La boucle while et les entrées-sorties fichier

1. La boucle while

Nous allons présenter la structure de boucle utilisant le mot-clé `while`. Comme pour la condition `if`, la boucle `while` nécessite une condition indiquant si son sous-bloc doit être ré-exécuté ou non. Tant que la condition est vraie, son sous-bloc est relancé. Prenons un exemple :

```
Ouverture d'un fichier texte
while ( la fin du fichier n'est pas atteinte ) :
    Charger la prochaine ligne dans le fichier et l'afficher
Fermeture du fichier
```

Cette approche permet de parcourir l'intégralité d'un fichier et de traiter les informations qu'il contient. Il n'est pas possible de connaître la quantité de lignes à lire dans un fichier. Ainsi, on ne peut utiliser une boucle `for` avec indice. Certes, la taille du fichier est connue, mais on ne peut pas en déduire le nombre de lignes. La boucle `while` s'adapte très bien à ce scénario.

Il est toujours possible de charger l'intégralité d'un fichier dans une chaîne de caractères et ensuite de parcourir cette chaîne avec une boucle `for` avec indice. Mais cette approche est une astuce qui devient dangereuse dès que la taille du fichier à traiter devient importante.

2. Ne pas remplacer une boucle for par une boucle while

On peut écrire le code suivant :

```
K = 10
while K < 13 :
     print(K)
     K += 1
print("Terminé")
```

Cette syntaxe est correcte. Par contre, elle est maladroite. Les trois éléments essentiels d'une boucle `for` sont présents : l'initialisation de l'indice de départ avec `K=10`, l'incrémentation présente en fin de bloc avec `K += 1` et l'indice de fin venant de la condition `K<13`. Dans ce cas de figure, il faut préférer l'utilisation d'une boucle `for` et écrire :

```
for i in range(10,13):
   print(K)
print("Terminé")
```

Pourquoi ? Au final, les deux syntaxes sont correctes et les deux exemples produisent exactement le même résultat. Il y a une différence, cependant peu évidente ici. Supposons que dans l'exemple du `while`, le corps de boucle fasse 50 lignes. L'incrémentation `K +=1` est alors très éloignée de l'instruction `while`. De plus, il se peut aussi que plusieurs lignes soient présentes entre l'affectation `K=10` et l'instruction `while`. Dans cette configuration, les trois points essentiels de la boucle, initialisation, incrémentation et condition, se retrouvent très éloignés dans le code les uns des autres et ce n'est pas bon pour la lisibilité. La boucle `for` a cet avantage, non pas d'être plus rapide, mais

de rassembler ces trois informations essentielles sur la même ligne, ce qui est très intéressant tactiquement lorsque l'on relit son code ou lorsque l'on recherche une erreur.

3. Lire et écrire dans un fichier

Pour écrire sur le disque, la fonction incontournable est `open()`. Son premier argument indique le chemin du fichier sur le disque, et le deuxième le mode d'ouverture. Pour créer un fichier, vous avez le choix entre deux modes :

- le mode `"w"` (abréviation du terme "write")
- le mode `"a"` (abréviation du terme "append")

Le premier mode remet à zéro le fichier, quelle que soit la configuration : s'il n'existe pas, il est créé, et s'il existe, les données existantes sont écrasées. Le mode append, lui, nécessite que le fichier soit existant, sinon une erreur est levée. Une fois le fichier ouvert en mode append, l'écriture des informations se fait à la suite des données déjà présentes dans le fichier. Ce mode permet d'ajouter des données dans un fichier existant.

Si la fonction `open()` réussit, elle retourne un objet fichier. À travers cet objet, on accède à des fonctions permettant de le manipuler, comme la fonction `write()` pour ajouter de l'information et la fonction `close()` pour le fermer.

L'écriture dans le fichier se fait au moyen de la fonction `write()` à partir de la variable stockant l'objet fichier. Contrairement à la fonction `print()`, cette fonction ne génère pas un retour à la ligne automatique, il faut donc le demander explicitement grâce au caractère spécial `\n`. La fonction `close()`, appelée depuis l'objet fichier, permet de terminer la séquence d'écriture : toutes les données en attente de transfert sont envoyées au disque et ensuite le fichier est libéré. Voici un exemple de fonctionnement :

```
f = open("C:\\Users\\MonNomDeCompte\\Desktop\\test.txt", "w") #PC
f = open("/Users/MonNomDeCompte/Desktop/test.txt","w")  # MAC
f.write("Bonjour !!!\n")
f.write("Ligne\n")
f.write("Terminé")
f.close()
```

Voici le contenu du fichier après le lancement :

```
Bonjour !!!
Ligne
Terminé
```

Pour voir le contenu d'un fichier avec une extension .txt, il suffit de double cliquer sur son icône présente sur le Bureau pour l'ouvrir dans l'éditeur par défaut du système.

Pensez à remplacer `MonNomDeCompte` par votre nom d'utilisateur sur la machine. Si vous ne le connaissez pas, lancez le Terminal. Sous Windows, le Terminal va afficher le début de votre chemin : C:\Users\MonNom, directement dans la fenêtre. Sous Mac, dans la fenêtre du Terminal, tapez la commande **pwd** suivie d'un appui sur la touche [Entrée] pour obtenir le début de votre chemin. Le répertoire Desktop sélectionne le Bureau, ainsi le fichier test.txt sera visible à l'écran juste après le lancement de ce script.

Sous Windows, les chemins indiquant la position d'un fichier doivent contenir des doubles barres \\. Cela vient du fait que dans une chaîne de caractères, le symbole \ indique la présence d'un caractère spécial. Par exemple, le code \n désigne un retour à la ligne. Ainsi, pour générer le caractère \ dans une chaîne, il faut utiliser le caractère spécial \\.

Dans tous les cas, vous devez choisir un chemin dans lequel vous avez les droits d'écriture, sinon une exception va être levée. Si vraiment vous n'y arrivez pas, supprimez le chemin et donnez juste le nom du fichier "test.txt" à la fonction `open()`. Cela marchera toujours, mais le fichier sera créé dans un endroit qui dépend de l'éditeur que vous utilisez et de votre environnement. Il faudra le rechercher sur le disque pour trouver son emplacement. La bonne nouvelle est que les sauvegardes suivantes seront faites au même endroit.

Changez le mode `"w"` pour le mode `"a"` et relancez le programme. Voici le contenu du fichier après ce deuxième lancement :

```
Bonjour !!!
Ligne 1
TerminéBonjour !!!
Ligne 1
Terminé
```

Lors du deuxième lancement utilisant le mode append, les informations sont ajoutées à la suite des informations déjà présentes dans le fichier. Comme il n'y avait aucun retour à la ligne derrière le mot "Terminé", l'insertion du mot "Bonjour" se fait juste derrière lui.

Un mécanisme de sécurité existe sur l'accès des fichiers car ce sont des ressources particulières. En effet, lorsque vous ouvrez un fichier pour le lire, normalement cela bloque son accès en écriture pour tous les autres programmes. Cela garantit que votre lecture peut se passer sans encombre. Car si un programme commence à modifier le fichier pendant que vous le lisez, l'accident est imminent. Cependant, plusieurs programmes peuvent avoir un accès en lecture sur le même fichier. À ce niveau, il n'y a aucun risque d'erreur, ils partagent une même ressource. Lorsqu'un programme demande l'accès en écriture à un fichier, il verrouille l'accès en lecture et en écriture pour tous les autres programmes. Ainsi, aucun autre programme ne pourra écrire dedans. La libération de ce verrou intervient lors de l'appel de la fonction `close()`. Si vous l'oubliez, cela veut dire qu'aucun autre programme ne pourra accéder à ce fichier et vous êtes quitte pour éteindre et redémarrer votre machine. Cette astuce antédiluvienne a le mérite d'effacer tous les verrous en cours sur la machine.

Pour la lecture, la logique est la même que pour l'écriture. Il faut d'abord demander l'accès en lecture au fichier en mode `"r"`, puis lire le contenu ligne à ligne et enfin refermer le fichier une fois terminé. À ce niveau, nous ne savons pas le nombre de lignes présentes dans le fichier. On ne peut donc utiliser une boucle `for` avec indice. L'utilisation d'une boucle `while` dans cette configuration s'avère pertinente, car la logique consiste à écrire une boucle de la forme : "tant que j'ai pu lire de l'information, alors je continue." Nous utilisons, à travers l'objet fichier, la fonction `readline()` retournant une chaîne de caractères correspondant à la ligne en cours dans le fichier. S'il n'y a plus de lignes à lire, elle retourne une chaîne vide. Ainsi, la condition de la boucle `while` détectant la fin de fichier sera : `len(ligne) > 0`. Il reste juste le cas particulier du démarrage à gérer. En effet, le test de la boucle `while` n'a aucun sens car la variable `ligne` n'a pas été initialisée. Pour que la boucle `while` puisse fonctionner correctement, il faut donc effectuer une première lecture avant de commencer la boucle. Ainsi, la variable `ligne` sera positionnée : si le fichier n'est pas vide, elle contiendra la première ligne ; s'il est vide, elle contiendra une chaîne vide et ainsi la boucle ne sera pas lancée.

Dans le corps de boucle, l'enchaînement logique est un peu inversé. Dans une boucle `for`, on a tendance à trouver : lecture de la ligne, puis affichage de cette ligne. Dans une boucle `while`, comme la première ligne a été chargée en dehors de la boucle, on trouve : affichage de la ligne, puis lecture de la ligne suivante qui met à jour le contenu de la variable `ligne` pour le prochain test de la boucle `while`. Ainsi, si la fin de fichier est atteinte, cette variable contient une chaîne vide et la boucle `while` ne se relance pas. Voici le code utilisant la boucle `while` pour une lecture de fichier :

```
f = open("C:\\Users\\MonNomDeCompte\\Desktop\\test.txt", "r") #PC
f = open("/Users/MonNomDeCompte/Desktop/test.txt","r")  # MAC
ligne = f.readline()

while len(ligne) > 0:
 print("Lecture : " , ligne)
 ligne = f.readline()
f.close()
```

Ce qui donne en sortie :

```
Lecture : Bonjour !!!

Lecture : Ligne 1

Lecture : TerminéBonjour !!!

Lecture : Ligne 1

Lecture : Terminé
```

Pourquoi avons-nous des lignes vides affichées entre nos lignes de texte ? La fonction `readline()` capture le texte présent sur la ligne, mais aussi le caractère spécial `"\n"` de retour à la ligne présent dans le fichier à la fin de cette ligne. Ainsi, lors de l'affichage, un premier retour à la ligne se déclenche, c'est celui déjà présent dans le fichier texte. Ensuite, par défaut, la fonction `print` génère un retour à la ligne et ce deuxième retour génère la ligne blanche. Si vous voulez avoir le même contenu affiché que celui présent dans le fichier texte, il faut utiliser le paramètre optionnel de la fonction `print()` : `end = ""`.