Chapitre (S1) 6 Modules - Fichiers

Objectifs

- Connaître les différentes manières d'utiliser un module.
- Connaître et savoir utiliser quelques modules et fonctions associées.
- Connaître la notion de fichier.
- Savoir utiliser les traitements élémentaires sur les fichiers textes.

1. MODULES

1.1. Présentation

1.1.1. Qu'est-ce qu'un module ?

Des fonctions Python enregistrées dans un fichier peuvent être mises à disposition d'autres programmes Python. Cette programmation modulaire offre une grande souplesse et évite de programmer plusieurs fois les mêmes fonctions.

Un *module* est un fichier qui contient des objets (déclarations de variables, définitions de fonctions et de classes [1] susceptibles d'être *importées*), étendant les fonctionnalités initiales de Python. Enregistré sous un nom `module_filename.py`, le script, une fois appelé, permet l'utilisation des objets du module.

Les modules regroupent des variables, fonctions ou classes qui tournent autour d'une même thématique (nous verrons ultérieurement quelques modules incontournables), comme le calcul numérique, la statistique, ou le tracé de fonctions. Utiliser un module permet d'étendre les possibilités de python sans avoir à reprogrammer les fonctions usuelles du domaine utilisé. L'ensemble des modules disponibles est très vaste et en constante évolution. On peut dire que si on se pose un problème un peu général, il existe probablement déjà un module qui résout ce problème.

1. Utilisées en programmation orientée objet.

1.1.2. Les différents modes d'appel

L'utilisation d'un module se fait à l'aide de l'instruction **`import`**. On peut l'utiliser soit directement dans la ligne de commande, soit dans un script python.

- Une première façon de faire est d'importer uniquement les objets (constantes, fonctions ou classes) qui nous intéressent, comme dans l'exemple suivant utilisant le module `math`. En l'absence de la ligne `from math import cos,pi` qui réalise l'importation, on obtient le résultat suivant :

```
>>> cos(pi)
Traceback (most recent call last):
  File "<input>", line 1, in <module>
NameError: name 'cos' is not defined
```

  On écrit donc plutôt :

```
>>> from math import cos,pi
>>> cos(pi)
-1.0
```

  Ce type d'importation n'est pas très utilisé car il nécessite de connaitre dès l'importation l'ensemble des objets que l'on souhaite utiliser.
- On peut aussi importer tous les objets d'un module, qui sont alors directement utilisables :

```
>>> from math import *
>>> sin(pi/2)
1.0
```

  Ce type d'importation est assez gourmand en mémoire et peut poser des problèmes de conflits entre objets (cas de deux objets différents appartenant à deux modules distincts ayant le même nom).
- Une manière très pratique (et à privilégier dans la plupart des cas), est d'importer le module (et non pas les objets qu'il renferme), le plus souvent en le renommant

à l'aide d'un *alias*[2] (mot clé `as`). Celui-ci est défini au moment de l'importation du module, et il offre un autre moyen de faire référence à un module. Les objets du module sont alors accessibles par la syntaxe générale `nom_alias.objet`. Par exemple :

```
>>> import math as m
>>> m.sin(m.pi/2)
1.0
```

ou bien :

```
>>> import matplotlib.pyplot as plt
>>> plt.figure(1) # crée une zone graphique
<Figure size 1280x960 with 0 Axes>
```

Des informations sur le contenu d'un module peuvent être obtenues à l'aide des instructions suivantes.

- `dir(module_filename)` renvoie l'ensemble des objets présents dans un module,
- `help(module_filename)` renvoie des informations sur les fonctions du module,
- `help(module_filename.objet)` ou `help(nom_alias.objet)` (dans le cas où on a importé et renommé le module) renvoie l'aide relative à cet objet.

Mais pour tout savoir, rien ne vaut le site officiel `https://docs.python.org/fr`!

1.1.3. Quelques modules parmi d'autres

Comme signalé précédemment, le nombre de modules existants est très important et on ne peut être exhaustif en la matière. On peut cependant signaler quelques modules particuliers :

- `math` – module de fonctions mathématiques usuelles,
- `cmath` – module qui reprend la plupart des fonctions du module `math` pour les nombres complexe `(a+bj)`,
- `random` – module de fonctions de hasard et de probabilité,
- `numpy` – Module principal de calcul scientifique. Il permet la définition et la manipulation efficace de **tableaux**. Il fournit des outils pour l'algèbre linéaire, pour l'analyse de Fourier, pour la manipulation de nombres pseudo-aléatoires, …
- `matplotlib` et le sous module `matplotlib.pyplot` – Module qui permet de créer des graphiques de type courbes, histogrammes, spectres, barres d'erreur, …
- `scipy` - Module de calcul scientifique qui complète `numpy` et `matplotlib`. Il fournit des outils d'optimisation, d'algèbre linéaire, de statistiques, de traitement du signal, du traitement d'images, de résolution d'équations différentielles.

2. Un alias peut être vu comme une abréviation d'une commande ou la définition d'une commande complétée par des options souvent utilisées.

1.2. Module numpy-Tableaux

1.2.1. Généralités

Ce module de calcul scientifique prend en charge les fonctionnalités des modules internes `math` et `random`. En important le module `numpy`, les modules précédents deviennent des sous-modules de ce dernier.

```
import numpy as np
np.random.randint(0,10)
np.sqrt(2)
```

1.2.2. Tableaux

L'un des atouts du module `numpy` est la possibilité de manipuler des **tableaux** multidimensionnels. Ces derniers sont des objets permettant le stockage d'éléments repérés par des indices (de manière similaire aux listes), avec la condition que tous leurs éléments sont du même type. Le module propose des outils de manipulation et de calcul sur ces objets qui sont particulièrement efficaces. Ce qui rend ce module très performant dans le cadre du calcul numérique.

La création d'un tableau utilise la fonction `array`.

```
>>> import numpy as np
>>> T1 = np.array([1,2,3,4])
>>> T1
array([1, 2, 3, 4])
>>> type(T1)
<class 'numpy.ndarray'>
>>> T2 = np.array([[1,2,3],[4,5,6]])
>>> T2
array([[1, 2, 3],
       [4, 5, 6]])
```

Le module numpy permet de manipuler ces tableaux ou d'en extraire certaines informations. Ces fonctions sont disponibles dans l'aide python. Nous en présentons quelques-unes ci-dessous.

- `T = numpy.array([[1,0], [-1,1]])` : définition du tableau à 2 lignes et 2 colonnes
- `T[i,j]` : renvoie l'élément d'indice (i,j) du tableau, situé à la ligne $(i+1)$ et à la colonne $(j+1)$
- `T.shape` : renvoie taille du tableau sous forme d'un tuple
- `T[i,:]` : renvoie les éléments de la ligne $(i+1)$ (sous forme d'un tableau)
- `T[:,j]` : renvoie les éléments de la colonne $(j+1)$ (sous forme d'un tableau)
- `numpy.zeros([a,b,c,...])` : renvoie un tableau rempli de zéros de dimensions a, b, c, ...
- `numpy.ones([a,b,c,...])` : renvoie un tableau remplis de 1 de dimensions a, b, c, ...
- `np.linspace(a,b,n)` : création d'un tableau unidimensionnel de n valeurs flottantes comprises entre a et b (tous deux inclus)
- `np.arange(a,b,c)` : création d'un tableau unidimensionnel de valeurs flottantes comprises entre a (inclu) et b (exclu) par pas de c

Les opérations algébriques sur les tableaux sont dites vectorielles ou élément par élément (« element wise » en anglais). Cette vectorisation évite le recours à des boucles **for**, ce qui contribue à l'efficacité de ce module. Cette propriété différentie fortement les listes des tableaux.

- somme de deux tableaux de même taille :

```
>>> A = np.array([1,2,3])
>>> B = np.array([4,5,6])
>>> A+B
array([5, 7, 9])
```

  alors que pour deux listes, on obtient :

```
>>> A = [1,2,3]
>>> B = [4,5,6]
>>> A+B
[1, 2, 3, 4, 5, 6]
```

- multiplication par un nombre

```
>>> A = np.array([1,2,3])
>>> 2*A
array([2, 4, 6])
```

- élévation d'un tableau à une puissance

```
>>> A = np.array([1,2,3])
>>> A**3
array([ 1,  8, 27])
```

Ces opérations peuvent être appliquées à des tableaux de dimension supérieure à 1, comme ci-dessous avec des tableaux de dimension 2 :

```
>>> A = np.array([[1,2],[3,4],[5,6]])
>>> A
array([[1, 2],
       [3, 4],
       [5, 6]])
>>> B = np.array([[0,1],[-1,2],[0,2]])
>>> A+B # somme des tableaux
array([[1, 3],
       [2, 6],
       [5, 8]])
>>> A*B # produit coefficient par coefficient
array([[ 0,  2],
       [-3,  8],
       [ 0, 12]])
>>> 2*A # produit de chaque coefficient par un réel
array([[ 2,  4],
       [ 6,  8],
       [10, 12]])
>>> A**2 # chaque coefficient au carré
array([[ 1,  4],
       [ 9, 16],
       [25, 36]])
```

Les fonctions mathématiques de numpy fonctionnent également de manière **vectorielles** [3] car on peut les appliquer sur des tableaux de nombres :

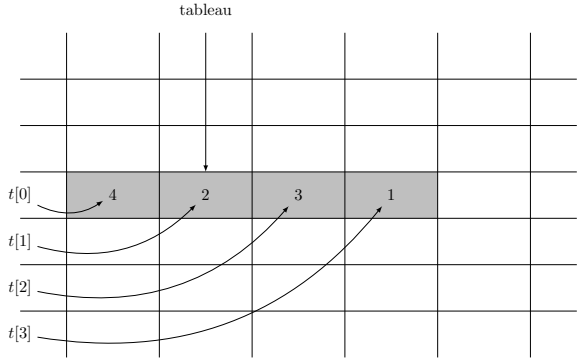
```
>>> import numpy as np
>>> x = np.linspace(0,2,10) #tableau de 10 valeurs de 0 à 2
>>> x
```

3. Ce qui n'est pas le cas des fonctions du module `math`.

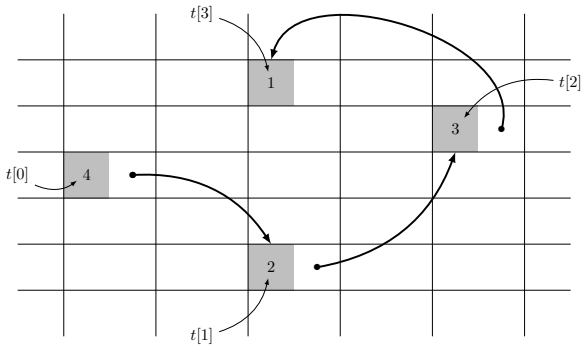
```
array([0.        , 0.22222222, 0.44444444, 0.66666667, \
↪ 0.88888889,
       1.11111111, 1.33333333, 1.55555556, 1.77777778, 2.        \
       ↪ ])
>>> np.exp(x) # on applique l'exponentielle à chaque valeur dans \
↪ x
array([1.        , 1.24884887, 1.5596235 , 1.94773404, \
↪ 2.43242545,
       3.03773178, 3.79366789, 4.73771786, 5.91669359, 7.3890561 \
       ↪ ])
```

1.2.3. Listes ou tableaux ?

En informatique, les tableaux et les listes constituent deux structures de données linéaires. Bien que présentant des similitudes syntaxiques, elles sont pourtant différentes d'un point de vue usage. Dans un **tableau**, les données sont stockées dans des emplacements consécutifs de la mémoire. En outre, les données, de même type, occupent toutes un même nombre c de cases mémoires. L'accès à chacune de ces données se fait à coût constant, la connaissance de l'adresse mémoire m du premier élément permettant la détermination de celle d'un élément d'indice i : $m + ic$. Une contrepartie à cette propriété est que la taille d'un tableau est fixée une fois pour toute au moment de sa création. Ce qui entraîne également que sa taille ne peut pas évoluer.



Dans une **liste**, les données sont stockées dans des emplacements quelconques de la mémoire. L'accès à ces données nécessite de conserver à la fois la donnée mais également son emplacement dans la mémoire. C'est pourquoi les listes sont appelées des **listes chaînées**. À chaque donnée est associé un pointeur qui précise la localisation dans la mémoire de la donnée suivante (sauf pour la dernière donnée qui pointe vers une valeur particulière `nil` de fin de liste). L'accès à une donnée d'indice i nécessite donc de parcourir la liste depuis son début jusqu'à la donnée recherchée. Le coût est alors en $O(i)$. Ce qui peut constituer un inconvénient. En revanche, la structure même de la liste autorise l'ajout et la suppression de données de manière dynamique. Il suffit de modifier les pointeurs.



On peut résumer les principales caractéristiques des listes et de tableaux qui constituent des avantages ou des inconvénients selon les besoins.

	Liste	Tableau
Structure de donnée	dynamique	statique
Accès aux données	coût variable	coût constant

Au sens strict du terme, les listes contiennent des données de même type. Les **listes Python**, de type `list`, sont une structure de donnée plus complexe. Elles conservent le caractère dynamique des listes, elles peuvent contenir des données de types différents et offrent un accès aux données de coût constant. Elles sont donc hybrides entre une liste au sens strict du terme et un tableau !

1.3. Module matplotlib

Le module `matplotlib` est l'une des bibliothèques python les plus utilisées pour représenter des graphiques. Nous nous intéresserons ici au sous-module `pyplot` de `matplotlib` qui regroupe un grand nombre de fonctions. Il s'agit d'un module particulièrement riche dont il n'est pas possible de décrire ici toutes les possibilités. Le

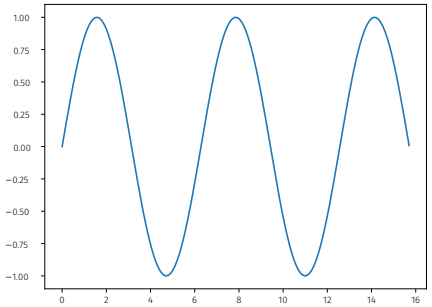
lecteur intéressé est invité à lire la documentation disponible sur internet sur le site http://matplotlib.org.

1.3.1. Graphe d'une fonction

L'un des premiers besoins graphiques est souvent le tracé d'une courbe représentative d'une fonction. Le script suivant présente le tracé d'un morceau de sinusoïde. L'affichage du graphique est obtenu grâce à l'appel de la fonction `show()`. Noter l'utilisation de la fonction `arange` du module `numpy` pour définir le tableau `x` des abscisses et l'application de la fonction sinus directement sur le tableau `x` pour obtenir le tableau des ordonnées. La fonction `plot` réalise alors automatiquement l'association abscisse-ordonnée.

```
import numpy as np
import matplotlib.pyplot as plt

x = np.arange(0, 5*np.pi, 0.1);
y = np.sin(x)
plt.plot(x, y)
```

```
plt.show()
```

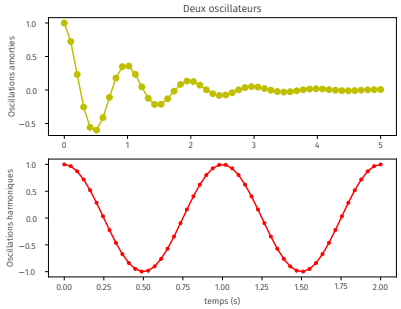


1.3.2. Plusieurs graphes

Plusieurs figures peuvent être représentées à l'aide de la fonction `subplot`.

```
import numpy as np
import matplotlib.pyplot as plt

x1 = np.linspace(0.0, 5.0)
x2 = np.linspace(0.0, 2.0)

y1 = np.cos(2 * np.pi * x1) * np.exp(-x1)
y2 = np.cos(2 * np.pi * x2)

plt.subplot(2, 1, 1)
plt.plot(x1, y1, 'yo-')
plt.title('Deux oscillateurs')
plt.ylabel('Oscillations amorties')

plt.subplot(2, 1, 2)
plt.plot(x2, y2, 'r.-')
plt.xlabel('temps (s)')
plt.ylabel('Oscillations harmoniques')
```

```
plt.show()
```



2. LES FICHIERS

Les structures de données que nous avons vues jusque là (chaînes, listes, tuples, dictionnaires) peuvent être utilisées au sein d'un programme, mais ne permettent pas de stocker durablement des données. Elles n'ont d'existence que dans la mémoire vive (ou RAM) de l'ordinateur, et à la fermeture de `python`, les données contenues dans ces variables sont perdues. Pour stocker ces données de manière durable (c'est à dire sur un disque dur, une clé USB ...), il faut utiliser une structure largement utilisée en informatique, dite de **fichier**. Dans ce chapitre nous nous limiterons aux fichiers **textes**. Un traitement sur un fichier se déroule toujours en trois temps :

1. l'**ouverture** du fichier ;
2. le **traitement** proprement dit ;
3. la **fermeture** du fichier.

Il existe trois types d'ouverture de fichiers texte, chacun se faisant par l'intermédiaire de l'instruction suivante :

```
open('NomDeFichier.extension','option')
```

Le choix de l'**option** permet :

- l'ouverture en **lecture** (suppose que le fichier existe déjà), avec l'option `'r'` (pour read) ;
- l'ouverture en **écriture** (pour créer un nouveau fichier ou écraser un fichier existant), avec l'option `'w'` (pour write) ;
- l'ouverture en **ajout** (permet d'ajouter du texte à la fin d'un fichier existant), avec l'option `'a'` (pour append) ;

Dans tous les cas, `open('NomDeFichier.extension','option')` renvoie un objet de type fichier, qu'il convient d'affecter dans une variable du programme.

2.1. Écriture d'un fichier

Commençons par créer un fichier texte en utilisant un script python. Pour que le fichier apparaissent dans **le même dossier** que celui dans lequel vous aurez sauvegardé le script, il faudra exécuter celui en choisissant dans le menu `pyzo` « Exécuter » puis « Démarrer le script » (raccourcis « Ctrl+Maj+E »). Pour exécutez le fichier en pressant simplement la touche « F5 » , il serait nécessaire d'indiquer le **chemin absolu** permettant d'accéder au fichier à partir de la racine (de la forme `C:\dossier \sousDossier \…\script.py`), ce qui est assez fastidieux. En fait, il n'est nécessaire d'exécuter la commande « Ctrl+Maj+E » qu'une seule fois, on peut se contenter de « F5 » par la suite (tant que l'on ne doit pas relancer `pyzo`).

Écrivons dans un fichier python le code suivant, puis exécutons-le par « Ctrl+Maj+E ».

```
Contenu = "Ceci est un exemple de fichier texte."
f = open("essai.txt", "w")
f.write(Contenu)
f.close()
```

La première ligne définit une chaîne de caractères qui formera le contenu du fichier. La deuxième ligne crée et ouvre en écriture un fichier nommé `essai.txt`, référencé par la variable `f`, dans le même répertoire que le script python (si un fichier portant le nom `essai.txt` existe déjà dans ce répertoire, celui-ci sera **détruit** et remplacé par le nouveau : **attention!** aux fausses manipulations qui pourraient vous amener à détruire certains fichiers importants). La troisième ligne recopie dans le fichier référencé par `f` la chaîne de caractères contenue dans la variable `Contenu`. La dernière ligne referme le fichier référencé par `f`.

Ce fichier `essai.txt` apparaît sur le disque dur de l'ordinateur, et en l'ouvrant par l'éditeur on observe sans surprise le contenu suivant (notez que les guillemets encadrant le texte n'apparaissent pas) :

```
Ceci est un exemple de fichier texte.
```

Le contenu du fichier apparaît ici sur une seule ligne. Pour séparer le texte sur plusieurs lignes, on insère le caractère spécial `\n` (bien que formé de deux signes, l'ensemble est considéré comme **un seul caractère** par Python) indiquant à l'ordinateur de revenir à la ligne à cet endroit. Par exemple :

```
Contenu = "Ceci est \n un exemple \nde fichier texte."
f = open("essai.txt","w")
f.write(Contenu)
f.close()
```

produit le nouveau fichier :

```
Ceci est
un exemple
de fichier texte.
```

Notez que la nouvelle ligne commence **juste après** le caractère « \n » , éventuellement par un caractère d'espace s'il y en a un. Il est possible d'effectuer plusieurs opérations d'écritures successives dans le fichier (avant sa fermeture), chacune venant s'insérer à la suite de la précédente. Par exemple :

```
Contenu1 = "Ceci est \n un exemple \nde fichier texte."
Contenu2 = "Et la suite."
f = open("essai.txt","w")
f.write(Contenu1)
f.write(Contenu2)
f.close()
```

produit :

```
Ceci est
 un exemple
de fichier texte.Et la suite.
```

Notez que le texte rajouté n'a pas été mis automatiquement à la ligne (il aurait pour cela fallu ajouter un caractère `\n`). Il existe une autre syntaxe permettant d'effectuer des écritures successives dans un fichier, consistant à utiliser la commande `writelines`, qui permet l'écriture successive des éléments d'une liste dont les éléments sont des chaînes de caractères (si ces éléments se terminent par le caractère `\n` , ce qui est généralement l'usage, ils formeront alors les lignes successives du fichiers). Par exemple :

```
Contenu = ["Ceci est \n","un exemple \n","de fichier texte."]
f = open("essai.txt","w")
f.writelines(Contenu)
f.close()
```

produit :

```
Ceci est
un exemple
de fichier texte.
```

Enfin, il arrive régulièrement que l'on souhaite stocker des valeurs **numériques** (entiers ou flottants) dans un fichier. Pour cela ces valeurs doivent être converties en les chaînes de caractères correspondantes, ce qui peut se faire par l'intermédiaire de l'instruction `str(valeur)`. Par exemple, le fichier

```
Toto
1.23
Tata
45.6
Tutu
789.0
```

peut s'obtenir par le script :

```
Noms = ["Toto", "Tata", "Tutu"]
Valeurs = [1.23, 45.6, 789.0]
f = open("essai.txt", "w")
for i in range(len(Noms)) :
    f.write(Noms[i] + "\n")
    f.write(str(Valeurs[i]) + "\n")
f.close()
```

Noter comment le caractère `\n` a été ajouté « manuellement » à la fin de chaque ligne. Le même résultat peut être produit en utilisant la commande `writelines` de la manière suivante :

```
Noms = ["Toto", "Tata", "Tutu"]
Valeurs = [1.23, 45.6, 789.0]
L = []
for i in range(len(Noms)) :
    L.append(Noms[i]+"\n")
    L.append(str(Valeurs[i])+"\n")
f = open("essai.txt","w")
f.writelines(L)
f.close()
```

Noter que les lignes #3 à #6 ont pour effet de construire la liste `L = ["Toto\n", "1.23\n", "Tata\n", "45.6\n", "Tutu\n", "789.0\n"]`

Enfin, l'ouverture du fichier en mode **ajout**, avec l'option `'a'`, fait en sorte que si le fichier existe déjà il ne soit pas écrasé mais que les instructions d'écritures s'ajoutent **à la fin** du fichier. Par exemple si le fichier `essai.txt` contient :

```
Toto
1.23
Tata
45.6
Tutu
789.0
```

alors le script suivant :

```
f = open("essai.txt","a")
f.write("Titi\n")
f.write("13.5\n")
f.close()
```

a pour effet de modifier ce fichier (qui reste de même nom) en :

```
Toto
1.23
Tata
45.6
Tutu
789.0
Titi
13.5
```

2.2. Lecture d'un fichier

On suppose maintenant qu'un fichier existe (pour simplifier placé dans le même répertoire que le script python que nous allons utiliser, sans quoi il faudrait indiquer son chemin absolu) et nous souhaitons extraire son contenu. La démarche est similaire à celle d'écriture, mais l'option est `'r'` pour un accès en **lecture** (une tentative d'écriture dans un fichier ouvert en lecture provoque une erreur à l'exécution), et l'instruction est `read` pour une lecture globale. Par exemple, si le fichier `essai.txt` contient :

```
Ceci est
un exemple
de fichier texte.
```

alors le script suivant :

```
f = open("essai.txt","r")
Contenu = f.read()
f.close()
```

a pour effet de placer dans la variable `Contenu` la chaîne de caractères : `"Ceci est\nun exemple\nde fichier texte.\n"`

Sur le même fichier, la commande `readlines` permet de récupérer une liste dont chaque élément est une ligne du fichier :

```
f = open("essai.txt","r")
L = f.readlines()
f.close()
```

a pour effet de placer dans la variable `L` la liste de chaînes de caractères : `"Ceci est\n","un exemple\n","de fichier texte.\n"`

Dans le même esprit, il est possible d'utiliser une boucle **`for`** pour parcourir successivement toutes les lignes du fichier :

```
f = open("essai.txt","r")
for ligne in f :
    instructions
f.close()
```

Dans cet exemple, la variable `ligne` recevra successivement les chaînes de caractères `"Ceci est\n"` puis `"un exemple\n"` et enfin `"de fichier texte.\n"`, sur lesquelles un traitement pourra être opéré. Par exemple, si le fichier `essai.txt` contient :

```
Toto
1.23
Tata
45.6
Tutu
789.0
```

et que l'on veut reconstruire les listes `Noms=["Toto","Tata","Tutu"]` et `Valeurs=[1.23,45.6,789.0]`, on peut procéder ainsi :

```
f = open("essai.txt","r")
Noms = []
Valeurs = []
i = 1
for ligne in f :
    if i%2 == 1:
        Noms.append(ligne[:-1])
    else :
        Valeurs.append(float(ligne))
    i += 1
f.close()
```

Notons plusieurs points. D'abord, le traitement doit être différents suivant les lignes. Celles contenant un nom doivent être ajoutées à la liste `Noms` après avoir enlevé le caractère `\n` final (ce qui peut se faire par l'opération de « slicing » sur les indices `[:-1]`, puisque l'indice -1 désigne de dernier caractère). Par contre celles qui contiennent un nombre doivent être ajoutées à la liste `Valeurs` après avoir converti la chaîne de caractères en nombre, ce qui se fait par l'instruction `float(chaîne)` puisque la chaîne correspond à une valeur flottante ici ; si c'était un entier on utiliserait bien sûr `int(chaîne)` (il n'est pas nécessaire ici d'enlever le caractère `\n` final, qui ne perturbe pas la conversion : par exemple `float("1.23\n")` renvoie bien le flottant 1.23). La distinction entre les deux types de lignes se fait à l'aide d'un compteur dont on teste la parité (si impaire c'est une ligne contenant un nom, si paire c'est une ligne contenant une valeur).

2.3. Manipulations autour des fichiers

Lorsqu'on travaille avec des fichiers, on peut avoir besoin de faire certaines manipulations « systèmes » du type : renommer un fichier, le changer de répertoire, créer un répertoire, …. On peut effectuer ces opérations directement en langage Python en utilisant des commandes regrouper dans le module `os`. En voici par exemple

quelques unes (les informations de cette partie sont donnée à titre de complément et ne sont pas à connaître) :

- `mkdir(rep)` : crée un répertoire ;
- `listdir(rep)` : renvoie la liste des fichiers contenus dans le répertoire ;
- `remove(fichier)` : efface un fichier (opération non réversible !) ;
- `rename(source,dest)` : renomme le fichier *source* en *dest*.

2.4. Exercices

Exercice 1 [Sol 1]

Écrire une fonction `fichAlea(n : int, nomFichier : str) -> None` où n est un entier naturel non nul, et qui va créer un fichier (s'il existe déjà il sera écrasé), dont le nom sera la chaîne contenue dans le paramètre `nomFichier`, et contenant n lignes correspondant chacune à un nombre flottant tiré au sort entre 0 et 1.

Par exemple, l'appel `fichAlea(4,"essai.txt")` pourra produire le fichier `essai.txt` suivant :

```
0.17051593186373637
0.0895356653826912
0.6449238466556816
0.40568348964764167
```

L'obtention d'un nombre flottant tiré au sort entre 0 *et* 1 *se fait par la commande* `random()`*, après l'avoir importée du module* `random` *par la commande* `from random import random`.

Exercice 2 [Sol 2] Écrire une fonction `moyFich(nomFichier : str) -> float` où `nomFichier` contient le nom d'un fichier dont chacune des lignes correspond à un nombre, et renvoie la moyenne de ces nombres.

Par exemple, l'appel `moyFich("essai.txt")`, sur le fichier `essai.txt` produit à l'exercice précédent, va renvoyer la moyenne des quatre nombres

0.17051593186373637 ; 0.0895356653826912

0.6449238466556816 ; 0.40568348964764167

qui vaut 0.32766473338743773.

SOLUTIONS DES EXERCICES

Solution 1

```
def moyFich(nomFichier : str) -> None :
    """
    Renvoie la moyenne des nombres figurant
    sur chacune des lignes du fichier de nom
    nomFichier
    """
    f = open(nomFichier,"r")
    n, S = 0, 0
    for ligne in f :
        S += float(ligne)
        n += 1
    f.close()
    return(S/n)
```

```
from random import random

def fichAlea(n : int, nomFichier : str) -> None :
    """
    Crée un fichier de nom nomFichier
    et formé de n lignes contenant
    chacune un flottant tiré au sort
    entre 0 et 1.
    """
    f = open(NomFichier,"w")
    for _ in range(n) :
        f.write(str(random())+"\n")
    f.close()
```

Solution 2