1 Un exemple de manipulation de fichiers

On peut par exemple écrire une procédure `merge` qui prend 3 arguments de type chaîne de caractères (`f1`, `f2` et `f3`) contenant des noms de fichiers : en supposant que les fichiers `f1` et `f2` contiennent des lignes triées par ordre alphabétique (selon l'ordre `ascii`), la procédure `merge` crée le fichier `f3` contenant l'union des lignes de `f1` et `f2`, également triées.

```
>>> def merge(f1,f2,f3) :
...   a = open(f1)
...   b = open(f2)
...   c = open(f3,"w")
...   x = a.readline()
...   y = b.readline()
...   while x != "" and y != "" :  # une ligne à voir dans chaque fichier
...     if x < y :
...       c.write(x)               # on écrit la plus petite des 2 lignes
...       x = a.readline()         # et on lit la suite du fichier utilisé
...     else :
...       c.write(y)               # idem
...       y = b.readline()
...   while x != "" :              # on termine le fichier non épuisé,
...     c.write(x)
...     x = a.readline()
...   while y != "" :              # un seul des 2 derniers while est exécuté
...     c.write(y)
...     y = b.readline()
...   a.close()
...   b.close()
...   c.close()
...   print("Le fichier " + f3 + " est rempli.")
```

En supposant que le fichier `toto.txt` contienne

```
alain
marc
pierre
robert
yves
zorro
```

et que `titi.txt` contienne

```
bernard
joe
luc
thomas
```

la commande `merge("toto.txt","titi.txt","tutu.txt")` fabrique le fichier `tutu.txt` contenant :

```
alain
bernard
joe
luc
marc
pierre
```

```
robert
thomas
yves
zorro
```

2 Itération en lecture de fichier

Ajoutons qu'on peut faire un `for` sur un fichier : cela énumère les lignes du fichier :

```
>>> f = open("gamin.txt")
>>> for ligne in f :
...     print(len(ligne))
...
14
15
13
15
>>> f.close()
```

On peut par exemple écrire d'une procédure `more(fichier)` qui affiche à l'écran le contenu du fichier, par pages de 24 lignes :

```
>>> def more(f):
...   hauteur=24
...   r=open(f)
...   vues=0
...   for l in r:
...     print(l[:-1])   # enlève le passage à la ligne de l
...     if vues < hauteur:
...       vues=vues+1
...     else:
...       s=input("suite... ")
...       vues=0
...   r.close
```

La dernière ligne, si elle ne se termine pas par un retour chariot, voit son dernier caractère disparaître. Pour bien faire, il faudrait tester si la dernière ligne contient une fin de ligne (`"\n"`) ou non.

3 Les modules en Python

En ce point du cours, nous avons étudié la majorité des structures de données classiques (manquent les *arbres* et les *graphes* qui relèvent de cours plus avancés) et la quasi-totalité des structures de contrôle (le *traitement d'exceptions* et la *récursivité* relèvent de cours plus avancés). On peut même affirmer que quel que soit le problème posé, s'il existe un programme qui le résoud alors nous avons vu tous les éléments pour le faire.

Il existe cependant des questions « standard » qu'il serait peu productif de résoudre chacun pour soi : ces questions sont tellement courantes que les programmes qui les résolvent sont stockés dans des *bibliothèques*(*libraries* en anglais). En Python, un telle bibliothèque est constituée de nombreux *modules* et un module est constitué de plusieurs fonctions (ou procédures) déjà programmées. Dans un module, on prend soin de regrouper les fonctions qui permettent de gérer un type de problème donné, bien souvent ce sont simplement les opérations associées à une structure de données adaptée au problème. Pour la biologie, il existe une bibliothèque de modules très riche : BioPython.

On peut écrire soi-même des modules. Lorsque l'on veut écrire un gros logiciel, c'est généralement une bonne idée de le découper en modules (qui seront du même coup réutilisables par ailleurs), en suivant généralement le principe « au moins un module par nouvelle structure de donnée ». Il suffit alors « d'importer » les modules pour utiliser les fonctions qui y ont été programmées.

4 Gestion des fichiers : le module « os »

Pour utiliser un module, il faut d'abord le charger avec la commande `import`. Parmi les modules vraiment utiles, il y a celui qui permet de gérer l'arborescence des fichiers de l'ordinateur. Il se nomme `os`, comme « operating system », et nous nous servirons de ce module comme exemple.

```
>>> import os
```

Remarquez que, exceptionnellement, `os` n'est pas écrit comme une chaîne de caractère (donc pas entre guillemets) et pourtant `import` ne considère pas `os` comme un nom de variable... Avant d'utiliser `os` pour gérer le système de fichiers, il faut savoir :

- qu'il peut y avoir plusieurs disques durs sur un ordinateur, ou qu'un disque dur peut être partagé en plusieurs partitions. Sous Mac ou un système Unix comme Linux cette séparation des disques et des partitions est gérée pour qu'on ait l'impression de n'avoir qu'un seul disque. Sous Windows, ces partitions sont vues comme plusieurs « lecteurs » différents numérotés `A`, `B`, `C`, `D`, etc. En général `A` et `B` sont présents pour des raisons historiques et sont réservés à deux lecteurs de disquettes. Le disque `C` est généralement celui sur lequel le système Windows est installé et les suivants sont des disques de données ou des lecteurs/graveurs de CD, DVD, *etc.*
- Un disque est organisé en arbre avec des répertoires(directories en anglais ou « dossiers » pour les utilisateurs naïfs) qui peuvent contenir des sous-arbres et des fichiers (files) qui contiennent les « vraies » données et sont nécessairement des feuilles de l'arbre.
- Un nœud de l'arbre est repéré par sont chemin depuis la racine, qui est souvent appelé son adresse.
- Compte tenu de la taille de cet arbre et de la longueur des chemins, et du fait qu'on travaille longtemps sous un même répertoire, il est pratique d'avoir un répertoire dit « courant ». Cela permet de ne donner que les adresses à partir du répertoire courant.

Pour utiliser une fonction ou une procédure d'un module, il faut la préfixer par le nom du module (une fois qu'il a été importé). Par exemple `getcwd` est une fonction qui retourne l'adresse du répertoire dans lequel on est en train de travailler (« cwd » pour *current working directory*).

```
>>> os.getcwd()
'/home/bernot'
```

Le module `os` offre aussi une procédure qui crée un répertoire, appelée `mkdir` (make directory) :

```
>>> os.mkdir("test")
```

Pour se déplacer dans les répertoires, et donc changer l'adresse du répertoire de travail, on utilise la procédure `chdir` (change directory) et l'on peut lister le contenu d'un répertoire avec la fonction `listdir`, qui retourne la liste des éléments contenus dans le répertoire :

```
>>> os.chdir("test")
>>> os.listdir("/home/bernot/test")
[]
>>> f=open("toto.txt")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IOError: [Errno 2] No such file or directory: 'toto.txt'
>>> f=open("toto.txt","w")
>>> f.close()
>>> os.listdir("/home/bernot/test")
['toto.txt']
```

Il existe des abréviations : la chaîne `"."` remplace le répertoirede travail courant et la chaîne `".."` désigne son répertoire parent :

```
>>> os.listdir(".")
['toto.txt']
>>> os.mkdir("repfils")
```

```
>>> os.listdir(".")
['repfils', 'toto.txt']
>>> os.chdir("repfils")
>>> os.getcwd()
'/home/bernot/test/repfils'
>>> os.listdir(".")
[]
>>> os.listdir("..")
['repfils', 'toto.txt']
```

D'autres fonctions ou procédures utiles pour la gestion du système de fichiers sont :

- `rename` prend en argument les chemins source et cible : peut déplacer dans l'arbre, aussi bien un fichier qu'un sous-arbre complet
- `remove` prend en argument le chemin d'un fichier et le supprime
- `rmdir` comme `remove`, mais pour un répertoire vide

On remarque que la liste obtenue par `listdir` mélange des répertoires et les « vrais » fichiers sans moyen de les distinguer. Le module `os` contient un « sous-module » `os.path` qui est automatiquement importé avec `os` et fournit d'autres fonctions bien utiles :

- `isfile` dit si le chemin pointe sur une fichier standard
- `isdir` ... si c'est un répertoire
- `exists` dit si le chemin donné en argument existe (fonction booléenne)
- `getsize` taille en octets du fichier désigné par le chemin donné en argument
- `abspath` prend un chemin en entrée et retourne sa version en adresse absolue (c'est-à-dire en partant de la racine de l'arbre des répertoires)
- `basename` fournit le dernier nom du chemin
- `dirname` le contraire

```
>>> os.isfile("toto.txt")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'module' object has no attribute 'isfile'
>>> os.path.isfile("toto.txt")
True
>>> os.path.isdir("toto.txt")
False
>>> os.path.basename("toto/tutu/titi")
'titi'
>>> os.path.dirname("toto/tutu/titi")
'toto/tutu'
```

5 Parcours d'une arborescence de fichiers

Un exemple de programme utilisant le module `os` : calculer le nombre de « feuilles » (c'est-à-dire de fichiers qui ne sont pas des répertoires) dans une arborescence de fichiers. On donne en argument de la fonction `nbfeuilles` l'adresse de la racine de l'arborescence à parcourir et la fonction va parcourir tout l'arbre, répertoire après répertoire, pour compter le nombre de feuilles.

L'idée de ce programme est de mémoriser deux choses : d'une part le nombre de feuilles déjà rencontrées au cours du programme (initialisation à 0 donc), et d'autre part la liste des adresses qu'il va falloir explorer (initialisation à la liste de longueur 1 ne contenant que l'adresse donnée en argument).

```
>>> import os
>>> def nbfeuilles(x):
...   if not ( os.path.exists(x) ) :
...     print("L'adresse %s n'existe pas !!" % x)
...   else :
...     n=0      #nbr de feuilles rencontrées
...     todo=[x] #on veut explorer l'adresse x
...     while todo != [] : #tant qu'il reste des adresses à explorer
```

```
...       if not ( os.path.isdir(todo[0]) ) :
...           n = n + 1 #pas un répertoire donc une feuille de plus
...       else :
...         for y in os.listdir(todo[0]) :
...           z = todo[0] + "/" + y   #adresse de chaque fils du répertoire
...           todo = todo + [z] #il faudra explorer cette adresse
...       todo = todo[1:] #adresse traitée, on l'enlève
...     return(n)
```