouverture du fichier et sa fermeture, nous allons vouloir lire et/ou écrire des lignes dans ce fichier.

- Si l'on veut seulement lire le fichier, on supposera que le corps du script est précédé de la commande :
  `donnees = open("document_contenant_les_donnees.txt","r")`
- Si l'on veut écrire dans le fichier, on supposera que le corps du script est précédé de la commande :
  `donnees = open("document.txt","w")`
  Remarque: notons que si *document.txt* n'existait pas déjà, il est alors créé automatiquement (vierge).

- **Lecture** avec la méthode `readlines()`

Une fois que la variable qui stocke tout le fichier texte a été créée (je l'ai notée `donnees` ci-dessus), on peut lui appliquer la méthode `readlines()` qui découpe la variable en une **liste de *str***. Par défaut, la méthode n'a pas besoin d'argument car elle suppose que le **séparateur de lignes** est le *str* `"\n"`. Exemple 1 :

Supposons que le fichier texte initial dans `document_contenant_les_donnees.txt` était le suivant :

VIVE
LA
PTSI

Alors à l'ouverture avec *Python* (fonction **open**), la variable `donnees` contient ≈ la valeur : `"VIVE\nLA\nPTSI"` mais attention, son type est `_io.TextIOWrapper` : ce n'est pas un type *str*.

Dans ce cas, `donnees.readlines()` renverra la liste `["VIVE\n" , "LA\n" , "PTSI" ]`

Exemples type :

```
doc = open("table_des_donnees.csv","r") # J'imagine que c'est le nom du document
# Supposons qu'à cette étape, on obtient :
#  doc = "Temps;Donnée 1 ;Donnée2\n0.00 ;56 ;-2.15\n0.01;79;-1.05\n0.02;137;-0.41\n0.03;174;0.52\n0.04;191;1.46"

Liste_data = doc.readlines()    # à cette étape, on obtient :
#  Liste_data = [ "Temps;Donnée 1 ; Donnée2\n" , "0.00 ;56 ;-2.15\n" , "0.01;79;-1.05\n" ,
                  "0.02;137;-0.41\n" , "0.03;174;0.52\n "0.04;191;1.46" ] une liste de 5 éléments (5 lignes)

N = len(Liste_data) # nombre d'éléments dans la liste, donc de lignes du document

""" Je suppose que j'aimerais créer maintenant 3 listes : une liste des temps, une liste des donnée_1 et une liste des donnée_2.
Mais il faut donc que je dégage la 1ère ligne, qui correspond aux noms des variables, puisque les données ne commencent qu'à la seconde ligne """

liste_temps = [ ]
liste_donnee1 = [ ] # quand on travaille avec des listes, il faut les initialiser vides
liste_donnee2 = [ ]

for i in range(1,N) : # Je pars de la ligne 1 (donc 2ème ligne) pour enlever l'en-tête
    ligne = Liste_data[i]    # la variable ligne va prendre successivement les valeurs
                             # des différents éléments (lignes) de la liste.

    ligne = ligne.strip("\n")  # j'enlève les "\n" à chaque fin de ligne. Par exemple
                               # pour i=1, avant cette instruction ligne ← "0.00 ;56 ;-2.15\n"
                               # donc ligne.strip("\n") renvoie "0.00 ;56 ;-2.15" et c'est la
                               # nouvelle valeur qu'on affecte à ligne, donc après cette
                               # instruction, on aura ligne "0.00 ;56 ;-2.15"

    # Si les valeurs numériques étaient mal codées ( 0,01 au lieu de 0.01 par ex.),
    # on aurait écrit ici : ligne = ligne.replace(",",".") voir fin de page 4, B.2.
```

```
        temps_i , donnee1_i , donnee2_i = ligne.split(";") # ici, on sépare la ligne en
                                    # éléments, en découpant suivant les délimiteurs de
                                    # colonnes, c'est-à-dire le caractère ";".
                                    # Exemple : pour i=1, on a vu ligne = "0.00 ;56 ;-2.15"
                                    # donc ligne.split(";") renvoie une liste des éléments
                                    # de la ligne. Ici, on a donc affecté temps_i ← "0.00"
                                    # donnee1_i ← "56" et donnee2_i ← "-2.15"
                                    # Attention, ce sont des str il va falloir les convertir…

        liste_temps.append( float(temps_i) ) # on convertit temps_i en float (décimal)
                                             # puis on l'ajoute à la fin de la liste des temps
        liste_donnee1.append( int(donnee1_i) ) # car au vu du tableau, donnee1_i a été
                                             # exportée comme une valeur entière
        liste_donnee2. append ( float(donnee2_i) )

    """ À l'issue des N – 1 itérations de la boucle for, on aura bien créé les 3 listes désirées, contenant chacune N – 1 éléments """
    doc.close() # et à la fin du script, on n'oublie pas de refermer le fichier !
```

C.3 / Écriture dans un fichier avec la méthode `write()`

Une fois la variable représentant le fichier créée (par ex : `fichier = open("document.txt","w")`), on peut écrire dans ce fichier avec la commande `fichier.write("en_argument_le_texte_à_écrire")`.

Attention, l'argument de `write()` doit être un type *str*. Donc si on a des données numériques, on doit les convertir.

Exemples:

si on a `x` ← `2.04` et `y` ← `-3.5` qu'on veut insérer comme nouvelle ligne dans le document, séparés par une tabulation, il faut taper le code : `fichier.write( str(x) + "\t" + str(y) + "\n" )`

```
# + pour la concaténation, "\t" pour la tabulation, "\n" pour retour à la ligne
```