Travaux pratiques d'introduction 7

Les boucles `while`

Informatique tronc commun MPSI et PCSI

I Rappels de cours

I.1 Les boucles itératives

Une *boucle* est une syntaxe permettant d'écrire des instructions une seule fois et d'en demander la répétition. Il y en a de deux sortes :

- les boucles**`for`**, si le nombre de répétitions des instructions est connu d'avance
- les boucles**`while`**, si les instructions doivent être répétées jusqu'à ce qu'une condition cesse d'être vérifiée, peu importe combien de fois.

Chaque répétition des instructions s'appelle un "tour de boucle" ou une "itération".
Bien sûr, les boucles**`for`** peuvent être vues comme un cas particulier des boucles**`while`** (il suffit que la condition porte sur le nombre de tours), mais quand une boucle**`for`** est possible, elle est préférable à la boucle**`while`** au sens où elle assure d'éviter le problème des boucles infinies.

I.2 Syntaxe de la boucle `while`

La syntaxe fait intervenir le mot clef **`while`** et une clause à valeur booléenne (typiquement un test booléen) :

À retenir : Syntaxe de la boucle while

```
while clause:
    ...
    instructions
    ...
```

Tant que (while) la clause vaut `True`, la boucle continue ses itérations.

I.3 Dangers de la boucle infinie

Une cause d'erreur très classique est que la clause ne puisse jamais devenir égale à `False`, de sorte que la boucle ne s'arrêtera jamais ("boucle infinie"). Il faudra alors utiliser votre IDE (Integrated Development Environnement) pour interrompre en force l'exécution du programme.

À retenir : Condition de sortie

Quand vous utilisez une boucle **`while`**, assurez-vous que la clause peut réellement devenir égale à `False`.

À titre d'exemple, créons ensemble des "virus", qui mobilisent les ressources de l'ordinateur, sans jamais s'arrêter. Ces virus sont très basiques, puisqu'il suffira de forcer leur arrêt pour empêcher toute nuisance.

```
compteur = 0
while True:
    compteur += 1
```

Ce virus ne fait rien, hormis calculer sans cesse la valeur de `compteur`, ce qui ralentit l'exécution des autres tâches.
En enregistrant une donnée à chaque tour de boucle, ce virus peut également remplir inutilement la place mémoire.

```
liste_entiers = []
entier = 0
while entier > -2:
    liste_entiers.append()
    entier += 1
```

Heureusement, à la fermeture du programme, ces données seront supprimées. Cela devient plus gênant si elles sont conservées, puisqu'alors la mémoire disponible sera vite saturée ! C'est ce qu'effectue le programme suivant en créant un nouveau fichier à chaque tour de boucle !

```
num = 0
while num > -2:
    nom_fichier = "merci_virus"+str(num)+".txt"
    fichier = open(nom_fichier,"w")
    fichier.write("Et encore un fichier créé !")
    fichier.close()
    num += 1
```

Evidemment, en tant que programmeur, une boucle infinie est rarement voulue, et est souvent due à une erreur de syntaxe triviale comme par exemple remplacer un > par un <...

I.4 Astuces de programmation avec la boucle `while`

I.4.a Utilisation d'un compteur temporaire

Durant le tâtonnement inhérent à la création d'un programme, il peut être intéressant d'utiliser un compteur temporaire, initialisé à 0 et incrémenté à chaque passage dans la boucle **`while`** (Ce compteur compte donc le nombre de passages dans la boucle). Il suffit alors d'ajouter à la clause de la boucle**`while`**, "**`and compteur < ...`**" une certaine valeur choisie (10, 20 ou 100), pour être sûr de sortir de la boucle.

```
compteur = 0
while condition and compteur < 10:
    ...
    instructions
    ...
    compteur += 1
```

Dès que l'on s'est assuré que le programme fonctionne correctement, les instructions relatives au compteur peuvent alors être supprimées.

I.4.b Préférer les boucles `for` aux boucles `while`

Bien que plus puissantes que les boucles **`for`**, les boucles **`while`** nécessitent un grand soin pour s'assurer de la sortie de la boucle, elles ne seront utilisées que lorsqu'elles sont nécessaires.

I.5 Instructions `break`, `continue` et `pass` (pour les initiés)

Ce paragraphe est réservé aux personnes maîtrisant déjà les boucles **`while`**. *Les autres, vous pouvez l'ignorer.*
Certains programmes nécessitent l'utilisation d'un facteur externe venant influencer la façon dont le programme fonctionne. Le cas échéant, il est possible de forcer le programme à :

- quitter complètement une boucle, en utilisant l'instruction **`break`**
- sauter une partie d'une boucle avant de continuer, en utilisant l'instruction **`continue`**
- ignorer ce facteur externe, en utilisant l'instruction **`pass`**

Ces 3 instructions sont utilisables pour les boucles **`for`** ou **`while`**

I.5.a instruction `break` pour quitter une boucle

L'instruction break donne la possibilité de quitter une boucle au moment où une condition externe est déclenchée. Elle s'intègre dans le bloc du code qui se trouve en dessous de l'instruction de boucle, généralement après une instruction conditionnelle **`if`**.

Exemple 1 :

```
for letter in "Python":
    if letter == "h":
        break
    print("Current letter : ", letter)
print("The end")
```

affiche :

```
Current letter : P
Current letter : y
Current letter : t
The end
```

Dans ce court programme, l'itération s'effectue sur chaque lettre du mot `"Python`. Si la lettre rencontrée est h, la boucle est cassée. Ainsi, toutes les lettres sont affichées jusqu'au h. Au delà, la boucle s'arrête.

Exemple 2 :

```
for nombre in range(2, 10):
    for facteur in range(2, n):
        if nombre % facteur == 0:
            print(nombre, ' = ', facteur, ' * ', nombre//facteur)
            break
        else:
            print(nombre, 'est un nombre premier')
```

affiche :

```
2 est un nombre premier
3 est un nombre premier
4 = 2 * 2
5 est un nombre premier
6 = 2 * 3
7 est un nombre premier
8 = 2 * 4
9 = 3 * 3
```

I.5.b instruction `continue` pour sauter une partie de boucle

L'instruction **`continue`** permet d'interrompre l'itération actuelle de la boucle, mais contrairement à l'instruction **`break`**, le programme poursuite les tours de boucle suivant.
L'instruction continue se trouve dans le bloc de code sous l'instruction de boucle, généralement après une instruction conditionnelle **`if`**.
Exemple 1 :

```
for letter in "Python":
    if letter == "h":
        continue
    print("Current letter : ", letter)

    print("The end")
```

affiche :

```
Current letter : P
Current letter : y
Current letter : t
Current letter : o
Current letter : n
The end
```

Dans ce court programme, l'itération s'effectue sur chaque lettre du mot `"Python`. Si la lettre rencontrée est h, le reste du tour de boucle est ignoré et l'itération continue. Ainsi, toutes les lettres sont affichées sauf le(s) h.

I.5.c instruction `pass` pour ne rien faire

L'instruction **`pass`** ne fait rien. Elle peut être utilisée dans des situations où une déclaration est syntaxiquement nécessaire, mais que le programme ne requiert aucune action. Par exemple :

```
if x < 2:
    print(x)
elif x < 6:
    pass
else:
    print(x**2)
```

Elle peut être également utilisée pour réserver de la place pour ultérieure. Par exemple :

```
def une_fonction_que_je_programmerai_plus_tard(arguments):
    pass
```

II Exercices de niveau facile : 1ère approche avec la boucle `while`

Exercice 1 - Indice de dépassement
La suite (u_n) est définie par : $u_0 = 2$ et $\forall n \in \mathbb{N}^*, u_n = 2 * u_{n-1} - 1$
Trouver le plus petit indice n tel que $u_n \geq 1000$. Vérifier le résultat.

Exercice 2 - Indice de dépassement 2
Mêmes questions pour la suite (v_n) est définie par : $v_0 = 1$, $v_1 = 3$, et $\forall n \in \mathbb{N}, v_{n+2} = v_{n+1} + v_n$

On peut charger la fonction `randint` du module `random` à l'aide de l'instruction :

```
from random import randint
```

Un tirage de pile-ou-face est simulé par le choix aléatoire d'un entier 0 ou 1 (`randint(0,1)`).

Exercice 3 - Hasard
Écrire une fonction qui fait des tirages successifs jusqu'à tirer 3 fois 1 consécutivement et renvoie le nombre de tirages qui ont été nécessaires.
Par exemple si les tirages donnent 0,1,1,0,0,1,0,1,1,0,0,0,1,0,1,1,1,...la valeur renvoyée doit être 17.

III Exercices indépendants de niveau moyen

Exercice 4 - Logarithme entier
Écrire une fonction `log_entier(n)` qui renvoie l'entier p tel que $2^p \leq n < 2^{p+1}$, sans utiliser la fonction logarithme.
Vérifier votre résultat en important la fonction `log2` depuis la bibliothèque `math`

Exercice 5 - Somme de deux termes
Écrire une fonction `valeur_egale_2elements(liste_nb, valeur)` qui renvoie deux indices *indice*1 et *indice*2, *indice*1 < *indice*2, tels que
`liste[indice1] + liste[indice2] == valeur`; la fonction renverra `(-1, -1)` s'il n'existe pas de tels indices.

IV Exercices interdépendants de niveau moyen : suite de Collatz

La suite de Collatz est définie sa valeur initiale $u_0 \in \mathbb{N}^*$, pour $n \in \mathbb{N}$,

$$u_{n+1} = \begin{cases} 3u_n + 1 & \text{si } u_n \text{ est impair} \\ u_n/2 & \text{si } u_n \text{ est pair} \end{cases}$$

Dans un premier temps, la boucle `for` sera utilisée.

Exercice 6 - Valeur de la suite de Collatz
Écrire une fonction `Collatz(u_0, n)` qui détermine terme d'indice n de la suite de Collatz de terme initial u_0.
Par exemple, pour $u_0 = 13$, la suite est $13, 40, 20, 10, 5, 16, 8, 4, 2, 1, 4, 2, 1, \ldots$.
La suite $13, 40, 20, 10, 5, 16, 8, 4, 2, 1, 4, 2, 1$ est le vol depuis $u_0 = 13$.

IV.1 Valeurs caractéristiques

Il semble[1] que, pour toute valeur initiale $u_0 \geq 1$, la suite atteint la valeur 1 après un nombre fini d'itérations et devient alors périodique.
Dans les exercices, u_0 sera supposé ≥ 1 sans avoir besoin de le vérifier.

Exercice 7 - Longueur de vol
Écrire une fonction `longueur_Collatz(u_0)` qui détermine la longueur de vol pour la suite de Collatz de terme initial u_0, c'est-à-dire le premier entier n tel que $u_n = 1$.

Les valeurs de u_n peuvent être grandes, même pour u_0 assez petit. Par exemple, pour $u_0 = 703$, $u_{82} = 250504$.

Exercice 8 - Hauteur de vol
Écrire un programme `hauteur_Collatz(u_0)` qui détermine la hauteur de vol de la suite de Collatz de terme initial u_0 , c'est-à-dire la valeur u_n la plus grande de toute la suite de Collatz, de premier terme u_0.

Exercice 9 - Hauteur de vol, bis
Modifier la fonction précédente pour qu'elle renvoie, en plus l'indice n en lequel u_n atteint son maximum.

IV.2 Seuils

Le but de ce paragraphe est de détecter les valeurs de u_0 en lesquelles les caractéristiques ci-dessus dépassent une valeur posée.

Exercice 10 - Longueur de vol à atteindre
Écrire une fonction `longueur_seuil(long_vol_mini)` qui renvoie le plus petit entier u_0 tel que la longueur de vol?? de la suite de Collatz de premier terme u_0 est supérieure à `long_vol_mini`. La fonction renverra aussi la longueur atteinte.
`longueur_seuil(100)` renverra `(27, 111)`,
`longueur_seuil(250)` renverra `(6171, 261)`.

1. C'est une conjecture non prouvée.

Exercice 11 - Hauteur de vol à atteindre
Écrire une fonction `hauteur_seuil(hauteur_mini)` qui calcule le plus petit entier u_0 tel que la hauteur de vol**??** de la suite de Collatz de premier terme u_0 est supérieure à `hauteur_mini`. La fonction renverra le triplet formé de la valeur initiale u_0, de la hauteur atteinte et de l'indice $n_hauteur$ en le quel la hauteur est atteinte.
On recevra le résultat de `hauteur_Collatz` par l'instruction

```
hauteur, indice_hauteur = hauteur_Collatz(u_0)
```

`hauteur_seuil(10000)` renverra `(13120, 255, 15)`,
`hauteur_seuil(10**9)` renverra `(1570824736, 77671, 71)`.

V Exercices de niveau "difficile"

Exercice 12 - Nombres parfaits
Un nombre est dit **parfait** *s'il est la somme de ses diviseurs, distincts de lui-même, ses diviseurs* **propres**.
Les deux premiers nombres parfaits sont $6 = 1 + 2 + 3$ *et* $28 = 1 + 2 + 4 + 7 + 14$.
Écrire une fonction qui calcule la somme des diviseurs propres d'un entier, puis trouver les 2 nombres parfaits suivants.

Exercice 13 - Somme de deux termes 2
Reprendre l'exercice 5 en supposant que la liste est déjà triée par ordre croissant. On doit parvenir à un algorithme plus rapide, avec une seule boucle.

VI Exercices de niveau "très difficile"

Exercice 14 - Nombre indéterminé de boucles for imbriquées
Un programme donné fonctionne à l'aide un certain nombre de boucles **`for`** imbriquées :

```
for j in range(2,8,2) :
    for k in range(3):
        for l in range(3,6):
            for m in range(2,8,2)
                ...
                partie utile de la boucle
                ...
```

Le nombre de boucles **`for`** et les arguments `(start, stop, step)` de la fonction **`range`** sont fournis en argument du programme dans une liste. Dans l'exemple ci-dessus, la liste est `[[2,8,2], 3, [3,6], [2,8,2]]`. La partie utile de la boucle n'utilise pas les variables de boucle `j, k, m, ...`
Écrire une fonction `for_imbriquees(liste_range)` qui prend en argument `liste_range`, une liste d'instructions, et qui effectue les boucles **`for`** imbriquées, comme dans l'exemple. Ici, aucune partie utile de boucle n'est demandée en particulier. Le programme pourra par exemple se contenter donc d'afficher la liste d'instructions avec un compteur qui défile pour chacune des variables de boucle, afin de vérifier que le programme fonctionne correctement. Dans l'exemple ci-dessus, il affichera la liste des [start, stop, step, compteur] pour chaque boucle :

```
[[2,8,2,2], [0,3,1,0], [3,6,1,3], [2,8,2,2]]
[[2,8,2,2], [0,3,1,0], [3,6,1,3], [2,8,2,4]]
[[2,8,2,2], [0,3,1,0], [3,6,1,3], [2,8,2,6]]
[[2,8,2,2], [0,3,1,0], [3,6,1,4], [2,8,2,2]]
[[2,8,2,2], [0,3,1,0], [3,6,1,4], [2,8,2,4]]
[[2,8,2,2], [0,3,1,0], [3,6,1,4], [2,8,2,6]]
[[2,8,2,2], [0,3,1,0], [3,6,1,5], [2,8,2,2]]
...
[[2,8,2,6], [0,3,1,2], [3,6,1,5], [2,8,2,4]]
[[2,8,2,6], [0,3,1,2], [3,6,1,5], [2,8,2,6]]
```