ITC – cours n°9

Chaînes de caractères – fichiers

Pour manipuler de plus grandes quantités de données et les enregistrer entre deux sessions de travail, on utilise des fichiers : nous allons voir les méthodes de base permettant la lecture et l'écriture de ces fichiers texte.

I. Chaînes de caractères

1.1. Définition

Les chaînes de caractère permettent de stocker des lettres, mots ou phrases. On les déclare à l'aide de guillemets simples ou triples, ou d'apostrophes :

```
nom_du_prof = 'Adrien Licari-Guillaume'      # première méthode
nom_du_prof = "Adrien Licari-Guillaume"      # deuxième méthode
nom_du_prof = """Adrien Licari-Guillaume"""  # troisième méthode
```

La méthode avec les triples guillemets est plus lourde, mais elle est la plus robuste : la chaîne de caractères ainsi définie peut elle-même contenir des guillemets ou des apostrophes :

```
texte_long = """
Il toqua à la porte ; c'est alors que la porte
répondit : "Le mot de passe SVP ?"
"""
```

est valable, tandis que

```
essai = "une telle chaîne contenant " est incorrecte "
```

produira l'erreur `SyntaxError: Invalid syntax`.

Attention : la chaîne de caractère `"2.5"` n'est pas un nombre, mais bien une chaîne de caractère. Ainsi, l'opération suivante soulève une erreur :

```
In [1]: a = 2
In [2]: b = "2.5"
In [3]: c = a + b
TypeError: unsupported operand type(s) for + : 'int' and 'str'
```

ce qui est bien normal : comment sommer un nombre et une chaîne de caractère ? En revanche, on peut transformer `b` en nombre, puisque la chaîne de caractère peut être **interprétée comme** un nombre : on écrit pour cela `float(b)` pour un réel et `int(b)` pour un entier :

```
In [1]: a, b = 2, "2.5"
In [2]: b_reel = float(b)
In [3]: c = a + b_reel  # pas d'erreur, on obtient c == 4.5
```

On prendra bien garde à faire la distinction entre un nom (qui n'est pas entouré de guillemets) et une chaîne de caractères :

```
In [1]: chaîne = "Hello"
In [2]: chaîne == "Hello"
Out[1]: True
In [3]: chaîne == "chaîne"
Out[2]: False
```

1.2. Parcours et manipulation de chaînes

On retrouve pour parcourir ou manipuler les chaînes des fonctions très similaires à celles des listes : l'opérateur **concaténation** `+`, l'opérateur répétition `*` ainsi que la fonction `len` :

```
In [1]: a, b = "Bonjour", " les MPSI-1"
In [2]: c = a + b   # c contient "Bonjour les MPSI-1"
In [3]: d = "x"*50  # d contient la répétition de 50 'x'
In [4]: len(d)
Out[1]: 50
```

On peut également accéder à un caractère ou un ensemble avec l'opérateur `[ ]` et les tranches :

```
In [5]: c[3]
Out[2]: 'j'
In [6]: c[3:7]
Out[3]: 'jour'
```

Ajoutons qu'il existe une fonction de l'objet `str` qui recherche un caractère de séparation et découpe la chaîne en sous-chaînes selon ce caractère : `split` :

```
In [1]: ch1, ch2 = "Bonjour les amis", "Hé ! Tagada ! Tsoin !"
In [2]: ch1.split()  # par défaut le caractère est " "
Out[1]: ['Bonjour', 'les', 'amis']
In [3]: ch2.split('!')
Out[2]: ['Hé ', ' Tagada ', ' Tsoin ', '']
```

1.3. Gestion en mémoire

Contrairement aux listes, **les chaînes sont immuables** ; ainsi on ne peut pas changer le 4e caractère avec une instruction su type `c[3] = 'x'`, cela soulèvera une erreur. Autrement dit : une chaîne ne peut pas être en partie modifiée, on ne peut que créer une nouvelle chaîne et y ré-affecter l'étiquette précédente :

```
ch = "Bonjour"
ch[2] = "o"                 # impossible, str immuables -> erreur
ch = ch[:2] + "o" + ch[3:]  # possible: on créé une nouvelle str
```

> En conséquence, il n'existe pas d'équivalent de `liste.append(valeur)` pour les chaînes de caractère.

Une conséquence importante est la suivante : on ne peut pas modifier une variable externe depuis une fonction :

```
def fonction(chaîne):
    chaîne += " !"  # créé une nouvelle str par concaténation et remplace
                    # la référence locale
ch = "Hello"
fonction(ch)  # ch n'est pas modifiée
```

1.4. Hors-programme mais si pratique : le formattage

Pour insérer des nombres/tuples ou autres types dans une chaîne de façon appropriée, on peut utiliser la fonction `format` : on donne l'emplacement à remplacer entre accolades, et on donne l'ensemble des valeurs à la fin :

```
In [1]: n, console = 12, "PS4"
In [2]: print("Je possède { :} jeux sur ma { :}.".format(n, console))
Out[1]: Je possède 12 jeux sur ma PS4.
```

On peut alors utiliser des options entre les accolades pour mettre en forme à notre convenance, par exemple :

- le nombre de chiffres après la virgule pour les réels ;
- ajouter des espaces pour que l'affichage prenne exactement le même nombre de caractères ;
- ajouter des 0 devant les entiers ;
- et plein d'autres choses...

Si vous avez besoin d'une mise en forme particulière, recherchez en ligne de l'aide sur la fonction `format` . Dans le cadre de ce cours, il vous est recommandé de simplement retenir qu'elle existe (afin de chercher quand elle pourrait vous être utile), et que l'on peut gérer l'affichage des réels ainsi :

```
In [1]: x = 1/3
In [2]: print("simple : { :}".format(x))
Out[1]: simple : 0.3333333333333333
In [3]: print("3 chiffres après la virgule : { :.3f}".format(x))
Out[2]: 3 chiffres après la virgule : 0.333
In [4]: print("notation exponentielle, 2 chiffres : { :.2e}".format(x))
Out[3]: notation exponentielle, 2 chiffres : 3.33e-01
```

II. Entrées-sorties console : print et input

2.1. Sortie console : print

Si on veut faire apparaître une chaîne de caractères dans la console, on peut utiliser la fonction `print` :

```
print("Bonjour à toi l'ami de mon amie !")
```

La fonction print peut également prendre en paramètre des variables d'autres types : elle va alors les **convertir** en `str` , puis l'affiche. La conversion est assez naturelle pour les nombres (par exemple le flottant `4.56` est converti en `'4.56'`), les booléens et les listes (on affiche les crochets et des virgules entre chaque élément), mais elle existe aussi pour des types moins évidents comme les fonctions :

```
a, b, c = 3, [4, 2, 4], "Bonjour"

def fonction():
    """
    Une docstring
    """
    return False

print(a)
print(b)
print(c)
print(len)
print(fonction)
--------------------------------------------------
3
[4, 2, 4]
Bonjour
<built-in function len>
<function fonction at 0x7f15940425e0>
```

On peut donner plusieurs arguments pour les afficher à la suite (`print` les sépare alors avec une espace) :

```
In [1]: x = 3
In [2]: print("la variable x vaut", 3)
Out[1]: la variable x vaut 3
```

La fonction `print` a pour seul et unique but d'afficher un résultat afin que celui-ci soit lu par un être humain : il s'ensuit que si vous ne comptez pas faire quelque chose/avoir une réflexion à partir de ce qui est affiché, en bref si vous ne comptez pas le lire, alors la fonction `print` n'a probablement aucun intérêt dans votre contexte.

En particulier, il est hors de question de confondre `print` et `return` : le second permet au reste du code de récupérer le résultat d'une fonction, le premier non...

2.2. Caractères spéciaux

Certains caractères sont présent dans la chaîne mais ne sont pas censés être affichés tel quels : ils servent à la mise en forme. Il s'agit des caractères spéciaux `'\n '` pour les retours à la ligne et `'\t` pour la tabulation. On voit la différence en affichant la variable directement ou en utilisant `print`, qui va interpréter cela :

```
In [1]: x = "Bonjour\tà tous\nComment allez-vous?"
In [2]: x
Out[1]: 'Bonjour\tà tous\nComment allez-vous?'
In [3]: print(x)
Out[2]: Bonjour     à tous
        Comment allez-vous?
```

2.3. Entrée console : input

À l'inverse, on peut demander en direct une valeur à l'utilisateur avec la fonction `input` : par exemple l'instruction

```
âge = input("Inscrivez votre âge svp : ")
```

affichera le message « Inscrivez votre âge svp : » dans la console, puis attendra que l'utilisateur saisisse quelque chose ; une fois cette saisie effectuée, ce qui a été récupéré sera inscrit dans la variable `âge`, **qui est donc une chaîne de caractères**. Si la valeur entrée peut être convertie en nombre, booléen ou autre, on pourra le faire avec une conversion explicite :

```
âge = int(âge)    # pour un entier
âge = float(âge)  # pour un réel
rep = bool(rep)  # pour un booléen
```

Aucune vérification n'est faite à ce stade : si l'utilisateur répond « -425.8 » ou encore « Le même âge que le capitaine », ce sont ces valeurs qui seront stockées sous l'étiquette `âge` ; et si cela rend la suite du programme incohérent, c'est à vous de prendre des précautions. De même, si vous essayez de convertir en nombre une saisie qui ne peut pas l'être (comme « coucou ») Python renverra une erreur :

```
ValueError: could not convert string to float: 'coucou'.
```

Globalement, on se servira aussi peu que possible de cette fonction.

III. Fichiers texte

Le programme officiel indique que vous devez savoir comment utiliser les fonctions proposées dans la suite, mais que leur fonctionnement doit vous être rappelé au concours par une documentation succincte. Ainsi, il faut vous rappeler des grands principes et des différences possibles plutôt que de la spécification exacte de `read` ou `readlines` ...

3.1. Principes généraux

Un fichier texte contient simplement des caractères (pas de mise en forme de type gras, italique etc... : ceci correspond à des fichiers de **traitement de texte**).

Pour interagir avec un fichier dans un code Python, il faut créer une variable d'un type adapté, nommé flux d'entrée-sortie, à l'aide de l'instruction `open` :

```
fichier = open("nom_du_fichier.txt", 'w')
```

Le premier paramètre est le nom du fichier, qui doit se trouver (s'il existe) dans le répertoire de travail courant. Dans SPyder, ce répertoire est indiqué (et peut être modifié) dans la barre d'adresse en haut à droite de la fenêtre. Le second représente le mode d'ouverture du fichier, qui indique les opérations possibles :

`'r'`	(*read*) ouverture pour lecture seule (option par défaut). Le ficher doit exister dans le répertoire courant
`'w'`	(*write*) ouverture pour écriture seule. Lorsque le fichier n'existe pas il est créé dans le répertoire courant (ou le répertoire donné) ; si le fichier existe déjà il est écrasé.
`'a'`	(*append*) ouverture pour écriture seule. Lorsque le fichier n'existe pas il est créé dans le répertoire courant ; si le fichier existe alors les données sont écrites à la suite.
`'r+'`	ouverture pour lecture et écriture. Le fichier doit exister, il est alors réactualisé.

Le nom du fichier à fournir est le nom complet, c'est-à-dire **avec son extension** s'il en a une. Pour les fichiers de texte brut (ceux que nous allons employer), l'usage est de mettre une extension `.txt`, ou `.dat` si cela représente des données à analyser.

Par la suite, toutes les instructions liées au fichier seront de la forme `fichier.fonction(paramètr` pour les fonctions qui agissent sur une liste ou un dictionnaire. De même, une fois fini l'utilisation du flux, il est préférable de le refermer avec l'instruction `close` :

```
fichier.close()
```

3.2. Écriture

Une fois un fichier ouvert en mode écriture (`'w'` ou `'a'` en général), on peut le « remplir » avec l'instruction `write` :

```
fichier.write("C'est une bonne situation ça, scribe ?")
```

Attention cependant : cette instruction ne rajoute pas de retour à la ligne. Si on veut demander un tel retour à la ligne, il faut le demander explicitement en écrivant le caractère `'\n '` .

3.3. Lecture

Si un fichier est ouvert en lecture (`'r'` en général), on peut récupérer tout le contenu avec la fonction `read` : cette fonction renvoie tout le contenu du fichier sous forme d'une unique chaîne de caractères :

```
In [1]: tout_le_texte = fich.read()
In [2]: tout_le_texte
Out[1]: 'Bonjour\nÇa va ?\n'
In [2]: print(tout_le_texte)
Out[2]: Bonjour
        Ça va ?
```

On peut également utiliser la fonction `readline` pour ne lire qu'une seule ligne : la fonction lit la ligne courante, renvoie une chaîne de caractère correspondante, et passe à la ligne suivante pour le prochain appel. Si elle atteint la fin du fichier, elle renvoie une chaîne vide. Une façon de lire tout un fichier peut être la suivante :

```
ligne = fich.readline()
while ligne != '':
    # instructions utiles avec la ligne en cours : affichage ou autre
    ligne = fich.readline()
```

Une dernière méthode permet de récupérer immédiatement toutes les liges dans une liste de `str` : il s'agit de la fonction `readlines` :

```
ensemble_lignes = fich.readlines()
for ligne in ensemble_lignes:
    # instructions utiles avec la ligne en cours : affichage ou autre
```

ou encore plus simplement

```
for ligne in fich.readlines():
    # instructions utiles avec la ligne en cours : affichage ou autre
```

Dans tous les cas, le caractère `'\n '` de fin de ligne **est gardé** par ces fonctions. Dans le cas de `readline` et `readlines` , on peut simplement s'en débarrasser puisqu'il s'agit par construction du dernier caractère de la chaîne.

3.4. Hors-programme : déplacements et lecture-écriture

Lorsqu'on utilise `readline` ou `write` , Python garde en mémoire sa position dans le fichier sous forme d'un curseur au caractère numéro n (initialement 0, puis qui avance). On peut connaître cette position avec la fonction `tell()` et se placer à la position n de son choix avec `seek(n)`. Ces fonctions ne sont cependant pas très pratiques d'usage. Une utilité courante est de revenir au début du fichier sans avoir à le fermer et le rouvrir, avec `fich.seek(0)`.

IV. Exercices

4.1. Chaînes

1 – Définir trois chaînes de caractères avec votre nom dans la première, prénom dans la deuxième, et une phrase d'un film/une chanson/autre que vous aimez.

2 – Définir une nouvelle chaîne rassemblant votre prénom et votre nom à partir des variables précédentes.

3 – Définir une chaîne contenant 5 espaces, 8 signes − puis 4 points d'exclamation.

4 – Définir une chaîne contenant les 6 caractères du 5^e au 10^e (inclus) de la phrase.

5 – Écrire une fonction `compter_lettre` qui prend en paramètres une lettre et un texte, et renvoie le nombre d'occurrences de la lettre dans le texte.

6 – Écrire une fonction `chercher_séquence` qui prend en paramètres une chaîne `recherche` et un texte, et renvoie `True` si la chaîne est une sous-chaîne du texte, `False` sinon.

7 –Un palindrome est un texte qui peut être lu indifféremment dans les deux sens, comme « kayak » ou « elle ». Écrire une fonction récursive `est_palindrome` qui renvoie `True` si une chaîne est un palindrome et `False` sinon.

8 – Chaque caractère (lettres, ponctuation etc) est associé dans Python à un nombre ; par exemple, `'A'` a le numéro 65, `'!'` le 33...La fonction fournie `ord` permet de récupérer ces numéros. La fonction `chr` fait l'opération inverse. Les lettres de l'alphabet latin sont rangées dans l'ordre, en majuscules et en minuscules.

8.1 – Utiliser la fonction `ord` pour déterminer les numéros associés à `'A'`, `'a'` et `'?'`.

8.2 – Utiliser les résultats précédents et la fonction `chr` pour créer deux listes `alph_min` et `alph_maj` contenant respectivement toutes les lettres de l'alphabet en minuscule et en majuscule.

4.2. Fichiers

1 – Ouvrir le fichier `moliere.txt` et afficher son contenu ligne à ligne.

2 –Écrire une fonction `compter_répliques(fichier, protagoniste)` qui prend en paramètres le nom du fichier à lire et le nom d'un personnage et compte le nombre de répliques qu'il prononce dans l'extrait.

3 – Écrire une procédure `écrire_scores(fichier, noms, scores)` qui prend en paramètres le nom du fichier à écrire, une liste de noms et une liste d'entiers, et écrit pour chaque couple nom/score une ligne dans le fichier du type « Picsou : 203874 ». Quelle précondition doit être respectée pour cette procédure ?

4 – Utiliser les deux fonctions précédentes pour compter l'ensemble des répliques de Sganarelle, Valère et Lucas, et les écrire dans un fichier `nb_repliques.txt`.

5 – Expliquer pourquoi le code suivant ne donne pas lieu à une boucle infinie :

```
fich = open("moliere.txt", 'r')
while fich.readline() != '':
a = 0
```