Chapitre 4
Piles, Files - Manipulations de fichiers

Simon Dauguet
simon.dauguet@gmail.com

29 novembre 2024

Le but de l'informatique, est de réaliser des tâches fastidieuses et souvent nombreuses de façons automatisés. Le but est de délégué à l'ordinateur tous les calculs pénibles. L'inconvénient de faire de nombreux calculs, c'est qu'on obtient de nombreux résultats. On a déjà une façon de conserver de nombreux résultats, en les mettant dans des listes, par exemple. Mas cette technique a ses limites. Dans le cas d'une trop grande quantité de données, l'affichage pose problème à Python et il n'est pas forcément aisé de s'y retrouvé.

Une meilleure méthode pour manipuler une grande quantité de données, consiste à les sauvegardées dans des fichiers. Cette méthode a l'avantage de ne pas surchargé la mémoire de l'ordinateur. L'inscription des données dans in fichier permettra plus facilement de manipuler un plus grande de données.

Nous allons donc voir ici comment manipuler des fichiers et des dossiers avec Python.

Table des matières

1 Les piles et les files

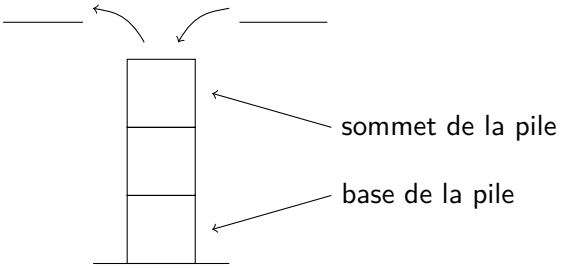
1.1 Les piles

Définition 1.1 (Pile) :
Une pile est une structure de données où seul le dernier élément entré est accessible. Cette structure respecte le principe "dernier entré, premier sorti" (en anglais **`LIFO : Last In, First Out`**).

Ainsi comme une pile d'assiettes, un pile est un ensemble d'éléments qui sont empilés les uns sur les autres afin de ne former qu'une seule pile.

⚠ Toutes les opérations possibles ne se font donc qu'au sommet de la pile. On ne peut pas accéder à un élément au milieu de la pile (retirer l'assiette du milieu ferait s'effondrer la pile), ni accéder au premier élément de la pile.

- Lorsque l'on retire un élément à la pile, on dit qu'on *dépile* cet élément.
- Lorsque l'on ajoute un élément à la pile, on dit qu'on *empile* cet élément.



La difficulté que nous allons rencontrer ici, c'est que le type "pile" n'existe pas en python. Ce qui s'en rapproche le plus sont les `list`. Mais les éléments d'une liste en python peuvent accéder un peu n'importe comment. Le concept de pile est donc un objet qui existe mais qui prend surtout son sens dans d'autres langages informatiques.

Toutefois, la notion est importante. C'est une notion fondamentale et fondatrice dans la théorie de l'informatique. Et qui plus est, certains objets se manipulent comme des piles en python. C'est ce que nous verrons avec la lecture de fichier.

Par conséquent, pour continuer à étudier les piles, nous le ferons d'un point de vue théorique en python (on ne présentera pas de type particulier ni de fonctions spécifiques). Et donc, afin d'utiliser un objet pile, il faut *a minima* trois fonctions :

- `creer_pile() -> 'Pile'`, renvoie une pile vide
- `depiler(p: 'Pile') -> object`, dépile et renvoie le sommet de la pile `p` (=*pop* en anglais).

- `empiler(p: 'Pile', e: object) -> None`, empile l'élément `e` sur le sommet de la pile `p` (=*push* en anglais).

Exercice 1 :
Écrire les instructions pour créer une pile puis la remplit successivement avec | A | A | C | (C étant le sommet).

Exercice 2 :
Dessiner la succession d'états de la pile lors des instructions suivantes

```
>>> p = creer_pile()
>>> empiler(p, 10)
>>> empiler(p, 13)
>>> depiler(p)
>>> empiler(p, 7)
```

Afin d'accroître la facilité d'utilisation de la pile, on peut y ajouter d'autres fonctions qui ne modifient pas la pile, telles que :

- `taille(p: 'Pile') -> int`, renvoie le nombre d'éléments dans la pile `p`
- `est_vide(p: 'Pile') -> bool`, renvoie `True` si la pile `p` est vide, `False` sinon
- `sommet(p: 'Pile') -> object`, renvoie, sans dépiler, l'élément au sommet de la pile `p`

1.2 Files

Définition 1.2 (File) :
Une file est une structure de données où les éléments sont introduits par un bout et ne sont accessibles que depuis l'autre bout. Cette structure respecte le principe "premier entré, premier sorti" (en anglais `FIFO : First In, First Out`).

Remarque :
Les files sont très courantes dans la vie quotidienne. C'est le cas des queues à une caisse, par exemple. Ou des distributeurs de bonbon comme les Pez.

Là non plus, en python, il n'y a pas de type "file" à proprement parlé. Ce qui s'en rapproche le plus correspond de nouveaux aux listes.

Toutefois, on aura besoin de ce concept un peu plus tard dans l'année, dans la théorie des graphes.

Pour pourvoir penser les files en python, on a donc besoin de quelques fonctions fictives :

- La fonction `queue(f:"file", obj:"object") -> "file"` qui permet de mettre `obj` à la queue dans la file.

- La fonction `dequeue(f:"file") -> "obect"` qui permet de retirer le premier élément de la file (donc celui qui a été mis le premier dans la file) et qui renvoie cet élément. La file est alors modifiée et contient un élément de moins (comme la méthode `pop` pour les piles).
- La fonction `taille(f:"file") -> int` qui permet de connaître le nombre d'élément dans la file.
- La fonction `est_vide(f:"file") -> bool` qui permet de savoir si la liste est vide ou non.
- La fonction `tete_de_file(f:"file") -> "object"` qui permet de renvoyer la tête de la file.

1.3 Utilisation en python

Le langage python implémente les piles et les files à l'aide du type `list` au travers des fonctions `pop()`, `append()` et `len()`. Ainsi,

Piles	
Fonction	Équivalent Python
`p = creer_pile()`	`p=[]`
`empiler(p, e)`	`p.append(e)`
`depiler(p)`	`p.pop()`
`taille(p)`	`len(p)`
`est_vide(p)`	`len(p) == 0`
`sommet(p)`	`p[-1]`

Files	
Fonction	Équivalent Python
`f=creer_file()`	`f=[]`
`queue(f,e)`	`f.append(e)`
`dequeue(f)`	`f.pop(0)`
`taille(f)`	`len(f)`
`est_vide(f)`	`len(f) == 0`
`tete_de_file(f)`	`f[0]`

2 Manipulations de fichiers

On se contentera d'une manipulation somme toute assez sommaire. On ne se préoccupera que des fichiers les plus simples. Le but n'est que la sauvegarde de données d'algorithmes simples, typiquement des nombres ou des lettres. On ne se préoccupera donc pas des problèmes de sauvegarde de données complexes issues de logiciels très spécifiques (logiciels vectoriels de dessins, sauvegardes de jeux vidéos, photos trafiquées, musique etc). En gros, on se bordera aux fichiers textes bruts.

2.1 Présentation générale

2.1.1 Les types de fichiers

Définition 2.1 (Fichier texte) :
Un fichier texte est un fichier dans lequel les données sont sauvegardées suivant le codage ASCII, ou Unicode ou Utf-8, qui sont des façons de coder chaque caractère du clavier. Les caractères et les nombres sont donc enregistrés selon ce codage.

Un fichier texte peut avoir plusieurs extensions possible : .txt, .csv, .py, .html etc.

Un fichier texte est l'un des fichiers de données les plus simples qui soient. Il peut être ouvert avec des logiciels extrêmement basiques tel que Notepad, Bloc-note et aussi plus compliqués comme Word.

Remarque :
Attention aux problèmes d'encodage entres les différents logiciels et entre les différents systèmes d'exploitation (Windows/Mac). Chaque logiciel et chaque système d'exploitation à un type d'encodage des fichiers spécifique par défaut qui ne sont pas toujours les mêmes.

Définition 2.2 (Fichier binaire) :
Un fichier binaire est un fichier dans lequel les données enregistrées sont codées en binaire. Les nombres sont enregistrés selon leur valeur en binaire.

Parmi les extensions de fichier binaire, on a : .dat, .mpeg, .bmp, .mp3 etc.

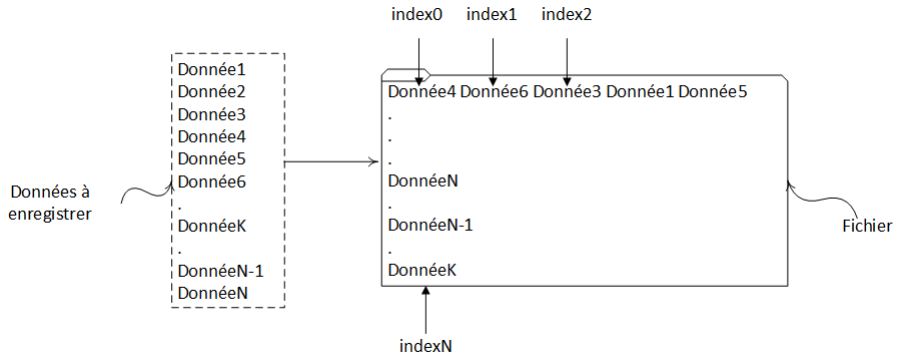
2.1.2 Les différentes types d'accès aux données

Un ordinateur n'étant pas un être humain, il ne manipule pas un fichier de la même manière que nous manipulons une feuille de papier. On peut voir un fichier informatique comme une sorte de pile géante. On empile alors les données dans cette immense colonne. Il y a essentiellement deux façons d'organiser cette pile.

2.1.2.1 Accès direct

C'est le type d'enregistrement, d'empilement des données la plus naturelle (pour nous) qui soit.

Définition 2.3 (Fichier en accès direct) :
Dans un fichier en accès direct, les données enregistrées sont globalement les mêmes mais pas forcément dans le même ordre que les données initiales à enregistrer. Pour pouvoir accéder à ces données, le fichier est muni d'un index répertoriant la place de toutes les données dans le fichier.



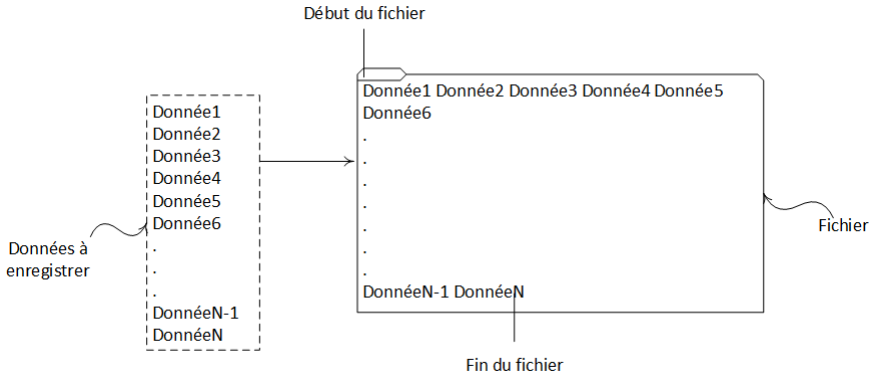
Remarque :
Ce type de fichier a l'avantage de pouvoir permettre un accès rapide et donc la lecture ou la modification d'une donnée précise très rapidement sans modification des autres données.

En revanche, elle a l'inconvénient de nécessiter d'avoir un index et de connaître la table index/données.

C'est un peu la façon dont on fonctionne. Nous avons, nous humains, une vision d'ensemble de la feuille ce qui nous permet de pouvoir faire une sorte d'index à vue d'œil. C'est un peu comme ça que l'on conçoit intuitivement un fichier.

2.1.2.2 Accès séquentiel

Définition 2.4 (Fichier séquentiel) :
Dans ce type de fichier, les données sont rangées selon l'ordre chronologique d'inscription. Les premières données enregistrées sont en début de fichier et les dernières données enregistrées en fin de fichier.



On remplit la pile étage par étage. Les données les plus vieilles sont au fond, les plus récentes en haut de la colonne.

Remarque :
L'avantage est que ce type de fichier s'auto-suffit. Il n'est pas nécessaire d'avoir un index en plus qui est une sorte de guide de lecture, de mode d'emploi.

L'inconvénient majeur est que, pour atteindre une certaine donnée, il est impératif de lire toutes les données qui précèdent. Il faut donc à chaque fois relire les premières données du fichier, puisque l'on ne peut pas savoir, a priori, où se trouve la donnée cherchée.

Ce type de fichier est une sorte de fichier à l'aveugle. Pour retrouver une donnée dans le noir, on est obligé à chaque fois de repartir du début et de compter les pas que l'on fait, à tâtons.

Bien sûr, c'est comme ça que va fonctionner Python ...

Remarque :
Dans les deux types de fichiers, il est nécessaire d'avoir des séparateurs permettant d'identifier les différentes données, de les séparer. Les séparateurs sont les tabulations, les sauts de lignes ou les retours chariots qui ont des marqueurs particuliers.

2.2 Gestion de fichier en Python

2.2.1 Principe général

Python fonctionne avec le système de fichier en accès séquentiel. Il ne faudra surtout pas l'oublier. Et en plus, Python est mono-tâche à l'extrême. L'esprit humain est capable de prendre du recul et pouvoir faire plusieurs tâches en même temps. Python ne peut pas. Il ne peut faire qu'une seule chose à la fois. Et ça va être le souci majeur. C'est complètement contre-intuitif.

Plus particulièrement, il faudra spécifier à chaque fois pour quelle tâche Python ouvre un fichier. Si un fichier est ouvert pour lecture, Python ne pourra que le lire et ne pourra pas écrire. Et si le fichier est ouvert pour écriture, Python ne pourra pas lire le fichier. Et bien sûr, Python ne peut pas ouvrir le même fichier deux fois en même temps. On ne peut pas ouvrir un fichier en lecture et écriture en parallèle pour pouvoir vérifier ce que l'on écrit. Si un fichier est ouvert pour une tâche, le fichier est occupé. On ne peut donc pas le réutiliser pour faire autre chose.

Il faudra constamment faire un jeu d'ouverture/fermeture des fichiers. Dès qu'un fichier est ouvert, il faudra immédiatement penser à le fermer. Sans quoi, on aura des erreurs systèmes.

Il faut également voir Python comme une sorte de bras à tête de lecture. Une fois une donnée lue, on ne peut plus revenir en arrière. Il faut fermer le fichier et le ré-ouvrir pour pouvoir y accéder à nouveau. Un fichier ne se parcourt que dans un seul sens. Ce sont des voies en sens unique. Impossible de faire demi tour ou de se souvenir de quelque chose de passer si ça n'a pas été sauvegardé dans une variable.

2.2.2 Ouverture / Fermeture de fichiers

Les commandes pour pouvoir manipuler des dossiers sont continues dans un module qu'il faudra importer à chaque fois, bien entendu.

Définition 2.5 (Module `os`) :
Le nom correspond à l'acronyme bien connu de *operating system*. C'est le système d'exploitation (Windows, MacOS, Linux etc). Le module `os` donne des fonctionnalités de manipulation de dossiers et de fichiers. Il permet de pouvoir circuler dans l'arborescence de l'ordinateur.

Il existe deux façons pour décrire informatiquement le chemin d'un fichier : le chemin relatif et le chemin absolu.

Définition 2.6 (Chemin absolu / Chemin relatif) :
Il existe deux façons pour décrire l'emplacement d'un fichier ou d'un dossier.

- On appelle chemin absolu d'un fichier son adresse complète dans le système. C'est l'adresse depuis la racine du système. Sous Windows, un chemin absolu est donc un chemin de la forme `C:\Doss1\Doss2\Fichier.ext`. Ne pas oublier l'extension du fichier.
- On appelle chemin relatif d'un fichier le chemin à parcourir à partir du dossier courant. Le répertoire courant est représenté par un point. Le répertoire au dessus est indiqué par deux

points à la suite. Dans un chemin relatif, il faut toujours commencer par le répertoire courant. Par exemple `".\.."` est l'adresse du répertoire parent. Ne pas oublier l'extension du fichier.

Remarque :
Le type de délimiteur entre deux noms de dossier (\ ou /) dépend de l'os que l'on utilise. Sous Linux, il faut utiliser / ; sous windows, il faut utiliser \ etc. Il ne faut donc pas oublier d'adapter les règles typographiques du chemin d'accès en fonction de l'os.

De même, la racine n'est pas la même selon l'os que l'on utilise. Un chemin absolu sous windows démarre toujours par `C:\` ; il démarre toujours par / sous Linux etc. Là aussi, il faut adapter les adresses en fonction de votre ordinateur.

Remarque (Capytale) :
Capytale utilise un mode de fonctionnement proche de Linux. En particulier, il permet de faire des arborescences virtuelles de dossiers. Tous les dossiers créés lors d'une session sont perdus lors de la clôture de la session. Ils ne sont donc disponible que tant que vous êtes en travailler dans le même fichier (et que vous ne recharger par la page).

Par ailleurs, Capytale utilise le symbole / comme délimiteur de dossier. Le répertoire de travail virtuel est toujours **`/home/capytale_user`** en début de session.

Le module `os` contient essentiellement les commandes suivantes permettant de manipuler les fichiers :

Commande	Description
`os.getcwd()`	(pour *get current working directory*) Donne le dossier de travail actuel sous forme d'une chaîne de caractères. C'est dans ce dossier que les fichiers seront sauvegardés et cherchés par défaut.
`os.chdir("C:\path")`	(pour *change directory*) Change le dossier de travail. L'entrée `"C:\path"` doit être une chaîne de caractères donnant l'adresse du nouveau dossier en chemin absolu ou relatif.
`os.mkdir("NouvDoss")`	(pour *make directory*) Création d'un nouveau dossier de nom **`NouvDoss`** (nom donné en chemin relatif ou absolue). Les dossiers intermédiaires doivent être créés les uns à la suite des autres.
`os.remove("NomFich")`	Suppression du fichier **`NomFich`** du répertoire courant. On peut supprimer un fichier d'un autre répertoire en indiquant le chemin relatif ou absolu du fichier à supprimer. Ne pas oublier l'extension du fichier.
`os.rmdir("Doss")`	Supprime le dossier **`Doss`** s'il est vide. Il faudra au préalable supprimer tous les fichier du dossier.

`os.listdir("Doss")`	Renvoie la liste de tous les fichiers et dossiers dans le dossier `"Doss"` (donné en chemin relatif ou absolu).

Remarque :
Lors de la saisie du chemin d'un dossier ou d'un fichier, il est (parfois) possible d'utiliser l'autocomplétion en appuyant sur Tab dans l'interpréteur. Si les premières lettres correspondent à un unique dossier ou fichier, le système complétera automatiquement l'adresse pour correspondre. Ça permet de ne pas tout écrire à chaque fois, de gagner du temps et aussi d'éviter des fautes de frappes.

Définition 2.7 (Fonction `open` et `close`) :

- La fonction `open` permet d'ouvrir un fichier pour une tâche particulière (lecture ou écriture). Une fois ouvert, il faut faire quelque chose de ce fichier. Il faut le mettre dans une variable. La syntaxe est la suivante :

```
1 fich=open(Chemin:str, mode:str)
```

  L'entrée **`Chemin`** est une chaîne contenant le nom d'un fichier dans le répertoire courant ou l'adresse relative ou absolue d'un fichier avec l'extension du fichier. L'entrée **`mode`** est une chaîne de caractères définissant comment le fichier doit être ouvert (pour lecture ou écriture essentiellement).
- Pour "libérer" le fichier afin qu'il soit réutilisable par la suite et ne pas avoir de message d'erreur disant que le fichier est déjà en cours d'utilisation, il faut refermer le fichier. La fermeture se fait par la commande

```
1 fich.close()
```

Parmi les différents modes possibles d'ouverture d'un fichier et leurs combinaisons, ceux qui nous intéresserons le plus sont :

Commande	Description
`open("Nom","r")`	Lecture (*read mode*). Le fichier n'est ouvert que pour lecture.
`open("Nom","w")`	Écriture (*writing mode*). Le fichier n'est ouvert que pour écriture. ATTENTION ! Si le fichier contient déjà des données, elles seront écrasées. Python écrit dès le début du fichier. ATTENTION ! Si le fichier contient déjà des données, celle-ci seront écrasées par les nouvelles. ATTENTION ! Si le fichier n'existe pas, il sera créé.

`open("Nom","a")`	Écriture à la suite (*append mode*). Le fichier est ouvert pour écriture mais les données sont rajoutées à la suite du fichier. Les données déjà présentes (si elles existent) ne sont pas modifiées. ATTENTION ! Si le fichier n'existe pas, il sera créé.
`open("Nom","r+")`	Lecture/Écriture. L'écriture et la lecture des données sont possibles. L'écriture se fait en fin de fichier. Et on ne pourra voir ce qui a été ajouté qu'à l'ouverture suivante du fichier. Python fait une sorte de photo de fichier à un instant t et c'est cela qu'il lit. Si le fichier est modifié, ce n'est pas la photographie qu'on a sous les yeux. Il faudra reprendre un cliché pour voir les changements.

Remarque :
Il est possible de préciser en plus `"t"` ou `"b"` pour préciser si c'est un fichier texte ou binaire. Par défaut, Python n'ouvrira que des fichiers texte. On pourra donc avoir des modes `"r+b"` par exemple pour l'ouverture pour lecture et écriture d'un fichier binaire.

2.2.3 Écriture

Définition 2.8 (Fonction `write`) :
Une fois un fichier ouvert pour écriture dans une variable **`fich`** (avec un mode `a` ou `w`), l'écriture se fait avec la fonction **`write`** dont la syntaxe est :

```
1 fich.write("Str")
```

La fonction **`write`** ne prend en paramètre que des chaînes de caractères. Il est possible cependant de composer la fonction **`write`** avec les manipulations de chaînes.

Définition 2.9 (Commande de mise en forme lors de l'écriture) :
Lors de l'écriture, il est possible de mettre en valeur les données en les séparant. Les commandes disponibles sont les suivantes :

- **`\n`** : Retour chariot, changement de ligne (*newline*). Elle s'utilise sans espace. On rappelle qu'un espace est un caractère particulier. Donc par exemple, la chaîne `"blablabla\nbliblibli"` est une chaîne sur 2 lignes. La première ligne contient la chaîne `"blablabla\n"` et la deuxième ligne contient la chaîne `"bliblibi"`.
- **`\t`** : Espacement (*tabulation*). C'est juste un séparateur indiqué pour séparer les données (on rappelle encore que l'espace est un caractère particulier qui ne sépare rien). Par exemple, la chaîne `"bla\tblu\n\bli\tblo"` est une chaîne de caractères sur deux lignes dont la première ligne est composée du mot `"bla\t"` et du mot `"blu\n"` et la seconde ligne est composée du

mot "bli\t" et du mot "blo".

!!! ATTENTION !!!

⚠ Dans le cas où l'on veut écrire dans un fichier la valeur d'une variable et pas le nom de la variable, il ne faudra pas mettre le nom de la variable entre guillemets. Il faudra d'abord convertir la variable en une chaîne. C'est le but de la fonction `str()`.

Exemple 2.1 :
On considère le code suivant permettant de calculer l'aire de 5 couronnes concentriques :

```
1 def AireDisque(r1,r2,r3,r4,r5) :
2     r=[r1,r2,r3,r4,r5]
3     A=[mt.pi*r[0]**2]
4     fichier=open("Aire.txt","w")
5     fichier.write("Premier disque : "+str(np.pi*r[0]**2)+"\n")
6     for i in range(1,len(r)) :
7         A=A+[np.pi*r[i]**2]
8         fichier.write("Couronne "+str(i)+" : "+str(A[i]-A[i-1])+"\n")
9     fichier.close()
```

2.2.4 Lecture

Une fois un fichier ouvert pour lecture dans une variable **fich**, (avec un mode "r" par exemple), la lecture du fichier est une chaîne de caractères. Pour pouvoir y accéder, il est fortement recommandé de la sauvegarder dans une variable. Il y a trois fonctions pour lire un fichier :

Commandes de lectures

Commande	Description
`lecture=fich.read()`	La variable **`lecture`** est une chaîne de caractères contenant l'intégralité du fichier. Avec les commandes de tabulation, retour chariot etc.
`lecture=fich.read(n)`	La variable **`lecture`** est une chaîne de caractères ne contenant que les n premiers caractères du fichier. La découpe se fait au nombre de caractères indiqués et pas relativement aux marqueurs **`\t`** ou **`\n`**. Et on démarre toujours du premier caractère du fichier (qui est le numéro 0 chez Python).
`lecture=fich.readlines()`	La variable **`lecture`** est une liste contenant chacune des lignes du fichier. Le fichier est donc découpé selon les marqueurs **`\n`** et les lignes sont sauvegardées dans une liste. C'est en gros la même chose que **`fich.read()`** mais où l'on a découpé le fichier par ligne.
`lecture=fich.readline()`	La variable **`lecture`** est alors une chaîne de caractères contenant la ligne en cours. Avec cette fonction, le fichier est lu ligne par ligne, chacune à son tour. C'est une version étape par étape de la fonction **`readlines()`**.

Exemple 2.2 :

```
>>> fichier=open("Aire.txt","r")
>>> lecture=fichier.read()
>>> fichier.close()
>>> print(lecture)
```

Exemple 2.3 :

```
def Rayon(chem):
    fichier=open(chem,"r")
    L=fichier.readlines()
    fichier.close()
    R=[np.sqrt(float(L[0].strip()[-18:-1])/np.pi)]
    l=[R[0]]
    for k in range(1,len(L)) :
        l=l+[float(L[k].strip()[-18:-1])]
        R=R+[np.sqrt(sum(l)/np.pi)]
    return(R)
```

Une fois le fichier lu, on obtient une chaîne de caractères brute. Il faut pouvoir ensuite la manipuler pour en extraire les informations qui nous intéressent. On donne ici les dernières commandes de manipulation de chaînes de caractères qui nous manquaient. La liste n'est pas exhaustive et d'autres commandes sont possibles. Mais celle-ci nous suffiront.

Commandes de manipulations de chaînes de caractères

Commande	Description
`Chaine.strip()`	Permet d'effacer les éventuels marques de mises en formes aux extrémités de la chaîne de caractères. Cette fonction se décline en une version à droite seulement `rstrip()` et une version à gauche `lstrip`.

`Chaine.split("mot")`	Permet de scinder la chaîne de caractères selon toutes les occurrences de la sous-chaîne `"mot"` pour renvoyer un n-uplet contenant tous les bouts de la chaîne après découpage. Par défaut, `Chaine.split()` découpe la chaîne selon les caractères de séparation `\t` et `\n`.
`Chaine.upper()`	Modifie tous les caractères minuscules par leur version majuscule. Tous les autres caractères ainsi que les caractères déjà majuscules sont laissés intacts.
`Chaine.lower()`	Modifie tous les caractères majuscules par leur version minuscule. Tous les autres caractères ainsi que les caractères déjà minuscules sont laissés intacts.
`Chaine.isalpha()`	Renvoie le booléen `True` si la chaîne ne contient que des lettres (majuscules ou minuscules) et rien d'autre, et `False` dans le cas contraire. La chaîne ne doit contenir aucun espace ou saut de ligne. Il ne doit y avoir que des lettres et rien d'autre.
`Chaine.isdigit()`	Renvoie le booléen `True` si la chaîne ne contient que des chiffres et rien d'autres, et `False` dans le cas contraire. Il ne doit y avoir que des entiers, pas de points ni aucune ponctuation.
`chaine.join(Chaine)`	Insère la chaîne de caractères `chaine` entre chaque caractère de la chaîne de caractères `Chaine`.
`Chaine.replace("mot","MOT",k)`	Remplace les k premières occurrences de `"mot"` de la chaîne `Chaine` par la chaîne `"MOT"`. Si k dépasse le nombre d'occurrences de `"mot"`, elles seront toutes remplacées. Si k est omis, toutes les occurrences de `"mot"` sont remplacées.
`Chaine.count("mot",i,j)`	Compte le nombre d'occurrences de `"mot"` en entier entre les caractères d'index i et $j-1$ de `Chaine`. Si j est omis, compte en partant de l'index i jusqu'à la fin de la chaîne. Si i et j sont omis, compte sur l'ensemble de la chaîne.
`Chaine.find("mot",i,j)`	Renvoie -1 si `"mot"` n'apparaît pas en entier entre les caractères i et $j-1$ de `Chaine`. Si `"mot"` apparaît, renvoie l'index de la première lettre dans la première occurrence de `"mot"` dans `Chaine`.

Exemple 2.4 :

```
>>> C="aBcDeFg\n"
>>> print(C)
aBcDeFg

>>> C.strip()
'aBcDeFg'
>>> C
'aBcDeFg\n'
>>> C=C.strip()
>>> print(C)
aBcDeFg
>>> C.upper()
'ABCDEFG'
>>> C
'aBcDeFg'
>>> C.isalpha()
True
>>> "0".join(C)
'a0B0c0D0e0F0g'
>>> C
'aBcDeFg'
>>> "0".join(C).isdigit()
False
>>> C=C.replace("cDe","bAb")
>>> C
'aBbAbFg'
>>> C.lower().count("b")
3
```