II - Éléments de programmation (suite)

1) - Boucles (suite)

Rappel : Les boucles permettent de répéter une suite d'instructions. Il existe deux types de boucles, selon que le nombre d'itérations est connu à l'avance ou non.

a) - Boucles for ou boucles pour (suite)

Syntaxe d'une boucle for :

```
for k in objet_itérable:
   Bloc d'instructions de la boucle for
```

EXERCICE 1 : Suite géométrique
Écrire un programme qui calcule à l'aide d'une boucle la somme $S_n = \sum_{k=1}^{n} q^k$ où n et q sont fournis par l'utilisateur (notez que vous disposez d'un moyen de vérification...).

Une **fonction Python** est un programme qui porte un nom. L'exécution d'un tel programme est obtenu en appelant cette fonction par son nom suivi éventuellement de paramètres entre parenthèse.
Exemple de fonctions :
* `print, input, int, float, bool, abs`,...qui sont prédéfinies (on parle de fonctions natives ou primitives),
* `exp, log, sqrt, sin, cos, tan, acos, asin, atan, floor, factorial`,...de la bibliothèque math ou numpy,
* `random, randint, choice` de la bibliothèque random.

Les fonctions précédentes sont prédéfinies ou appartiennent à des bibliothèques classiques (math, random, numpy, scipy, matplotlib,...). On peut également définir ses propres fonctions avec le code suivant :

```
def nomDeLaFonction(liste de paramètres):
   Bloc d'instructions de la fonction
```

On prendra soin d'indenter (obligatoire) le bloc d'instructions. Lors de l'appel de la fonction `nomDeLaFonction`, le bloc d'instructions est exécuté avec les paramètres effectifs entrés par l'utilisateur.
On peut appeler une fonction d'un autre programme par son nom suivi de valeurs numériques pour les paramètres.

EXERCICE 2 : Suite géométrique bis
Dans un fichier, définir une fonction de paramètres n et q qui calcule et affiche la valeur de la somme $S_n = \sum_{k=1}^{n} q^k$.

Tester cette fonction pour plusieurs valeurs de n et de q.

b) - Boucles while ou boucles conditionnelles

Elles sont adaptées aux cas où on ne connaît pas à l'avance le nombre de répétitions d'un suite d'instructions. Leur syntaxe est la suivante :

```
while condition:
    Bloc d'instructions de la boucle while
```

`Bloc d'instructions` est répété tant que le booléen `condition` est vrai.
On prendra soin d'indenter (obligatoire) le bloc d'instructions.

EXERCICE 3 : Algorithme de Héron ou de Babylone. On considère la suite définie par $u_0 = x > 0$ et $u_{n+1} = \frac{u_n + \frac{x}{u_n}}{2}$.
Donner une interprétation géométrique de cette suite en remarquant que u_{n+1} est la moyenne arithmétique de la longueur et de la largeur d'un rectangle d'aire x et de longueur u_n puis conjecturer la valeur de la limite de (u_n). Vérifier cette conjecture à l'aide de la fonction Python *heron* dont on complètera le code.

```
def heron(x, precision):
   u = x
   v = (x + 1)/2
   while abs(v - u) > precision:
      u, v = ________________________
   print('Une valeur approchée de ____________________',x,' est',v)
```

Assurez-vous d'avoir bien compris ce que fait cette fonction, et notamment d'avoir saisi l'importance de l'ordre des deux instructions à l'intérieur de la boucle. Au besoin construire un tableau montrant l'évolution de u et v.
Appeler `heron` plusieurs fois pour $x = 2$ et $x = 3$ et pour des précisions de plus en plus petites.
Signalons tout de suite un risque d'erreur important inhérent à ce type de boucle : celui de faire une boucle infinie, si la condition de maintien dans la boucle est toujours réalisée. C'est notamment le cas lorsque la condition ne dépend d'aucune variable ou que les variables intervenant dans la condition n'évoluent pas dans le bloc d'instruction de la boucle.
Appeler cette fonction avec les paramètres 2 et $\frac{1}{10}$ puis taper `v` et la touche entrée dans le shell. Que se passe-t-il ? Cette erreur s'explique par le fait que la variable `v` est **locale**, elle est donc supprimée après l'appel de *heron*.
Pour que cette fonction se comporte comme une vraie fonction mathématique, on remplacera la commande `print('Une valeur approchée de la racine carrée de ',x,' est ',v)` par `return v`. Cette commande renvoie la valeur de `v` à chaque appel de la fonction nouvellement nommée `heron_bis`, permettant d'intégrer cette valeur dans un calcul.

Définition de la commande *return* : dès que le programme de la fonction rencontre l'instruction `return v`, celui-ci s'interrompt après avoir renvoyé la valeur de `v`.

Dans la console, exécuter les instructions `heron_bis(2,0.1)-sqrt(2)`, `heron_bis(2,0.01)-sqrt(2)`, `heron_bis(2,0.001)-sqrt(2)`, `heron_bis(2,0.0001)-sqrt(2)`. Donner une interprétation des résultats obtenus.

Dans une boucle, la condition située après le mot réservé `while` inclut des tests de comparaisons entre des expressions (supérieur, inférieur, égal), et il est possible de relier des conditions par des connecteurs logiques (et, ou, non). Résumons tout ceci dans des tableaux :

Symboles de comparaison	Signification littérale
==	égal à
!=	différent de
<	strictement inférieur à
<=	inférieur ou égal à
>	strictement supérieur à
>=	supérieur ou égal à

Connecteurs logiques	Signification littérale
and	et
or	ou (inclusif)
not	non

Définition de la commande *break* : dès que le programme rencontre l'instruction *break* dans une boucle, celle-ci s'interrompt et le programme continue après la boucle.

EXERCICE 4 : Factorielle

Écrire une fonction d'argument n qui calcule $n!$ avec une boucle d'en-tête `while True:` contenant l'instruction `break`. Améliorer le code de cette fonction en remplaçant la commande *break* par la commande *return*.

c) - Choisir la bonne boucle

Maintenant que vous connaissez les deux grands types de boucles, entraînez-vous à en écrire quelques-unes, en essayant de choisir à chaque fois la méthode la plus adaptée.

EXERCICE 5

Écrire un script qui calcule le n-ième terme de la suite définie par $u_0 = a \geqslant 0$, $u_1 = b \geqslant 0$ et, pour tout entier naturel n, $u_{n+2} = \sqrt{u_{n+1}} + \sqrt{u_n}$ où n, a et b sont fournis par l'utilisateur. On pourra compléter le code suivant :

```
n = eval(input('Entrer n pour obtenir u[n-1] : '))
u = _____ : ')
v = _____ : ')
for _____ :
    u, v = ____, _____
print(_____)
```

Exécuter ce programme pour plusieurs valeurs du couple (a, b) puis conjecturer la limite de (u_n) dans tous les cas.

EXERCICE 6

Soit (u_n) la suite définie par $u_0 = 1$ et $\forall n \in \mathbb{N}$, $u_{n+1} = u_n + u_n^2$ et (v_n) la suite définie par $v_n = \frac{1}{2^n}\ln(u_n)$.

1. Écrire une fonction `suiteu` de paramètre n qui renvoie la valeur de u_n. On pourra compléter le code suivant :
   ```
   def suiteu(n):
       u = ____
       for k in range(____):
           u = __________
       return u
   ```
   Utiliser cette fonction pour calculer u_7 et u_8.
2. Écrire une fonction `suitev` de paramètre n qui renvoie la valeur de v_n. On pourra compléter le code suivant :
   ```
   def suitev(n):
       return _____
   ```
   Conjecturer à l'aide de cette fonction ($n = 7$ et $n = 8$) que la suite (v_n) converge vers une limite $a \in]0, +\infty[$.
3. Écrire une fonction `limite` de paramètre p qui renvoie u_{n+1} où n est le plus petit entier vérifiant $|u_n - u_{n+1}| \leqslant p$. Cette fonction renverra donc une approximation de a si p est "petit".
4. Pour n assez grand, $\frac{1}{2^n}\ln(u_n)$ sera proche de a donc $\ln(u_n)$ devrait être "proche" de $a2^n$ et u_n devrait être "proche" de e^{a2^n}. Vérifions cette dernière conjecture à l'aide d'un programme Python.
   Écrire une fonction `comparaison` de paramètres n et p qui renvoie le rapport $\dfrac{u_n}{e^{b2^n}}$ ainsi que la différence $u_n - e^{b2^n}$ où b est une approximation de a à la "précision" p obtenue par la fonction *limite*.
5. Appeler la fonction *comparaison* avec les paramètres $n = 6$ et $p = \frac{1}{10000}$. La conjecture de la question précédente semble-t-elle se confirmer ?