6 Manipulation de fichiers

1 Utilité des fichiers

Imaginons que nous voulions écrire un programme qui fasse apparaître à l'écran des questions à choix multiple, avec traitement automatique des réponses de l'utilisateur. Comment allons-nous mémoriser le texte des questions elles-mêmes ?

L'idée la plus simple consiste à placer chacun de ces textes dans une variable, en début de programme, avec des instructions d'affectation du genre :

```
a = "Quelle est la capitale du Guatemala ?"
b = "Qui à succédé à Henri IV ?"
c = "Combien font 26 x 43 ?"
        ... etc.
```

Cette idée est malheureusement beaucoup trop simpliste. Tout va se compliquer en effet lorsque nous essayerons d'élaborer la suite du programme, c'est-à-dire les instructions qui devront servir à sélectionner au hasard l'une ou l'autre de ces questions pour les présenter à l'utilisateur. Employer par exemple une longue suite d'instructions `if` ... `elif` ... `elif` ... comme dans l'exemple ci-dessous n'est certainement pas la bonne solution si nous souhaitons traiter un grand nombre de questions :

```
from random import randrange

choix = randrange(1000)

if choix == 1:
    selection = a
elif choix == 2:
    selection = b
elif choix == 3:
    selection = c
        ... etc.
```

La situation se présente déjà beaucoup mieux si nous faisons appel à une liste :

```
liste = ["Qui a vaincu Napoléon à Waterloo ?",
         "Comment traduit-on <informatique> en anglais ?",
         "Quelle est la formule chimique du méthane ?", ... etc ...]
```

On peut en effet extraire n'importe quel élément de cette liste à l'aide de son indice :

```
print(liste[2])          ⟹  "Quelle est la formule chimique du méthane ?"
```

Même si cette façon de procéder est déjà nettement meilleure que la précédente, nous sommes toujours confrontés à plusieurs problèmes gênants :

- La lisibilité du programme va se détériorer très vite lorsque le nombre de questions deviendra important. En corollaire, nous accroîtrons la probabilité d'insérer une erreur de syntaxe dans la définition de cette longue liste. De telles erreurs seront bien difficiles à débusquer.
- L'ajout de nouvelles questions, ou la modification de certaines d'entre elles, imposeront à chaque fois de rouvrir le code source du programme. En corollaire, il deviendra malaisé de retravailler ce même code source, puisqu'il comportera de nombreuses lignes de données encombrantes.
- L'échange de données avec d'autres programmes (peut-être écrits dans d'autres langages) est tout simplement impossible, puisque ces données font partie du programme lui-même.

Cette dernière remarque nous suggère la direction à prendre : il est temps que nous apprenions à **séparer les données et les programmes qui les traitent dans des fichiers différents.**

Pour que cela devienne possible, nous devrons doter nos programmes de divers mécanismes permettant de créer des fichiers, d'y envoyer des données et de les récupérer par la suite.

Les langages de programmation proposent des jeux d'instructions plus ou moins sophistiqués pour effectuer ces tâches. Lorsque les quantités de données deviennent très importantes, il devient d'ailleurs rapidement nécessaire de structurer les relations entre ces données, et l'on doit alors élaborer des systèmes appelés **bases de données relationnelles**, dont la gestion peut s'avérer très complexe. Lorsque l'on est confronté à ce genre de problème, il est d'usage de déléguer une bonne part du travail à des logiciels très spécialisés tels que **PostgreSQL**, **MySQL** ou **SQLite**, etc. Python est parfaitement capable de dialoguer avec ces systèmes, mais nous laisserons cela pour un peu plus tard.

Nos ambitions présentes sont plus modestes. Nos données ne se comptent pas encore par centaines de milliers, aussi nous pouvons nous contenter de mécanismes simples pour les enregistrer dans un fichier de taille moyenne, et les en extraire ensuite.

2 Travailler avec des fichiers

L'utilisation d'un fichier ressemble beaucoup à l'utilisation d'un livre. Pour utiliser un livre, vous devez d'abord le trouver à l'aide de son titre puis l'ouvrir. Lorsque vous avez fini de l'utiliser, vous le refermez. Tant qu'il est ouvert, vous pouvez y lire des informations diverses ou y écrire des annotations, mais généralement vous ne faites pas les deux à la fois. Dans tous les cas, vous pouvez vous situer à l'intérieur du livre, notamment en vous aidant des numéros de pages. Vous lisez la plupart des livres en suivant l'ordre des pages, mais vous pouvez aussi décider de consulter n'importe quel paragraphe dans le désordre.

Tout ce que nous venons de dire des livres s'applique également aux fichiers informatiques. Un fichier se compose de données enregistrées sur un disque dur, une disquette, une clef USB ou un CD. Vous y accédez grâce à son nom (lequel peut inclure aussi un nom de répertoire). Vous pouvez en première approximation considérer le contenu d'un fichier comme une suite de caractères, ce qui signifie que vous pouvez traiter ce contenu, ou une partie quelconque de celui-ci, à l'aide des fonctions servant à traiter les chaînes de caractères [1].

2.1 Répertoire courant

Pour simplifier les explications qui vont suivre, nous indiquerons seulement des noms simples pour les fichiers que nous allons manipuler. Si vous procédez ainsi dans vos exercices, les fichiers en question seront créés et/ou recherchés par Python dans le **répertoire courant**, c'est-à-dire le répertoire de travail par défaut. Il s'agit habituellement du répertoire où se trouve le script lui-même, sauf si vous lancez ce script depuis la fenêtre d'un shell *IDLE*, auquel cas le répertoire courant est défini au lancement de *IDLE* (sous *Windows*, la définition de ce répertoire fait partie des propriétés de l'icône de lancement).

Le module `os` contient toute une série de fonctions permettant de dialoguer avec le système d'exploitation (`os` = *operating system*), quel que soit celui-ci. La fonction `getcwd()` de ce module renvoie le nom du répertoire courant (`getcwd` = *get current working directory*) :

```
>>> from os import getcwd
>>> getcwd()
'/Users/Jules/Documents'
```

Ainsi, dans l'exemple ci-dessus, le répertoire courant est `/Users/Jules/Documents` qui correspond au répertoire `Documents` de l'utilisateur `Jules`.

Si vous souhaitez forcer Python à changer son répertoire courant, afin que celui-ci corresponde à vos attentes, vous devrez utiliser les commandes suivantes en début de session :

```
>>> from os import chdir
>>> cwd = "/Users/Jules/OC-Informatique/Exercices"
>>> chdir(cwd)
```

1. En toute rigueur, vous devez considérer que le contenu d'un fichier est une suite d'**octets**. La plupart des octets peuvent effectivement être représentés par des caractères, mais l'inverse n'est pas vrai : nous devrons donc plus loin opérer une distinction nette entre les chaînes d'octets et les chaînes de caractères.

La fonction `chdir()` du module `os` provoque le changement de répertoire (`chdir` = *change directory*). Dans l'exemple ci-dessus, le répertoire visé est le répertoire `/Users/Jules/OC-Informatique/Exercices`. Même si vous travaillez sous *Windows* (où ce n'est pas la règle), vous pouvez utiliser cette syntaxe. Python effectuera les conversions nécessaires, suivant que vous travaillez sous *Mac OS*, *Linux*, ou *Windows*[2].

- Vous avez également la possibilité d'insérer ces commandes en début de script, ou encore d'indiquer le chemin d'accès absolu dans le nom des fichiers que vous manipulez, mais cela risque peut-être d'alourdir l'écriture de vos programmes.
- Choisissez de préférence des noms de fichiers courts. Évitez dans toute la mesure du possible les caractères accentués, les espaces et les signes typographiques spéciaux. Dans les environnements de travail de type Unix (MacOS, Linux, BSD ...), il est souvent recommandé aussi de n'utiliser que des caractères minuscules.

Une chaîne de caractères comme `cwd` qui identifie l'emplacement d'une ressource dans l'arborescence du système de fichiers d'un ordinateur est appelé son **chemin d'accès**. Ce dernier peut s'exprimer de manière **relative** en décrivant la succession de répertoires menant à la ressource à partir du répertoire courant ou de manière **absolue** en décrivant l'emplacement de la ressource à partir du **répertoire racine**. Ce répertoire, appelé également **racine** (ou *root* en anglais), est le dossier de plus bas niveau de l'arborescence. Contrairement aux autres répertoires du système de fichiers, le répertoire racine ne se trouve lui-même dans aucun autre répertoire.

Ainsi, le chemin d'accès d'un fichier texte situé dans le répertoire courant de l'exemple ci-dessus sera :

- `monFichier.txt` (en écriture relative)
- `/Users/Jules/OC-Informatique/Exercices/monFichier.txt` (en écriture absolue)

2.2 Ecriture séquentielle dans un fichier

Sous Python, l'accès aux fichiers est assuré par l'intermédiaire d'un objet-interface particulier, que l'on appelle **objet-fichier**. On crée cet objet à l'aide de la fonction intégrée `open()`[3]. Celle-ci renvoie un objet doté de méthodes spécifiques, qui vous permettront de lire et écrire dans le fichier.

L'exemple ci-après vous montre comment ouvrir un fichier en écriture, y enregistrer deux chaînes de caractères, puis le refermer. Notez bien que si le fichier n'existe pas encore, il sera créé automatiquement. Par contre, si le nom utilisé concerne un fichier préexistant qui contient déjà des données, les caractères que vous y enregistrerez viendront s'ajouter à la suite de ceux qui s'y trouvent déjà. Vous pouvez faire tout cet exercice directement à la ligne de commande :

```
>>> obFichier = open('Monfichier.txt','a', encoding = 'utf-8')
>>> obFichier.write('Bonjour, fichier !')
18
>>> obFichier.write("Quel beau temps, aujourd'hui !")
30
>>> obFichier.close()
```

Analysons en détail les quelques lignes qui composent cet exemple :

- La première ligne crée l'**objet-fichier** `obFichier`, lequel fait référence à un fichier véritable (sur disque ou disquette) dont le **nom** sera `Monfichier`. Ne confondez pas le **nom du fichier** avec le **nom de l'objet-fichier** qui y donne accès ! À la suite de cet exercice, vous pouvez vérifier qu'il s'est bien créé sur votre système (dans le répertoire courant) un fichier dont le nom est `Monfichier.txt` et dont vous pouvez visualiser le contenu à l'aide d'un éditeur quelconque.
- La fonction `open()` attend trois arguments, qui doivent tous être des chaînes de caractères. Le premier argument est le nom du fichier à ouvrir, le second est le mode d'ouverture et le troisième le type d'encodage avec lequel il faut manipuler le fichier. Le mode d'ouverture `'a'` indique qu'il faut ouvrir le fichier en mode « ajout » (**append**), ce qui signifie que les données à enregistrer doivent être ajoutées à la fin du fichier, à la suite de celles qui s'y trouvent éventuellement déjà. Nous aurions pu utiliser aussi le mode `'w'` (pour **write**), mais lorsqu'on utilise ce mode, Python crée toujours un nouveau fichier (vide), et l'écriture des données commence à partir du début de ce nouveau fichier. S'il existe déjà un fichier de même nom, celui-ci

2. Dans le cas de *Windows*, vous pouvez également inclure dans ce chemin la lettre qui désigne le périphérique de stockage où se trouve le fichier. Par exemple : `C:/home/Jules/OC-Informatique/Exercices`
3. Une telle fonction, dont la valeur de retour est un objet particulier, est souvent appelée **fonction-fabrique**.

est effacé au préalable. Le type d'encodage `'utf-8'` signifie que les textes enregistré dans le fichier devront être encodé selon la norme UTF-8. Comme nous le verrons par la suite, l'argument `encoding` n'est pas obligatoire car la fonction `open()` dispose d'un paramétrage par défaut qui encode automatiquement les chaînes à enregistrer en suivant la norme par défaut du système d'exploitation. Il se peut néanmoins que l'on veuille enregistrer des textes en leur appliquant un encodage différent de celui qui est prévu par défaut, ne serait-ce que pour nous assurer que l'encodage réalisé soit bien celui que nous voulons ; l'option `encoding` prend dès lors tout son sens.

- La méthode `write()` réalise l'écriture proprement dite. Les données à écrire doivent être fournies en argument. Ces données sont enregistrées dans le fichier les unes à la suite des autres (c'est la raison pour laquelle on parle de fichier **à accès séquentiel**). Chaque nouvel appel de `write()` continue l'écriture à la suite de ce qui est déjà enregistré.
- La méthode `close()` referme le fichier. Celui-ci est désormais disponible pour tout usage.

2.3 Lecture séquentielle dans un fichier

Vous allez maintenant rouvrir le fichier, mais cette fois en lecture, de manière à pouvoir y relire les informations que vous avez enregistrées dans l'étape précédente. La méthode `read()` lit les données présentes dans le fichier et les transfère dans une variable de type chaîne de caractères (**string**) . Si on utilise cette méthode sans argument, la totalité du fichier est transférée :

```
>>> ofi = open('Monfichier.txt', 'r', encoding = 'utf-8')
>>> t = ofi.read()
>>> print(t)
Bonjour, fichier !Quel beau temps, aujourd'hui !
>>> ofi.close()
```

Analysons en détail les quelques lignes qui composent cet exemple :

- Le fichier que nous voulons lire s'appelle `Monfichier.txt`. L'instruction d'ouverture de fichier devra donc nécessairement faire référence à ce nom là.
- Le nom de l'**objet-fichier** est un nom de variable quelconque. Dans notre première instruction, nous avons choisi de créer un objet-fichier `ofi`, faisant référence au fichier réel `Monfichier.txt`, lequel est ouvert en lecture (argument `'r'`).
- Les deux chaînes de caractères que nous avions entrées dans le fichier sont à présent accolées en une seule. C'est normal, puisque nous n'avons fourni aucun caractère de séparation lorsque nous les avons enregistrées. Nous verrons un peu plus loin comment enregistrer des lignes de texte distinctes.
- La méthode `read()` peut également être utilisée avec un argument. Celui-ci indiquera combien de caractères doivent être lus, à partir de la position déjà atteinte dans le fichier :

```
>>> ofi = open('Monfichier.txt', 'r', encoding = 'utf-8')
>>> t = ofi.read(7)
>>> print(t)
Bonjour
>>> t = ofi.read(15)
>>> print(t)
, fichier !Quel
```

S'il ne reste pas assez de caractères au fichier pour satisfaire la demande, la lecture s'arrête tout simplement à la fin du fichier :

```
>>> t = ofi.read(1000)
>>> print(t)
 beau temps, aujourd'hui !
```

- N'oubliez pas de refermer le fichier après usage :

```
>>> ofi.close()
```

L'ouverture en lecture d'un fichier n'existant pas engendre automatiquement la levée d'une exception :

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

Afin d'intercepter l'erreur et éviter ainsi l'interruption d'un programme, il est possible de recourir au mécanisme de traitement d'exceptions de Python utilisant l'ensemble des instructions `try - except - else`.

Considérons par exemple un script qui demande à l'utilisateur d'entrer un nom de fichier, destiné à être ouvert en lecture. Si le fichier n'existe pas, nous ne voulons pas que le programme se plante. Nous voulons qu'un avertissement soit affiché, et que l'utilisateur puisse essayer d'entrer un autre nom :

```
filename = input("Veuillez entrer un nom de fichier : ")
try:
    f = open(filename, "r", encoding = 'utf-8')
except:
    print("Le fichier", filename, "est introuvable")
```

Ce genre de test peut également être inclus dans une fonction :

```
def existe(fname):
    try:
        f = open(fname,'r', encoding = 'utf-8')
        f.close()
        return 1
    except:
        return 0

filename = input("Veuillez entrer le nom du fichier : ")
if existe(filename):
    print("Ce fichier existe bel et bien.")
else:
    print("Le fichier", filename, "est introuvable.")
```

Vous aurez remarqué que l'instruction `if` utilisée dans ce programme est construite de manière un peu spéciale : la condition à évaluer a été remplacée par une simple constante générée par la fonction existe. Rappelons que l'interpréteur considère comme vrai toute valeur numérique différente de zéro.

3 Fichiers texte

Un **fichier texte** est un fichier contenant des caractères imprimables et des espaces organisés en lignes, ces lignes étant séparées les unes des autres par un caractère non-imprimable appelé **marqueur de fin de ligne**[4].

Les fichiers texte sont donc des fichiers que nous pouvons lire et comprendre à l'aide d'un simple éditeur de texte, par opposition aux **fichiers binaires** dont le contenu est – au moins en partie – inintelligible pour un lecteur humain, et qui ne prend son sens que lorsqu'il est décodé par un logiciel spécifique. Par exemple, les fichiers contenant des images, des sons, des vidéos, etc. sont presque toujours des fichiers binaires.

Il est très facile de traiter des fichiers texte avec Python. Par exemple, les instructions suivantes suffisent pour créer un fichier texte de quatre lignes :

```
>>> f = open("Fichiertexte.txt", "w", encoding="utf-8")
>>> f.write("Ceci est la ligne un\nVoici la ligne deux\n")
41
>>> f.write("Voici la ligne trois\nVoici la ligne quatre\n")
43
>>> f.close()
```

Notez bien le marqueur de fin de ligne `\n` inséré dans les chaînes de caractères, aux endroits où l'on souhaite séparer les lignes de texte dans l'enregistrement. Sans ce marqueur, les caractères seraient enregistrés les uns à la suite des autres, comme dans les exemples précédents.

4. Suivant le système d'exploitation, le codage correspondant au marqueur de fin de ligne peut être différent. En principe, vous n'avez pas à vous préoccuper de ces différences. Lors des opérations d'écriture, Python utilise la convention en vigueur sur votre système d'exploitation.

Lors des opérations de lecture, les lignes d'un fichier texte peuvent être extraites séparément les unes des autres : La méthode `readline()` ne lit qu'une seule ligne à la fois (en incluant le caractère de fin de ligne) :

```
>>> f = open('Fichiertexte.txt','r', encoding = "utf-8")
>>> t = f.readline()
>>> print(t)
Ceci est la ligne un

>>> print(f.readline())
Voici la ligne deux
.
```

La méthode `readlines()` transfère toutes les lignes restantes dans une liste de chaînes :

```
>>> t = f.readlines()
>>> print(t)
['Voici la ligne trois\n', 'Voici la ligne quatre\n']
>>> f.close()
```

Remarques

(a) La liste apparaît ci-dessus en format brut, avec des apostrophes pour délimiter les chaînes, et les caractères spéciaux sous leur forme conventionnelle. Vous pourrez bien évidemment parcourir cette liste pour en extraire les chaînes individuelles.

(b) La méthode `readlines()` permet donc de lire l'intégralité d'un fichier en une instruction seulement. Cela n'est possible toutefois que si le fichier à lire n'est pas trop gros : puisqu'il est copié intégralement dans une variable, c'est-à-dire dans la mémoire vive de l'ordinateur, il faut que la taille de celle-ci soit suffisante. Si vous devez traiter de gros fichiers, utilisez plutôt la méthode `readline()` dans une boucle, comme le montrera l'exemple suivant.

(c) Notez bien que `readline()` est une méthode qui renvoie une **chaîne de caractères**, alors que la méthode `readlines()` renvoie une **liste**. À la fin du fichier, `readline()` renvoie une chaîne vide, tandis que `readlines()` renvoie une liste vide.

Il va de soi que les boucles de programmation s'imposent lorsque l'on doit traiter un fichier texte dont on ne connaît pas nécessairement le contenu à l'avance. L'idée de base consistera à lire le fichier ligne après ligne jusqu'à ce que l'on ait atteint la fin du fichier. Le script qui suit vous montre comment créer une fonction destinée à effectuer un certain traitement sur un fichier texte. En l'occurrence, il s'agit ici de recopier un fichier texte, en omettant toutes les lignes qui commencent par un caractère `'#'` :

```
def filtre(source,destination):
    """ filtre(str source, str destination) --> None.
Recopie le fichier <source> dans le fichier <destination> en éliminant les commentaires
    """
    fs = open(source, 'r', encoding = 'utf-8')
    fd = open(destination, 'w', encoding = 'utf-8')
    while 1:
        txt = fs.readline()
        if txt == '':
            break
        if txt[0] != '#':
            fd.write(txt)
    fs.close()
    fd.close()
```

Pour appeler cette fonction, vous devez utiliser deux arguments : le nom du fichier original, et le nom du fichier destiné à recevoir la copie filtrée :

```
filtre('test.txt', 'test_f.txt')
```

Une boucle `while` construite comme nous l'avons fait ci-dessus devrait boucler indéfiniment, puisque la condition de continuation reste toujours vraie. Rappelons néanmoins que nous pouvons interrompre ce bouclage en faisant appel à l'instruction `break`, laquelle permet de mettre en place un mécanisme de sortie de boucle. Dans notre fonction `filtre`, il est facile de voir que l'instruction `break` s'exécutera seulement lorsque la fin du fichier aura été atteinte.

4 Fichiers binaires

Rappelons que sous Python 3, les chaînes de caractères sont des chaînes *Unicode*, ce qui signifie que les identifiants numériques de leurs caractères sont uniques (il ne peut exister qu'un seul caractère typographique pour chaque code) et universels (les identifiants choisis couvrent la gamme complète de tous les caractères utilisés dans les différentes langues du monde entier). Pour rappel, la norme *Unicode* ne fixe aucune règle concernant le nombre d'octets ou de bits à réserver pour l'encodage. Elle spécifie seulement la valeur numérique de l'identifiant associé à chaque caractère. En fonction des besoins, chaque système est donc libre d'encoder « en interne » cet identifiant comme bon lui semble.

Bien que nous n'ayons pas à nous préoccuper du format réel des caractères d'une chaîne en Python, il en va tout autrement pour les entrées/sorties. Les développeurs que nous sommes devons absolument pouvoir préciser sous quelle forme exacte les données sont attendues par nos programmes, que ces données soient fournies par l'intermédiaire de frappes au clavier ou par importation depuis une source quelconque. De même, nous devons pouvoir choisir le format des données que nous exportons vers n'importe quel dispositif périphérique, qu'il s'agisse d'une imprimante, d'un disque dur, d'un écran...

Pour toutes ces entrées ou sorties de chaînes de caractères, nous devrons donc toujours considérer qu'il s'agit **concrètement** de séquences d'octets, et utiliser divers mécanismes pour convertir ces séquences d'octets en chaînes de caractères, ou vice-versa.

Python met à votre disposition le nouveau type de données **bytes**, spécifiquement conçu pour traiter les séquences (ou chaînes) d'octets. Les données de type **bytes** sont très similaires aux données de type **string**, mais avec la différence que ce sont strictement des séquences d'octets, et non des séquences de caractères. Les caractères peuvent bien entendu être **encodés** en octets, et les octets **décodés** en caractères, mais pas de manière univoque : du fait qu'il existe plusieurs normes d'encodage/décodage, la même chaîne **string** peut être convertie en plusieurs chaînes **bytes** différentes.

À titre d'exemple[5], nous allons effectuer à la ligne de commande un petit exercice d'écriture/lecture d'un fichier texte, en exploitant quelques possibilités de la fonction `open()` que nous n'avions pas encore rencontrées jusqu'ici. Nous veillerons à faire cet exercice avec une chaîne contenant quelques caractères accentués, ou d'autres symboles **non-ASCII standard** :

```
>>> chaine = "Amélie et Eugène\n"
>>> of = open("test.txt", "w")
>>> of.write(chaine)
17
>>> of.close()
```

Avec ces quelques lignes, nous avons enregistré la chaîne de caractères `chaine` sous la forme d'une ligne de texte dans un fichier. Effectuons à présent une relecture de ce fichier, mais en veillant à ouvrir celui-ci **en mode « binaire »**, ce qui peut se faire aisément en transmettant l'argument `'rb'` à la fonction `open()`. Dans ce mode, les octets sont transférés à l'état brut, sans conversion d'aucune sorte. La lecture avec `read()` ne nous fournit donc plus une chaîne de caractères mais bien une chaîne d'octets, et la variable qui les accueille est pour cette raison automatiquement typée comme variable du type **bytes** :

```
>>> of = open("test.txt", "rb")          # "rb" => mode lecture (r) binaire (b)
>>> octets = of.read()
>>> of.close()
>>> type(octets)
<class 'bytes'>
```

En procédant ainsi, nous ne récupérons donc pas notre chaîne de caractères initiale, mais bien sa traduction concrète en octets, dans une donnée de type **bytes**. Essayons d'afficher cette donnée à l'aide de la fonction **print()** :

```
>>> print(octets)
b'Am\xc3\xa9lie et Eug\xc3\xa8ne\n'
```

5. Pour cet exemple, nous supposons que la norme d'encodage par défaut sur votre système d'exploitation est Utf-8. Si vous utilisez un système d'exploitation ancien, utilisant par exemple la norme Latin-1 (ou Windows-1252), les résultats seront légèrement différents en ce qui concerne les nombres et valeurs des octets, mais vous ne devriez pas avoir de mal à interpréter ce que vous obtenez.

Lorsqu'on lui demande d'afficher une donnée de type **bytes**, Python nous en fournit en fait une **représentation**, entre deux apostrophes pour indiquer qu'il s'agit d'une chaîne, mais celles-ci précédées d'un **b** minuscule pour spécifier qu'il s'agit d'une chaîne d'octets (**bytes)**, avec les conventions suivantes :

- Les octets de valeur numérique comprise entre 32 et 127 sont représentés par le caractère correspondant du code ASCII.
- Certains octets de valeur numérique inférieure à 32 sont représentés de manière conventionnelle, comme par exemple le caractère de fin de ligne (« `\n` »).
- Les autres octets sont représentés par leur valeur hexadécimale, précédée de « `\x` ».

Dans le cas de notre exemple, on voit que les caractères non accentués de la chaîne utilisée ont été encodés chacun à l'aide d'un seul octet correspondant à son code ASCII : nous les reconnaissons donc directement. Les caractères accentués, qui n'existent pas dans le code ASCII standard, sont quant à eux encodés chacun sur deux octets : `\xc3` et `\xa9` pour le « é », `\xc3` et `\xa8` pour le « è ». Cette forme particulière d'encodage correspond à la norme **Utf-8**, que nous décrirons un peu plus en détail dans les pages suivantes.

La représentation obtenue avec `print()` nous aide à reconnaître notre chaîne initiale, certes, mais elle ne nous montre pas assez bien qu'il s'agit en fait d'octets. Essayons donc autre chose. Vous savez que l'on peut aussi examiner le contenu d'une séquence, élément par élément, à l'aide d'une boucle de **parcours**. Voyons ce que cela donne ici :

```
>>> for oct in octets:
...     print(oct, end =' ')
...
65 109 195 169 108 105 101 32 101 116 32 69 117 103 195 168 110 101 10
```

Cette fois, nous voyons très clairement qu'il s'agit bien d'octets : le parcours nous en restitue toutes les valeurs numériques, en notation décimale.

Du fait que les caractères accentués sont encodés sur deux octets en **Utf-8**, la fonction `len()` ne nous renvoie pas la même valeur pour la chaîne de caractères, et pour son équivalent encodé en **Utf-8** dans une chaîne d'octets :

```
>>> len(chaine)
17
>>> len(octets)
19
```

Finalement, notons que, pour enregistrer une séquence d'octets, il faut toujours ouvrir le fichier en *mode binaire,* en utilisant l'argument `"wb"`au lieu de `"w"`dans la fonction `open`. Il n'est en effet pas possible d'enregistrer une chaîne d'octets telle quelle dans un fichier texte :

```
>>> of =open("test.txt", "w")
>>> of.write(octets)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: must be str, not bytes
```

4.1 L'encodage Utf-8

Tout ce qui précède nous indique que la chaîne de caractères initiale de notre exemple a dû être automatiquement convertie, lors de son enregistrement dans un fichier, en une chaîne d'octets encodés suivant la norme **Utf-8**. La séquence d'octets dont nous avons traité jusqu'ici correspond donc à une forme particulière d'encodage numérique, pour la chaîne de caractères : « Amélie et Eugène »

Même si cela peut vous paraître à première vue un peu compliqué, dites-vous bien que malheureusement l'encodage idéal n'existe pas. En fonction de ce que l'on veut en faire, il peut être préférable d'encoder un même texte de plusieurs manières différentes. C'est pour cette raison qu'ont été définies, en parallèle avec la norme **Unicode**, plusieurs **normes d'encodage** : **Utf-8, Utf-16, Utf-32,** et quelques variantes. Toutes ces normes utilisent les mêmes identifiants numériques pour encoder les caractères, mais elles diffèrent sur la manière d'enregistrer concrètement ces identifiants sous forme d'octets. Ne vous affolez pas, cependant : vous ne serez vraisemblablement jamais confronté qu'à la première d'entre elles (**Utf-8**). Les autres ne concerneront que certains spécialistes de domaines « pointus ».

La norme d'encodage **Utf-8** est désormais la norme préférentielle pour la plupart des textes courants car :

- d'une part, elle assure une parfaite compatibilité avec les textes encodés en « pur » **ASCII** (ce qui est le cas de nombreux codes sources de logiciels), ainsi qu'une compatibilité partielle avec les textes encodés à l'aide de ses variantes « étendues », telles que **Latin-1** ;
- d'autre part, cette nouvelle norme est celle qui est la plus économe en ressources, tout au moins pour les textes écrits dans une langue occidentale.

Suivant cette norme, les caractères du jeu **ASCII** standard sont encodés sur un seul octet. Les autres seront encodés en général sur deux octets, parfois trois ou même quatre octets pour les caractères les plus rares.

À titre de comparaison, rappelons que la norme la plus couramment utilisée avant **Utf-8** par les francophones, était la norme **Latin-1.** Cette norme permettait d'encoder sur un seul octet un jeu de caractères accentués restreint, correspondant aux principales langues de l'Europe occidentale (Français, Allemand, Portugais, etc.).

Les normes Utf-16 et Utf-32 encodent systématiquement tous les caractères sur deux octets pour la première, et quatre octets pour la seconde. Ces normes ne sont utilisées que pour des usages très spécifiques, comme par exemple pour le traitement interne des chaînes de caractères par un compilateur. Vous ne les rencontrerez guère.

4.2 Conversion (encodage/décodage) des chaînes

Il vous arrivera peut-être de devoir convertir des fichiers encodés suivant une norme ancienne et/ou étrangère : un programmeur digne de ce nom doit être capable d'effectuer ces conversions. Python vous fournit fort heureusement les outils nécessaires, sous la forme de **méthodes** des objets concernés.

Conversion d'une chaîne bytes en chaîne string

Considérons par exemple la séquence d'octets obtenue à la fin de notre précédent petit exercice. Si nous savons que cette séquence correspond à un texte encodé suivant la norme **Utf-8**, nous pouvons la **décoder** en chaîne de caractères à l'aide de la méthode `decode()`, avec l'argument `'Utf-8'` (ou indifféremment : `'utf-8'`, `'Utf8'` ou `'utf8'`) :

```
>>> ch_car = octets.decode("utf8")
>>> ch_car
'Amélie et Eugène\n'
>>> type(ch_car)
<class 'str'>
```

Le **parcours** de la chaîne obtenue nous fournit bien des caractères, cette fois :

```
>>> for c in ch_car:
...     print(c, end =' ')
...
A m é l i e   e t   E u g è n e
```

Conversion d'une chaîne string en chaîne bytes

Pour convertir une chaîne de caractères en une séquence d'octets, encodée suivant une norme particulière, on utilise la méthode `encode()`, qui fonctionne de manière parfaitement symétrique à la méthode `decode()`. Convertissons la même chaîne de caractères, à la fois en **Utf-8** et en **Latin-1** pour comparaison :

```
>>> chaine = "Amélie et Eugène"
>>> octets_u = chaine.encode("Utf-8")
>>> octets_l = chaine.encode("Latin-1")
>>> octets_u
b'Am\xc3\xa9lie et Eug\xc3\xa8ne'
>>> octets_l
b'Am\xe9lie et Eug\xe8ne'
```

Dans les séquences d'octets ainsi obtenues, on voit clairement que les caractères accentués « é » et « è » sont encodés à l'aide de deux octets dans le cas de la séquence **Utf-8**, et à l'aide d'un seul octet dans le cas de la séquence **Latin-1**.

Conversions automatiques lors du traitement des fichiers

Jusqu'à présent, nous n'avons pas attiré votre attention sur le problème constitué par la norme d'encodage de chaînes à mémoriser dans un fichier texte, parce que la fonction `open()` de Python dispose fort heureusement d'un paramétrage par défaut qui convient à la plupart des situations modernes concrètes. Lorsque vous ouvrez un fichier en écriture, par exemple, en choisissant `'w'` ou `'a'` comme deuxième argument pour `open()`, Python encode automatiquement les chaînes à enregistrer en suivant la norme par défaut de votre système d'exploitation (dans nos exemples, nous avons considéré qu'il s'agissait de **Utf-8**), et la conversion inverse est effectuée lors des opérations de lecture. Nous avons donc pu aborder l'étude des fichiers sans vous encombrer l'esprit avec des explications trop détaillées.

Voyons à présent comment enregistrer des textes en leur appliquant un encodage différent de celui qui est prévu par défaut à l'aide de l'argument `encoding` de la fonction `open`. À titre d'exemple, nous pouvons refaire les exercices des pages précédentes en forçant cette fois l'encodage en **Latin-1** :

```
>>> chaine ="Amélie et Eugène\n"
>>> of =open("test.txt", "w", encoding ="Latin-1")
>>> of.write(chaine)
17
>>> of.close()
>>> of =open("test.txt", "rb")
>>> octets =of.read()
>>> of.close()
>>> print(octets)
b'Am\xe9lie et Eug\xe8ne\n'
```

... etc.

(À vous d'effectuer divers contrôles et essais sur cette séquence d'octets, si vous le souhaitez).

C'est pareil lorsque vous ouvrez un fichier **en lecture**. Par défaut, Python considère que le fichier est encodé suivant la norme par défaut de votre système d'exploitation, mais ce n'est évidemment pas une certitude. Essayons par exemple de ré-ouvrir sans précaution le fichier **test.txt** que nous venons de créer dans les lignes précédentes :

```
>>> of =open("test.txt", "r")
>>> ch_lue =of.read()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/usr/lib/python3.1/codecs.py", line 300, in decode
    (result, consumed) = self._buffer_decode(data, self.errors, final)
UnicodeDecodeError: 'utf8' codec can't decode bytes in position 2-4:
invalid data
```

Le message d'erreur est explicite : supposant que le fichier était encodé en **Utf-8**, Python n'a pas pu le décoder [6]. Tout rentre dans l'ordre si nous précisons :

```
>>> of =open("test.txt", "r", encoding ="Latin-1")
>>> ch_lue =of.read()
>>> of.close()
>>> ch_lue
'Amélie et Eugène\n'
```

Dans les scripts élaborés, il sera probablement toujours préférable de préciser l'encodage supposé pour les fichiers que vous traitez, quitte à devoir demander cette information à l'utilisateur, ou à imaginer des tests plus ou moins élaborés pour le déterminer de façon automatique (ce qui est loin d'être évident !).

6. En informatique, on appelle **codec** (*codeur/décodeur*) tout dispositif de conversion de format. Vous rencontrerez par exemple de nombreux codecs dans le monde du multimedia (codecs audio, vidéo...). Python dispose de nombreux codecs pour convertir les chaînes de caractères suivant les différentes normes en vigueur.

4.3 Enregistrement et restitution de variables diverses

L'argument de la méthode `write()` utilisée avec un fichier texte 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 (**string**). Nous pouvons réaliser cela à l'aide de la fonction intégrée `str()` :

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

Cependant, si nous enregistrons les valeurs numériques en les transformant d'abord en chaînes de caractères, nous risquons de ne plus pouvoir les re-transformer correctement en valeurs numériques lorsque nous allons relire le fichier.

Exemple :

```
>>> a, b, c = 5, 2.83, 67
>>> f = open('Monfichier.txt', 'w', encoding = 'utf-8')
>>> f.write(str(a))
>>> f.write(str(b))
>>> f.write(str(c))
>>> f.close()
>>> f = open('Monfichier.txt', 'r', encoding = 'utf-8')
>>> print(f.read())
52.8367
>>> f.close()
```

Nous avons enregistré trois valeurs numériques. Mais comment pouvons-nous les distinguer dans la chaîne de caractères résultante, lorsque nous effectuons la lecture du fichier ? C'est impossible ! Rien ne nous indique d'ailleurs qu'il y a là trois valeurs plutôt qu'une seule, ou 2, ou 4...

Il existe plusieurs solutions à ce genre de problèmes. L'une des meilleures consiste à importer un module Python spécialisé : le module `pickle` (= *conserver* en anglais). Voici comment il s'utilise :

```
>>> import pickle
>>> a, b, c = 27, 12.96, [5, 4.83, "René"]
>>> f =open('donnees_test', 'wb')
>>> pickle.dump(a, f)
>>> pickle.dump(b, f)
>>> pickle.dump(c, f)
>>> f.close()
>>> f = open('donnes_test', 'rb')
>>> j = pickle.load(f)
>>> k = pickle.load(f)
>>> l = pickle.load(f)
>>> print(j, type(j))
27 <class 'int'>
>>> print(k, type(k))
12.96 <class 'float'>
>>> print(l, type(l))
[5, 4.83, 'René'] <class 'list'>
>>> f.close()
```

Comme vous pouvez le constater dans ce court exemple, le module `pickle` permet d'enregistrer des données avec conservation de leur type. Les contenus des trois variables `a`, `b` et `c` sont enregistrés dans le fichier `donnees_test`, et ensuite fidèlement restitués, avec leur type, dans les variables `j`, `k` et `l`. Vous pouvez donc mémorisera ainsi des entiers, des réels, des chaînes de caractères, des listes ou des dictionnaires.

Les fichiers traités à l'aide des fonctions du module `pickle` ne seront pas des des fichiers texte, mais bien des `fichiers binaires`. Pour cette raison, ils doivent obligatoirement être ouverts comme tels à l'aide de la fonction `open()`. Vous utiliserez l'argument `'wb'` pour ouvrir un fichier binaire en écriture (comme à la 3ème ligne de notre exemple), et l'argument `'rb'` pour ouvrir un fichier binaire en lecture (comme à la 8ème ligne de notre exemple) mais sans préciser l'encodage avec l'argument `encoding`.

La fonction `dump()` du module `pickle` attend deux arguments : le premier est la variable à enregistrer, le second est l'objet fichier dans lequel on travaille. La fonction `pickle.load()` effectue le travail inverse, c'est-à-dire la restitution de chaque variable avec son type.

5 Glossaire du chapitre

La liste suivante répertorie dans l'ordre alphabétique les principaux termes introduits dans ce chapitre. La connaissance de leur définition est indispensable à la suite de votre apprentissage :

base de données : fichier dont le contenu est organisé comme un dictionnaire avec des clés mises en correspondance avec des valeurs.

bytes : type de données Python composées de séquences d'octets.

chemin absolu : chemin identifiant l'emplacement d'un fichier à partir du répertoire-racine.

chemin relatif : chemin identifiant l'emplacement d'un fichier à partir du répertoire courant.

chemin : adresse identifiant l'emplacement d'un fichier dans l'arborescence du système de fichiers.

fichier binaire : fichier dans lequel les bits ne représentent pas directement une simple suite de caractères imprimables, d'espaces et de retours de ligne. Dès lors, un tel fichier ne peut être directement lu par un éditeur de texte. Un fichier binaire contient souvent du langage machine, du son, de la vidéo, des images,...

fichier texte : fichier qui contient des caractères imprimables et des espaces organisés en lignes successives, ces lignes étant séparées les unes des autres par un caractère spécial non-imprimable appelé *marqueur de fin de ligne.*

objet-fichier : objet référençant un fichier et possédant des méthodes et des attributs qui permettent d'obtenir des informations et de manipuler le fichier référencé.

répertoire-courant : répertoire de travail dans lequel un programme se trouve présentement.

répertoire-racine : base des répertoires dans la hiérarchie d'un système de fichiers qui peut contenir d'autres répertoires mais qui ne peut être contenu lui-même dans un répertoire parent.

répertoire : fichier spécial appelé aussi dossier et contenant la liste d'autres fichiers.

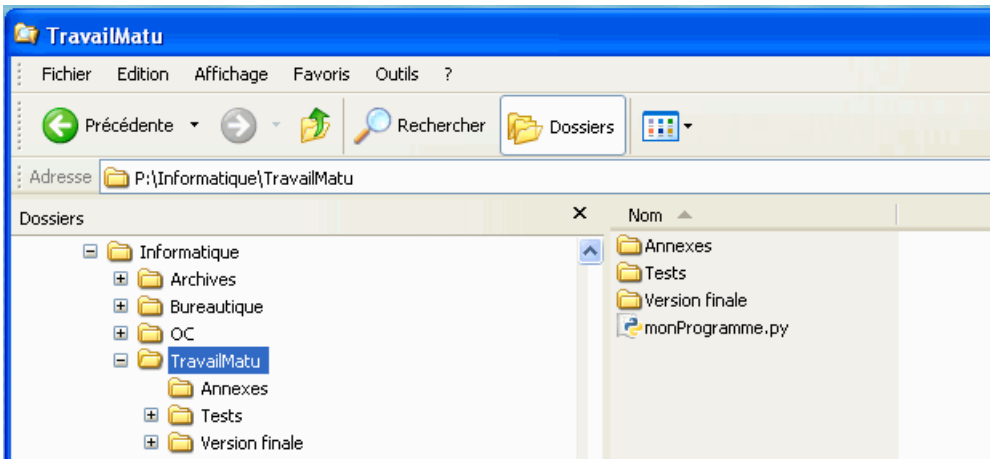
Unicode : norme d'encodage universelle visant à donner à tout caractère de n'importe quel système d'écriture de langage un nom et un identifiant numérique, et ce de manière unifiée, quelle que soit la plate-forme informatique ou le logiciel.

6 Exercices

Exercice 6.1
Écrivez un script qui permet de créer et de relire aisément un fichier texte. Votre programme demandera d'abord à l'utilisateur d'entrer le nom du fichier. Ensuite il lui proposera le choix, soit d'enregistrer de nouvelles lignes de texte, soit d'afficher le contenu du fichier. En cas de lecture d'un fichier inexistant, votre programme devra avertir l'utilisateur et lui demander un nom de fichier valide.
L'utilisateur devra pouvoir entrer ses lignes de texte successives en utilisant simplement la touche <Enter> pour les séparer les unes des autres. Pour terminer les entrées, il lui suffira d'entrer une ligne vide (c'est-à-dire utiliser la touche <Enter> seule).
L'affichage du contenu devra montrer les lignes du fichier séparées les unes des autres de la manière la plus naturelle (les codes de fin de ligne ne doivent pas apparaître).

Exercice 6.2
Écrivez une fonction `compare(str nomFichier1, str nomFichier2) --> int` qui compare le contenu des fichiers `nomFichier1` et `nomFichier2` et retourne l'indice de la première différence rencontrée ou -1 si les fichiers sont identiques.

Exercice 6.3
Le fichier texte *TM2021.txt* situé dans le sous-répertoire *Annexes* du dossier *TravailMatu* de votre lecteur P doit être traité par le script Python *monProgramme.py*, lui-même situé dans le dossier *TravailMatu* comme l'illustre l'arborescence suivante :



Suite à une restructuration de votre système de fichiers durant l'été, vous décidez de déplacer le contenu entier du dossier *TravailMatu* dans le répertoire *Archives* de votre lecteur P. Parmi les versions *A* et *B* du script *monProgramme.py* ci-dessous, laquelle ne sera-t-elle pas affectée par ce changement organisationnel ? Justifiez votre réponse en mettant notamment en évidence la différence entre ces deux versions.

Version A

```
...
# ouverture du fichier
of = open('P:/Informatique/TravailMatu/Annexes/TM2021.txt', 'r')
...
```

Version B

```
...
# ouverture du fichier
of = open('./Annexes/TM2021.txt', 'r')
...
```

Exercice 6.4
Vous disposez d'un fichier contenant des valeurs numériques représentant les diamètres d'une série de sphères. Écrivez un script qui utilise les données de ce fichier pour en créer un autre, organisé en lignes de texte qui exprimeront « en clair » les autres caractéristiques de ces sphères (surface de section, surface extérieure et volume), dans des lignes formatées comme suit :

```
Diam.  46.20 cm Section =  1676.39 cm2 Surf. =  6705.54 cm2 Vol. =  51632.67 cm3
Diam. 120.00 cm Section = 11309.73 cm2 Surf. = 45238.93 cm2 Vol. = 904778.68 cm3
Diam.   0.03 cm Section =     0.00 cm2 Surf. =     0.00 cm2 Vol. =      0.00 cm3
Diam.  13.90 cm Section =   151.75 cm2 Surf. =   606.99 cm2 Vol. =   1406.19 cm3
Diam.  88.80 cm Section =  6193.21 cm2 Surf. = 24772.84 cm2 Vol. = 366638.04 cm3
Diam.   7.45 cm Section =    43.59 cm2 Surf. =   174.37 cm2 Vol. =    216.50 cm3
Diam.   8.89 cm Section =    62.09 cm2 Surf. =   248.34 cm2 Vol. =    368.00 cm3
Diam.  34.58 cm Section =   939.00 cm2 Surf. =  3755.99 cm2 Vol. =  21645.11 cm3
etc.
```

Exercice 6.5
Écrivez une fonction `filtre(str source, str destination, str car) --> None` enregistrant les mots du fichier `source` ne contenant pas le caractère `car` dans le fichier `destination`. Le corps principal du programme permettra, à partir d'un fichier contenant tous les mots de la langue française, de récupérer dans un nouveau fichier tous les mots français ne contenant pas de "r". Le programme retournera de manière aléatoire 20 mots issus de ce nouveau fichier :

```
Entrez le nom du fichier source: mots-francais.txt
Entrez le nom du fichier d'enregistrement: mots-filtres.txt
Voici une liste de 20 mots exempts de "r":

pourvoirons
désappointant
bariolions
amincit
...
```

Exercice 6.6
Écrivez une fonction `plusLong(str source) --> str` retournant le plus long mot situé dans le fichier `source`, supposé sans ponctuation. Appliquée aux recueils *Alcools* de Guillaume Apollinaire, cette fonction retournera :

```
>>> plusLong('Alcools.txt')
'sténo-dactylographes'
```

Exercice 6.7
Écrivez une fonction `compteur(str source, str destination, str motCherche) --> None` comptant le nombre d'occurrences du mot `motCherche` dans le fichier `source` et enregistrant le résultat obtenu dans le fichier `destination` sous la forme

Le fichier "`source`" contient `n` occurrences du mot "`motCherche`",

où `source`, `n` et `motCherche` sont à remplacer par les valeurs correspondantes. Votre résolution devra en outre répondre aux critères suivants :

- Le fichier `source` est supposé encodé dans la norme Utf-8 et ne contient pas de marque de ponctuation.
- Le fichier `destination` est supposé encodé dans la norme Latin-1.
- La méthode `count` ne doit pas être utilisée.
- La chaîne de caractères à enregistrer dans le fichier `destination` doit être formatée à l'aide de la méthode `format`.

En appliquant la fonction `compteur` à la recherche du nombre d'occurrences du mot *soleil* dans le recueil *Alcools* de Guillaume Apollinaire, l'utilisateur devra récupérer 28 dans le fichier destination.

Exercice 6.8
Ecrivez un script permettant de calculer les fréquences d'apparition de chacune des lettres de l'alphabet situées dans un fichier texte. Après avoir comptabilisé les fréquences d'apparition des lettres accentués avec leur correspondant non accentué, votre script enregistrera les résultats obtenus selon l'ordre alphabétique dans un fichier. Chaque ligne de ce fichier indiquera la fréquence d'apparition d'une lettre selon le format

`lettre@pourcent%`,

où `lettre` correspond à la lettre étudiée et `pourcent` à sa fréquence d'apparition exprimée en pour-cent avec 3 chiffres après la virgule. Appliqué aux *3 Mousquetaires* d'Alexandre DUMAS, votre script retournera une fréquence d'environ 16.436% pour la lettre "e".

Exercice 6.9
Ecrivez un script permettant de créer une base de données formée d'une unique table dans laquelle seront mémorisés les noms d'une série de personnes, leur âge et leur taille. La table de données sera représentée par un dictionnaire - global au programme - dont les clés seront les noms des personnes et les valeurs seront constituées de tuples `(age, taille)`, dans lesquels `age` sera exprimé en années et `taille` en mètres. Votre script comportera quatre fonctions permettant d'interagir avec la table de données :

- `ajouter(dict table) --> None` permettra d'ajouter au dictionnaire `table` les informations concernant de nouvelles personnes à enregistrer. Cette opération se fera au moyen d'une boucle acceptant les données entrées par l'utilisateur ;
- `consulter(dict table) --> None` permettra à l'utilisateur de fournir en boucle un nom quelconque pour obtenir en retour le tuple `age, taille` correspondant dans le dictionnaire `table`. Le résultat d'une telle requête devra être une ligne de texte formatée sous la forme suivante :

  ```
  Nom: Jean-Claude Dusse - âge : 47 ans - taille : 1.55
  ```

  Une requête non fructueuse retournera quant à elle le message d'erreur suivant :

  ```
  *** Nom inconnu ! ***
  ```

- `sauvegarder(dict table) --> None` permettra d'enregistrer définitivement les éléments du dictionnaire `table` dans un fichier texte. Chaque élément de `table` sera enregistré dans le fichier sur une ligne distincte au format suivant :

  `Nom@Age#Taille%`

  où `Nom`, `Age` et `Taille` correspondent aux données associées à l'élément enregistré.
- `remplir(dict table) -> None` permettra de remplir le dictionnaire `table` à partir d'un fichier texte externe dont chaque ligne contiendra les données associées à une personne enregistrée selon le format suivant :

  `Nom@Age#Taille%`

  Cette fonction pourra être appelée par l'utilisateur au démarrage du programme afin de récupérer dans la table de données globale au programme toutes les données enregistrées lors de précédentes sessions.

Au niveau du menu principal de votre programme, le flux d'exécution sera dirigé à l'aide d'un dictionnaire. Votre programme affichera par exemple :

```
Choisissez :
(R)écupérer une base de données préexistante sauvegardée dans un fichier
(A)jouter des données à la base de données courante
(C)onsulter la base de données courante
(S)auvegarder la base de données courante dans un fichier
(T)erminer
```

Suivant le choix opéré par l'utilisateur, vous effectuerez alors l'appel de la fonction correspondante en la sélectionnant dans un dictionnaire de fonctions.

Exercice 6.10
Considérons que vous avez fait l'exercice précédent et que vous disposez à présent d'un fichier contenant les coordonnées d'un certain nombre de personnes. Écrivez un script qui permette d'extraire de ce fichier les coordonnées de toutes les personnes répertoriées qui ont un âge compris entre deux bornes **a** et **b** fourni par l'utilisateur. Les coordonnées de toutes les personnes vérifiant ce critère seront retournées au format :

```
Nom: Jean-Claude Dusse - âge : 47 ans - taille : 1.55
```

Exercice 6.11
Écrivez un script qui permette de générer et consulter un fichier binaire contenant les tables de multiplication de 2 à 30 sous forme de listes (chacune d'entre elles incluant 20 termes seulement). Votre programme demandera d'abord à l'utilisateur d'entrer le nom du fichier. Ensuite, il lui proposera le choix, soit d'enregistrer les tables de multiplications de 2 à 30, soit d'afficher l'une des tables de multiplication entre 2 et 30.

Exercice 6.12
Écrivez un script qui recopie un fichier texte en triplant tous les espaces entre les mots. Le fichier source sera fourni encodé à la norme Latin-1 et le fichier destinataire devra être encodé en UTF-8, la conversion se faisant de manière automatique. Les noms des deux fichiers devront être demandés en début de script.

Exercice 6.13
Variante de l'exercice précédent : effectuez les opérations de lecture et d'écriture des fichiers en mode binaire et les opérations de décodage/encodage sur les séquences d'octets de manière manuelle.