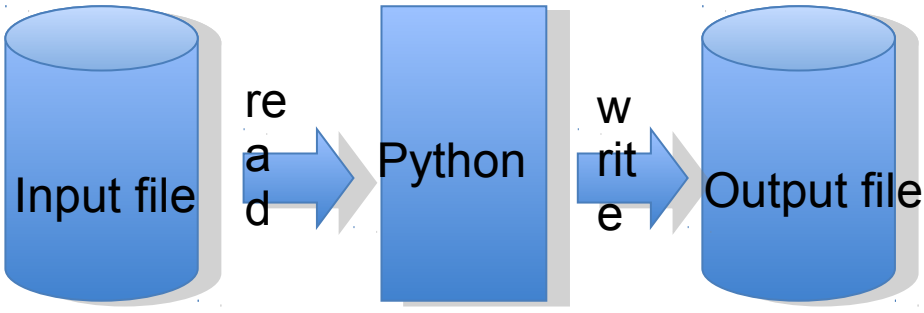
Chapitre 5

Fichiers

1 - De l'utilité des fichiers

Lorsqu'on doit traiter un grand flot de données comme c'est le cas en bioinformatique (tant en génomique qu'en post-génomique) il est indispensable de stocker ces données dans des fichiers (ou des bases de données) puis de lire ces fichiers (ou les fichiers extraits des bases de données) directement à l'aide de programme.
C'est ce que nous allons apprendre à faire dans ce chapitre. Python est également capable de réaliser l'interface avec des SGBD mais nous n'aborderons pas cet aspect dans ce cours.

Pour travailler avec des fichiers que ce soit en lecture ou en écriture, il est nécessaire de créer un lien logique entre le programme (code source en python) et les données. On parle d'ouverture de fichier (création d'un lien entre les données et le programme), ce lien est détruit lors de la fermeture du fichier de données.



2 - Ouverture d'un fichier

Sous Python l'accès aux fichiers est assuré via des *objets-fichiers* que l'on crée à l'aide de la fonction `open()`. Cette fonction crée le lien logique entre le programme et le fichier de données, qui peut être soit en mode lecture, soit en mode écriture.

- Si le fichier de données est ouvert en *mode lecture*, le programme pourra y lire des informations (comme il peut le faire sur l'entrée standard – stdin – le clavier) .
- S'il est ouvert en *mode écriture*, il pourra y écrire des informations (comme il peut le faire sur la sortie standard – stdout – ou la sortie erreur – stderr - l'écran).

Syntase : `open(filename [,mode])` par défaut le mode d'ouverture est la lecture seule.

Exemple :

```
obFichier = open('MonFichier','a')
```

`obFichier` : *objet_fichier* pour accéder au fichier via un programme python
`MonFichier` : *nom du fichier* de données à ouvrir en accès, MonFichier doit être dans le répertoire courant, sinon il faut indiquer le chemin d'accès ou changer de répertoire avant la fonction open() – avec la méthode chdir (voir ci-dessous).
`a` : *mode d'ouverture* du fichier 'a'pour append = mode écriture avec ajout des nouvelles données en fin de fichier.

*Mode d'ouverture**Signification*`r`Lecture seule`w`Ecriture (si fichier existe déjà alors il sera écrasé)`a`Ecriture en fin de fichier`r+`Lecture et écriture`w+`Lecture et écriture (fichier préexistant écrasé)`a+`Lecture et écriture (écriture en fin de fichier).

3 - Ecriture séquentielle dans un fichier

Une fois que le fichier a été ouvert dans l'un des modes d'écriture il suffit d'y écrire dedans à l'aide de la méthode `write`() appliquée à l'*objet-fichier* crée par la fonction `open`().

```
>>> obFichier = open('MonFichier','a')
>>> obFichier.write('Bonjour Fichier !')
>>> obFichier.write("Quel beau temps aujourd'hui.")
>>> obFichier.close()
```

Si le fichier existe déjà les 2 lignes seront rajoutés en fin de fichier, sinon le fichier sera crée et les 2 lignes écrites en début d fichier. Le lien logique entre le fichier et le programme est ensuite coupé. Attention il ne faut pas confondre le lien logique `obFichier` et le fichier physique `MonFichier` stocké en mémoire permanente et qui peut être visualisé avec un éditeur de texte.

```
>>> obFichier = open('MonFichier','w')
>>> obFichier.write('Bonjour Fichier !')
>>> obFichier.close()
```

Le fichier sera crée et la ligne écrite en début de fichier. **Attention si le fichier existait déjà il sera écrasé et remplacé par le nouveau fichier** constitué de la ligne 'Bonjour Fichier !'.

4 - Lecture séquentielle d'un fichier

On va ré-ouvrir le fichier en mode lecture maintenant pour pouvoir accéder aux informations que l'on y a stocké.

```
>>> ofi = open('MonFichier','r')
>>> t = ofi.read()
>>> print t
Bonjour Fichier !
>>> ofi.close()
```

Si on utilise la méthode `read()` sans argument la totalité du fichier est lue. La méthode read() retourne les donnée dans une chaîne de caractères.

Si le fichier à ouvrir n'existe pas le programme renvoie une erreur.

```
>>> ofi = open('monFichier','r')
IOError :[Errno 2] No such file or directory: 'monFichier'
```

Attention si le fichier n'est pas dans le répertoire courant alors il faut indiquer son chemin d'accès.

```
>>> ofi = open('/home/devauchelle/mesDonnees/Monfichier')
```

Mais le Python qui est un langage de script permet de **changer le répertoire courant** pendant l'exécution du programme.

```
>>>  from os import chdir
>>> chdir("/home/devauchelle/mesDonnees »)
>>> ofi = open('monFichier','r')
```

Le répertoire courant est désormais `mesDonnees` et même si le programme a été lancé depuis `/home/devauchelle/coursPython`, le programme pourra ouvrir `'MonFichier'` qui est dans le répertoire `mesDonnees` puisque c'est le répertoire courant (grâce à la fonction `chdir()` – change directory - du module `os` - operating sytem)

5 - Les fonctions de lecture / écriture

Méthode	*Description*
`read()`	le fichier jusqu'à EOF et renvoie une chaîne de caractères
`read(n)`	Lit `n` caractères dans le fichier à partir de la position courante
`readline()`	Lit une ligne du fichier jusqu'à \n et renvoie la chaîne
`readlines()`	Lit l'ensemble des lignes d'un fichier et les met dans une liste
`xreadlines()`	Méthode efficace pour la lecture de l'ensemble des lignes d'un gros fichier
`write(string)`	Ecrit la chaîne dans le fichier

6 - Enregistrement et restitutions de valeurs/types

L'argument de la méthode `write()` doit être une chaîne de caractères. Avec ce que nous avons appris jusqu'à présent, nous ne pouvons donc enregistrer d'autres types de valeurs qu'en les transformant d'abord en chaînes de caractères grâce à la fonction intégrée (*built in function*) `str()` :

```
>>> x = 52
>>> f.write(str(x))
```

En fait, il existe d'autres possibilités pour convertir des valeurs numériques en chaîne de caractères, que nous avons vu lors du formatage des chaînes :

```
>>> nom= 'toto'
>>> n=10
>>> print "le nom du %dième élève est %s" % (nom,n)
```

Une des autres possibilités offerte par Python est d'utiliser le module *pickle* (en anglais conserver) qui permet de restituer à la fois la donnée et le type de la donnée stockée dans le fichier, pourvu qu'elle ait été stocké également à l'aide du module *pickle*. Exemple :

```
>>> import pickle
>>> a = 5
>>> b = 2.83
>>> c = 67
>>> # enregistrement dans un fichier grâce au module pickle
>>> f = open('monFichier','w')
>>> pickle.dump(a,f)     # enregistrement de a dans le fichier
>>> pickle.dump(b,f)     # enregistrement de b dans le fichier
>>> pickle.dump(c,f)     # enregistrement de c dans le fichier
>>> f.close()
>>> # lecture de données stockées, grâce au module pickle
>>> f = open('monFichier','r')
>>> t = pickle.load(f)
>>> print t, type(t)     # la valeur et le type sont restitués
5, <type 'int'>
>>> t = pickle.load(f)
>>> print t, type(t)     # la valeur et le type sont restitués
2.83, <type 'float'>
>>> t = pickle.load(f)
>>> print t, type(t)     # la valeur et le type sont restitués
65, <type 'int'>
```

7 - Arguments de la ligne de commande

On passe souvent des arguments ou options en ligne de commande à un programme. Elles sont disponibles dans une liste qui s'appelle `argv` (import `sys`).

`sys.argv[0]` est le nom du script
`sys.argv[1]` est le premier argument
`sys.argv[2]` est le second etc...

Cette fonctionnalité est souent intéressante, par exemple pour passer à un programme le nom des fichiers d'entrée sur lesquels il va travailler.
Exemple

```
myprog.py nom_fichier1 nom_fichier2
f1= open(argv[1],'r')    #nom_fichier1 ouvert en lecture
f2 = open(argv[2],'w')   #nom_fichier2 ouvert en écriture
```

8 - Gestion des exceptions

Les exceptions sont les opérations que génère un compilateur ou un interpréteur lorsqu'une erreur est détectée lors de l'exécution d'un programme. En règle générale l'éxécution du programme s'arrête et un message d'erreur plus ou moins explicite est affiché.
Exemple :

```
>>> ofi = open('monFichier','r')
IOError :[Errno 2] No such file or directory: 'monFichier'
```

En plus du type d'erreur, le message comporte aussi une indication sur l'endroit du script où se produit l'erreur.

Dans les langages de niveau élevé comme Python, il est possible d'associer un mécanisme de surveillance à tout un ensemble d'instructions et donc de simplifier la gestion des erreurs qui peuvent se produire dans n'importe laquelle des ces instructions sous surveillance.

```
filename = raw_input('Entrez un nom de fichier : ')
try :
     f=open(filename,'r')
except :
     print "le fichier",filename,"est introuvable"
```

Chapitre 6

Expressions rationnelles

1 – Qu'est ce qu'une expression rationnelle ?

En anglais *regular expression* d'où e*xpressions régulières* par angliscisme. Les *expressions rationnelles* sont des expressions qui décrivent un modèle de chaîne de caractères (patron ou *pattern* en anglais). Ainsi plusieurs chaînes peuvent posséder le même patron et une chaîne peut posséder plusieurs patrons.

Exemple :

L'expression rationnelle qui décrit ex-aequo, ex-équo et et ex-equo est ex-(a.|e|é)quo

où (ae|e)quo veut dire aequo ou equo

(a?|e)quo veut dire aequo ou equo (car a? veut dire a présent 0 ou 1 fois)

donc ex-(a?e|é)quo désigne ex-aequo, ex-equo, ex-équo et ex-aéquo.

Les shells Unix (bash,csh,sh ...) utilisent nativement des expressions régulières pour les recherches de fichiers (grep). De même les programmes développés sous Unix tels que sed, awk, Perl, Python et bien d'autres utilisent la puissance des expressions rationnelles.

2 – Syntaxe des expressions rationnelles

Une expression rationnelle est une chaîne de caractères qui contient des caractères, des caractères spéciaux qu'il faut protéger par le caractère spécial \ , et des métacaractères tels que *, ? , etc...

Caractère	*Signification*
.	Remplace n'importe quel caractère
\	Sert à dé-spécialiser un caractère
[abc]	Les crochets servent à délimiter un choix de caractères
[a-zA-Z]	Les tirets à l'intérieur des crochets désignent un ens. de caractères consécutifs.
[^aeiou]	^ à l'intérieur des crochets et en 1° position exprime un choix négatif
|	Alternative entre deux caractères ou deux expressions
$	Correspond au début de la ligne
^	Correspond à la fin de la ligne
(m,n)	Caractère ou expression qui précède les accolades peut figurer entre m et n fois
.	Caractère ou expression qui précède est présent 0 ou 1 fois
*	Caractère ou expression qui précède est présent 0 ou plusieurs fois
+	Caractère ou expression qui précède est présent 1 ou plusieurs fois

Raccourci	*Signification*
\W	Tous les caractères non alphanumériques
\w	Tous les caractères alphanumériques
\S	Tous les caractères qui ne sont pas des caractères d'espacement
\s	Tous les caractères d'espacement [\t\n\r\f\v]

\D	Tous les caractères non numériques
\d	Tous les caractères numériques [0-9]
\b	Chaîne vide à la fin mot (fin de mot)

3 - Exemples d'expressions rationnelles

Numéro de téléphone sur une ligne:
`'\d\d[\.\-\\]\d\d[\.\-\\]\d\d[\.\-\\]\d\d[\.\-\\]\d\d\'`
`'(\d\d[\.\-\\])?4?\d\d\'`
`'(\d\d[\.\-\\])?4?\d\d\$'`
Mots se terminant par *aient*
`'\w*aient\b'`
Identificateurs de variables python
`'^[a-zA-Z][a-zA-Z0-9]*'`
Lien Hypertexte dans un fichier htlm
<a HREF = " kde-linux.html "> L'environnement graphique KDE
le lien est sur le texte : L'environnement graphique KDE
le lien est : kde-linux.html
l'expression régulière correspond à la balise html des liens hypertexte est :
`'^<[a|A]\s(HREF|href)>.*<//[a|A]>\b'`
`'^<[a|A]\s(HREF|href)>(.*)<//[a|A]>\b'`

attention les expressions regulières indiquées dans ce cours sont à vérifier !!!

4 - Le module re

Le module re est le module qui contient les fonctions permettant les recherches d'expressions rationnelles (`regexp` pour *regular expression*).

Fonction *Description* `search(regexp,string)` Renvoie un `MatchObject` correspondant à la recherche de `regexp` dans la chaîne `string`. Recherche le motif où qu'il soit dans la chaîne. `match(regexp,string)` Renvoie `None` si la chaîne ne correspond pas et un `MatchObject` sinon.

Recherche le motif en début de chaîne. `findall(regexp,string)` Renvoie une liste de tous les éléments trouvés `sub(regexp,repl,string)` Remplace toutes les sous-chaînes qui correspondent à `regexp` par `repl`. `split(regexp,string)` Découpe string selon toutes les sous-chaînes correspondant à l'expression et renvoie la liste obtenue. `split(regexp,str,maxsplit)` Idem mais s'arrête après `maxsplit` découpages.

Exemples :

```
>>> s = 'abababeaba'
>>> import re
>>> rgx = 'a(be)+'
>>> re.search(rgx,s)
<_sre.SRE_Match object at 0x94760>
>>> rge = 'abc'
>>> re.search(rge,s)     # il ne se passe rien
>>> re.match(rge,s)      # il ne se passe rien
>>> re.match(rgx,s)      # il ne se passe rien

>>> rgx = '\d+'
>>> s = 'Nous étions 500 braves, le 15 aout 1998'
>>> re.findall(rgx,s)
[500,15,1998]
```

λ5 - La classe MatchObject

La classe `MatchObject` définit les méthodes pour accéder au résultat de la recherche pour accéder au résultat de la recherche.
Les méthodes les plus courantes sont `start`() et `end()` qui donnent les indices de début et de fin de la sous-chaîne trouvée.

```
>>> s = 'abababeaba'
>>> rgx = 'a(be)+'
>>> m = re.search(rgx,s)
>>> m.start()
4
>>> m.end()
7
```

Remarque importante :
La construction d'une expression régulière est longue pour l'interpréteur, donc si c'est une expression régulière que vous devez utiliser plusieurs fois, il faut la compiler.

```
>>> motif1 = re.compile('a(be)+')
```

motif1 est une expression rationnelle compilée sur laquelle on peut appliquer les méthodes des expressions régulières.
Exemple :

```
>>> m = motif1.search(s)
>>> m.start()
4
>>> m.end()
7
```

6 - Les groupes dans les expressions rationnelles

Il arrive souvent qu'on veuille récupérer la valeur d'un mot ou d'un ensemble de mots qui matche une partie de l'expression régulière. Pour cela les expressions offrent la possibilité de créer des groupes de motifs à conserver. Dans l'exemple ci-dessous on veut conserver le suffixe des verbes se terminant par *aient*.

```
>>> s = 'ils allaient et bavardaient en cheminant'
>>> rgx = r'(\w*)aient\b'
```

Le groupe que l'on veut conserver est indiqué entre ()

```
>>> for m in re.finditer(rgx,s) :
...       print m.group(0), m.group(1)
...
allaient all
bavardaient bavard
```

groupe(0) est l'expression entière, groupe(1) correspond au premier groupe demandé (ici un seul).

7 - Documentation sur les expressions régulières

- Documentation en ligne du module re
  http://www.python.org/doc/current/lib/module-re.html
- L'excellent tutorial sur les expressions régulières :
  http://www.amk.ca/python/howto/regex

Exercices du TD3