# 9

# Manipuler des fichiers

*Jusqu'à présent, les programmes que nous avons réalisés ne traitaient qu'un très petit nombre de données. Nous pouvions donc à chaque fois inclure ces données dans le corps du programme lui-même (par exemple dans une liste). Cette façon de procéder devient cependant tout à fait inadéquate lorsque l'on souhaite traiter une quantité d'informations plus importante.*

## Utilité des fichiers

Imaginons par exemple que nous voulions écrire un petit programme exerciseur 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 (ce serait d'ailleurs bien pénible à écrire : n'oubliez pas que nous souhaitons traiter un grand nombre de questions !) :

```
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. Exemple :

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

> *- Rappel -*
> *L'indiçage commence toujours à partir de zéro.*

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 *Oracle*, *IBM DB2*, *Sybase*, *Adabas*, *PostgreSQL*, *MySQL*, etc.

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.

## 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, et vous pouvez aussi 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 normal 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 votre disque dur, sur 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[51].

## *Noms de fichiers – le 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*. Celui-ci est habituellement le 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* lui-même (sous *Windows*, la définition de ce répertoire fait partie des propriétés de l'icône de lancement).

Si vous travaillez avec *IDLE*, vous souhaiterez donc certainement forcer Python à changer son répertoire courant, afin que celui-ci corresponde à vos attentes. Pour ce faire, utilisez les commandes suivantes en début de session. Nous supposons pour la démonstration que le répertoire visé est le répertoire **`/home/jules/exercices`** . Même si vous travaillez sous *Windows* (où ce n'est pas la règle), vous pouvez utiliser cette syntaxe (c'est-à-dire des caractères **/** et non \ en guise de séparateurs : c'est la convention en vigueur dans le monde *Unix*). Python effectuera automatiquement les conversions nécessaires, suivant que vous travaillez sous *Mac OS*, *Linux*, ou *Windows*[52].

```
>>> from os import chdir
>>> chdir("/home/jules/exercices")
```

La première commande importe la fonction **chdir()** du module **os**. 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 seconde commande provoque le changement de répertoire (**chdir** =*change directory*).

> • *Vous avez également la possibilité d'insérer ces commandes en début de script, ou encore d'indiquer le chemin d'accès complet dans le nom des fichiers que vous manipulez, mais cela risque peut-être d'alourdir l'écriture de vos programmes.*

[51] 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.

[52] 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 : **D:/home/jules/exercices**.

> • ***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.***

## Les deux formes d'importation

Les lignes d'instructions que nous venons d'utiliser sont l'occasion d'expliquer un mécanisme intéressant. Vous savez qu'en complément des fonctions intégrées dans le module de base, Python met à votre disposition une très grande quantité de fonctions plus spécialisées, qui sont regroupées dans des *modules* . Ainsi vous connaissez déjà fort bien le module *math* et le module *tkinter* .

Pour utiliser les fonctions d'un module, il suffit de les importer. Mais cela peut se faire de deux manières différentes, comme nous allons le voir ci-dessous. Chacune des deux méthodes présente des avantages et des inconvénients.

Voici un exemple de la première méthode :

```
>>> import os
>>> rep_cour = os.getcwd()
>>> print rep_cour
C:\Python22\essais
```

La première ligne de cet exemple importe l'*intégralité* du module **os**, lequel contient de nombreuses fonctions intéressantes pour l'accès au système d'exploitation. La seconde ligne utilise la fonction **getcwd()** du module **os**[53]. Comme vous pouvez le constater, la fonction **getcwd()** renvoie le nom du répertoire courant (*getcwd = get current working directory*). Par comparaison, voici un exemple similaire utilisant la seconde méthode d'importation :

```
>>> from os import getcwd
>>> rep_cour = getcwd()
>>> print(rep_cour)
C:\Python31\essais
```

Dans ce nouvel exemple, nous n'avons importé du module **os** que la fonction **getcwd()**. Importée de cette manière, la fonction s'intègre à notre propre code comme si nous l'avions écrite nous-mêmes. Dans les lignes où nous l'utilisons, il n'est pas nécessaire de rappeler qu'elle fait partie du module **os**.

Nous pouvons de la même manière importer plusieurs fonctions du même module :

[53] Le point séparateur exprime donc ici une relation d'appartenance. Il s'agit d'un exemple de la ***qualification des noms*** qui sera de plus en plus largement exploitée dans la suite de ce cours. Relier ainsi des noms à l'aide de points est une manière de désigner sans ambiguïté des éléments faisant partie d'ensembles, lesquels peuvent eux-mêmes faire partie d'ensembles plus vastes, etc. Par exemple, l'étiquette **systeme.machin.truc** désigne l'élément **truc**, qui fait partie de l'ensemble **machin**, lequel fait lui-même partie de l'ensemble **systeme**. Nous verrons de nombreux exemples de cette technique de désignation, notamment lors de notre étude des **classes** d'objets.

```
>>> from math import sqrt, pi, sin, cos
>>> print(pi)
3.14159265359
>>> print(sqrt(5))          # racine carrée de 5
2.2360679775
>>> print(sin(pi/6))        # sinus d'un angle de 30°
0.5
```

Nous pouvons même importer *toutes* les fonctions d'un module, comme dans :

```
from tkinter import *
```

Cette méthode d'importation présente l'avantage d'alléger l'écriture du code. Elle présente l'inconvénient (surtout dans sa dernière forme, celle qui importe toutes les fonctions d'un module) d'encombrer l'espace de noms courant. Il se pourrait alors que certaines fonctions importées aient le même nom que celui d'une variable définie par vous-même, ou encore le même nom qu'une fonction importée depuis un autre module. Si cela se produit, l'un des deux noms en conflit n'est évidemment plus accessible.

Dans les programmes d'une certaine importance, qui font appel à un grand nombre de modules d'origines diverses, il sera donc toujours préférable de privilégier la première méthode, c'est-à-dire celle qui utilise des noms pleinement qualifiés.

On fait généralement exception à cette règle dans le cas particulier du module tkinter, parce que les fonctions qu'il contient sont très sollicitées (dès lors que l'on décide d'utiliser ce module).

## *Écriture 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()**[54]. 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','a')
>>> obFichier.write('Bonjour, fichier !')
>>> obFichier.write("Quel beau temps, aujourd'hui !")
>>> obFichier.close()
>>>
```

### *Notes*

- La première ligne crée l'*objet-fichier* **obFichier**, lequel fait référence à un fichier véritable (sur disque ou disquette) dont le nom sera **Monfichier**. Attention : *ne confondez pas le nom*

[54] Une telle fonction, dont la valeur de retour est un objet particulier, est souvent appelée *fonction-fabrique*.

*de 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** (et dont vous pouvez visualiser le contenu à l'aide d'un éditeur quelconque).

- La fonction **open()** attend deux arguments, qui doivent tous deux être des chaînes de caractères. Le premier argument est le nom du fichier à ouvrir, et le second est le mode d'ouverture. **'a'** indique qu'il faut ouvrir ce 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 est effacé au préalable.
- 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.

## Lecture séquentielle d'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 :

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

Comme on pouvait s'y attendre, 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.

### Notes

- Le fichier que nous voulons lire s'appelle **Monfichier**. L'instruction d'ouverture de fichier devra donc nécessairement faire référence à ce nom là. Si le fichier n'existe pas, nous obtenons un message d'erreur. Exemple :
```
>>> ofi = open('Monficier','r')
IOError: [Errno 2] No such file or directory: 'Monficier'
```
- Par contre, nous ne sommes tenus à aucune obligation concernant le nom à choisir pour *l'objet-fichier*. C'est un nom de variable quelconque. Ainsi donc, dans notre première instruction, nous avons choisi de créer un objet-fichier **ofi**, faisant référence au fichier réel **Monfichier**, 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', 'r')
>>> 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 !
```

Si la fin du fichier est déjà atteinte, **read()** renvoie une chaîne vide :

```
>>> t = ofi.read()
>>> print(t)
```

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

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

> *Dans tout ce qui précède, nous avons admis sans explication que les chaînes de caractères étaient échangées telles quelles entre l'interpréteur Python et le fichier. En réalité, ceci est inexact, parce que les séquences de caractères doivent être converties en séquences d'octets pour pouvoir être mémorisées dans les fichiers, et il existe malheureusement différentes normes pour cela. En toute rigueur, il faudrait donc préciser à Python la norme d'encodage que vous souhaitez utiliser dans vos fichiers : nous verrons comment faire au chapitre suivant. En attendant, vous pouvez tabler sur le fait que Python utilise par défaut la norme en vigueur sur votre système d'exploitation, ce qui devrait vous éviter tout problème pour ces premiers exercices. Si vous obtenez tout de même un rendu bizarre de vos caractères accentués, veuillez l'ignorer provisoirement.*

## *L'instruction break pour sortir d'une boucle*

*Remarque de DM : Les puristes déconseillent d'utiliser cette instruction, car elle nuit à la lisibilité du code. Il faut en tout cas ne pas en abuser.*

Il va de soi que les boucles de programmation s'imposent lorsque l'on doit traiter un fichier dont on ne connaît pas nécessairement le contenu à l'avance. L'idée de base consistera à lire ce fichier morceau par morceau, jusqu'à ce que l'on ait atteint la fin du fichier.

La fonction ci-dessous illustre cette idée. Elle copie l'intégralité d'un fichier, quelle que soit sa taille, en transférant des portions de 50 caractères à la fois :

```
def copieFichier(source, destination):
    "copie intégrale d'un fichier"
    fs = open(source, 'r')
    fd = open(destination, 'w')
    while 1:
        txt = fs.read(50)
        if txt =="":
            break
        fd.write(txt)
    fs.close()
    fd.close()
    return
```

Si vous voulez tester cette fonction, vous devez lui fournir deux arguments : le premier est le nom du fichier original, le second est le nom à donner au fichier qui accueillera la copie. Exemple :

```
copieFichier('Monfichier','Tonfichier')
```

Vous aurez remarqué que la boucle **while** utilisée dans cette fonction est construite d'une manière différente de ce que vous avez rencontré précédemment. Vous savez en effet que l'instruction **while** doit toujours être suivie d'une condition à évaluer ; le bloc d'instructions qui suit est alors exécuté en boucle, aussi longtemps que cette condition reste vraie. Or nous avons remplacé ici la condition à évaluer par une simple constante, et vous savez également[55] que l'interpréteur Python considère comme vraie toute valeur numérique différente de zéro.

Une boucle **while** construite comme nous l'avons fait ci-dessus devrait donc boucler indéfiniment, puisque la condition de continuation reste toujours vraie. Nous pouvons cependant interrompre ce bouclage en faisant appel à l'instruction **break**, laquelle permet éventuellement de mettre en place plusieurs mécanismes de sortie différents pour une même boucle :

```
while <condition 1> :
    --- instructions diverses ---
    if <condition 2> :
        break
    --- instructions diverses ---
    if <condition 3>:
        break
    etc.
```

Dans notre fonction **copieFichier()**, il est facile de voir que l'instruction **break** s'exécutera seulement lorsque la fin du fichier aura été atteinte.

## *Fichiers texte*

Un *fichier texte* est un 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 »[56].

[55] Voir page 54 : Véracité/fausseté d'une expression

[56] Suivant le système d'exploitation utilisé, le codage correspondant au marqueur de fin de ligne peut être différent. Sous Windows, par exemple, il s'agit d'une séquence de deux caractères (retour chariot et saut de

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. Nous donnons un petit exemple de traitement de fichier binaire un peu plus loin, mais dans le cadre de ce cours, nous nous intéresserons presqu'exclusivement aux fichiers texte.

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", "w")
>>> f.write("Ceci est la ligne un\nVoici la ligne deux\n")
>>> f.write("Voici la ligne trois\nVoici la ligne quatre\n")
>>> 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.

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

```
>>> f = open('Fichiertexte','r')
>>> 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*

- 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 (à l'aide d'une boucle **while**, par exemple) pour en extraire les chaînes individuelles.
- 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,

ligne), alors que dans les systèmes de type Unix (comme Linux) il s'agit d'un seul saut de ligne, MacOS pour sa part utilisant un seul retour chariot. 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. Pour la lecture, Python interprète correctement chacune des trois conventions (qui sont donc considérées comme équivalentes).

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.

- 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.

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):
    "recopier un fichier en éliminant les lignes de remarques"
    fs = open(source, 'r')
    fd = open(destination, 'w')
    while 1:
        txt = fs.readline()
        if txt =='':
            break
        if txt[0] != '#':
            fd.write(txt)
    fs.close()
    fd.close()
    return
```

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. Exemple :

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

## 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))
```

Nous verrons plus loin qu'il existe d'autres possibilités pour convertir des valeurs numériques en chaînes de caractères (voir à ce sujet : *Formatage des chaînes de caractères*, page 140). Mais la question n'est pas vraiment là. Si nous enregistrons les valeurs numériques en les transformant d'abord en chaînes de caractères, nous risquons de ne plus pouvoir les retransformer correctement en valeurs numériques lorsque nous allons relire le fichier. Exemple :

```
>>> a = 5
>>> b = 2.83
>>> c = 67
>>> f = open('Monfichier', 'w')
>>> f.write(str(a))
>>> f.write(str(b))
>>> f.write(str(c))
```

```
>>> f.close()
>>> f = open('Monfichier', 'r')
>>> 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**[57]. 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, et d'autres types de données encore que nous étudierons plus loin.

***Attention :*** les fichiers traités à l'aide des fonctions du module *pickle* ne seront pas des des fichiers texte, mais bien des *fichiers binaires*[58]. 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 3e ligne de notre exemple), et l'argument 'rb' pour ouvrir un fichier binaire en lecture (comme à la 8e ligne de notre exemple).

[57] En anglais, le terme *pickle* signifie « conserver ». Le module a été nommé ainsi parce qu'il sert effectivement à enregistrer des données en conservant leur type.

[58] Dans les versions de Python antérieures à la version 3.0, le module *pickle* s'utilisait avec des fichiers texte (mais les chaînes de caractères étaient traitées en interne avec des conventions différentes). Les fichiers de données créés avec ces différentes versions de Python ne sont donc pas directement compatibles. Des convertisseurs existent.

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.

## Gestion des exceptions : les instructions try – except – else

Les *exceptions* sont les opérations qu'effectue un interpréteur ou un compilateur lorsqu'une erreur est détectée au cours de l'exécution d'un programme. En règle générale, l'exécution du programme est alors interrompue, et un message d'erreur plus ou moins explicite est affiché. Exemple :

```
>>> print(55/0)
ZeroDivisionError: int division or modulo by zero
```

> *D'autres informations complémentaires sont affichées, lesquelles indiquent notamment à quel endroit du script l'erreur a été détectée, mais nous ne les reproduisons pas ici.*

Le message d'erreur proprement dit comporte deux parties séparées par un double point : d'abord le type d'erreur, et ensuite une information spécifique de cette erreur.

Dans de nombreux cas, il est possible de prévoir à l'avance certaines des erreurs qui risquent de se produire à tel ou tel endroit du programme et d'inclure à cet endroit des instructions particulières, qui seront activées seulement si ces erreurs se produisent. Dans les langages de niveau élevé comme Python, il est également possible d'associer un mécanisme de surveillance à *tout un ensemble d'instructions*, et donc de simplifier le traitement des erreurs qui peuvent se produire dans n'importe laquelle de ces instructions.

Un mécanisme de ce type s'appelle en général *mécanisme de traitement des exceptions*. Celui de Python utilise l'ensemble d'instructions **try** - **except** – **else**, qui permettent d'intercepter une erreur et d'exécuter une portion de script spécifique de cette erreur. Il fonctionne comme suit.

Le bloc d'instructions qui suit directement une instruction **try** est exécuté par Python *sous réserve*. Si une erreur survient pendant l'exécution de l'une de ces instructions, alors Python annule cette instruction fautive et exécute à sa place le code inclus dans le bloc qui suit l'instruction **except.** Si aucune erreur ne s'est produite dans les instructions qui suivent **try**, alors c'est le bloc qui suit l'instruction **else** qui est exécuté (si cette instruction est présente). Dans tous les cas, l'exécution du programme peut se poursuivre ensuite avec les instructions ultérieures.

Considérons par exemple un script qui demande à l'utilisateur d'entrer un nom de fichier, lequel fichier étant 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 éventuellement que l'utilisateur puisse essayer d'entrer un autre nom.

```
filename = input("Veuillez entrer un nom de fichier : ")
try:
    f = open(filename, "r")
except:
```

```
    print("Le fichier", filename, "est introuvable")
```

Si nous estimons que ce genre de test est susceptible de rendre service à plusieurs endroits d'un programme, nous pouvons aussi l'inclure dans une fonction :

```
def existe(fname):
    try:
        f = open(fname,'r')
        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.")
```

Il est également possible de faire suivre l'instruction **try** de plusieurs blocs **except**, chacun d'entre eux traitant un type d'erreur spécifique, mais nous ne développerons pas ces compléments ici. Veuillez consulter un ouvrage de référence sur Python si nécessaire.

## *Exercices*

9.1 Écrivez un script qui permette 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.
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).

9.2 Écrivez un script qui génère automatiquement un fichier texte contenant les tables de multiplication de 2 à 20 (chacune d'entre elles incluant 20 termes seulement).

9.3 Écrivez un script qui compare les contenus de deux fichiers et signale la première différence rencontrée.

9.4 Écrivez un script qui permette d'encoder un fichier texte dont les lignes contiendront chacune les noms, prénom, adresse, code postal et n° de téléphone de différentes personnes (considérez par exemple qu'il s'agit des membres d'un club).

9.5 Écrivez un script qui recopie le fichier utilisé dans l'exercice précédent, en y ajoutant la date de naissance et le sexe des personnes (l'ordinateur devra afficher les lignes une par une et demander à l'utilisateur d'entrer pour chacune les données complémentaires).

9.6 Considérons que vous avez fait les exercices précédents 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 lignes qui correspondent à un code postal bien déterminé.