1 Problème : approximation de la racine carrée

Soit $A > 0$, on cherche à trouver une approximation du nombre $a = \sqrt{A}$, c'est à dire l'unique nombre positif tel que $a^2 = A$. Pour faire cela, il est possible d'utiliser la méthode de Héron : si $c \neq a$ alors le nombre c' défini par

$$c' = \frac{1}{2}(c + \frac{A}{c})$$

est plus proche de a que ne l'était c. En outre, en remplaçant c par c' suffisamment de fois, on se rapproche autant que l'on veut de c.

On peut donc écrire la fonction suivante :

```
def approx_sqrt(A, n):
    """renvoie une approx de sqrt(A), A>=0 apres n iteration de la formule de Heron"""
    c = A
    for i in range(n):
        c = 0.5*(c+A/c)
    return c
```

Cette fonction est intéressante mais ne donne aucune garantie sur la précision du résultat : pour `n = 10` a-t-on un précision d'un dixième, d'un centième, etc. ? Ce que l'on voudrait, c'est répéter l'instruction `c = 0.5*(c+A/c)` **tant que** le résultat n'est pas suffisamment proche de la racine cherchée.

2 Boucle `while`

Lorsque l'on veut répéter un bloc d'instructions non pas un nombre de fois fixé à l'avance mais plutôt **tant qu'**une condition est ou n'est pas vérifiée, on peut utiliser une boucle `while`.

Tester le programme suivant :

```
a = 1
while a < 1000000:
    print("A cette etape a =", a, "< 1000000", "on continue")
    a = 2*a
print("Maintenant a =", a, " >= 1000000", "on s'arrete")
```

Ce programme double la valeur de la variable `a` valant initialement `1` tant que `a<1000000`, c'est à dire jusqu'à ce que `a` soit supérieur ou égal à `1000000`.

La syntaxe d'une boucle `while` est la suivante :

```
while <condition>:
    <instructions repetees tant que la condition n est pas verifiee>
```

Un boucle while répète une suite d'instructions comme une boucle `for`. Cependant :

- on ne maîtrise pas (par défaut) le nombre de passages dans la boucle ;
- on ne dispose pas (par défaut) d'une variable décrivant le numéro du passage.

Retour sur la racine carrée : Dans notre exemple, on peut maintenant contrôler l'erreur d'approximation faite en répétant un certain nombre de fois l'instruction `c = 0.5*(c+A/c)`.

En effet, supposons que l'on veuille garantir que notre approximation c de $a = \sqrt{A}$ soit telle que

$$|c - a| < \epsilon$$

où $\epsilon > 0$ est une certaine marge d'erreur (la plus proche possible de zéro). On peut montrer mathématiquement que, pour cela, il suffit d'avoir :

$$|c^2 - A| < \epsilon^2$$

D'où la fonction suivante :

```
def approx_sqrt(A, epsilon):
    """renvoie une approx de sqrt(A) avec une erreur d'au plus epsilon"""
    c = A
    while abs(c**2-A)>=epsilon**2 :
        c = 0.5*(c+A/c)
    return c
```

Dès lors, l'appel de `approx_sqrt(2,0.001)` renvoie une approximation de $\sqrt{2}$ précise à 3 décimales. Comparer les résultats de cette fonction à la fonction `sqrt` de la bibliothèque `math` (qui renvoie elle même une approximation).

La différence entre cette fonction et celle proposée dans la partie 1 est la suivante :

— dans la partie 1, on donne le nombre d'étapes à effectuer mais on ne contrôle pas l'erreur entre la valeur du résultat `c` et le résultat théorique $\sqrt{A}$;

— dans cette partie, on contrôle cette erreur (en précisant `epsilon`) **mais on ne contrôle pas le nombre d'étapes effectuées**.

3 Instruction `break`

Il est possible de terminer l'exécution d'une boucle avec l'instruction `break`. Bien que cela puisse être fait en modifiant légèrement la structure du bloc d'instruction de la boucle, cela peut simplifier grandement le code.

Lorsque l'instruction `break` est rencontrée par l'interpréteur, celui-ci sort immédiatement de la boucle en cours et passe à la suite du code. Tester le programme suivant :

```
while True :
    a = input("Racontez moi votre vie... ("stop" pour arreter)")
    if a == "stop":
        break
print("fin")
```

La boucle `while True:` correspond normalement à une boucle infinie, en effet la condition `True` étant toujours vraie, la boucle ne s'arrête jamais. Cependant, le `break` permet d'en sortir si l'utilisateur entre `"stop"`.

4 Recherche dans un tableau

On se penche ici sur un problème classique utilisant une boucle `while` : la recherche d'un élément dans un tableau.

On veut écrire une fonction `recherche(t,x)` prenant en argument un tableau `t` et une valeur `x` qui renvoie `True` si `v` se trouve dans `t` et `False` sinon.

Il y a plusieurs manières de répondre à cette question, mais le principe est toujours le même : on parcours le tableau puis :

— si on a trouvé `x` on renvoie `True` ;

— sinon on renvoie `False`.

Voyons une première solution avec une boucle `for` :

```
def recherche(t, x):
    """teste si la valeur x est dans le tableau t"""
    present = False
    for i in range(len(t)):
        if t[i]==x:
            present = True
    return present
```

Ici on a par défaut une variable `present` valant `False` (tant qu'on a pas parcouru le tableau, on ne peut pas affirmer que `x` s'y trouve) qui passe à `True` si on rencontre `x`. On renvoie la valeur de cette variable à la fin.

L'inconvénient de cette méthode est que l'on parcourt tout le tableau, même si `x` est trouvé rapidement. Par exemple, sur le tableau `t = [1,2, ..., 1000000]`, l'appel de `recherche(2,t)` parcourt les 1000000 valeurs alors que dès le deuxième passage dans la boucle, on sait qu'on va répondre `True`. On peut résoudre se problème en utilisant une boucle `while` :

```
def recherche(t, x):
    """teste si la valeur x est dans le tableau t"""
    present = False
    i = 0
    while i<len(t) and not(present):
        if t[i]==x:
            present = True
        i = i+1
    return present
```

Cette fois, le parcours du tableau s'arrête dès que la valeur à été trouvée. Ainsi, **dans le pire cas**, l'intégralité du tableau n'est parcourue uniquement si `x` en est absent ou à la dernière place.

Remarques :

— La structure

```
i = 0
while i<n:
    <bloc d instructions>
    i = i+1
```

est équivalent à

```
for i in range(n):
    <bloc d instructions>
```

— Il est également possible pour obtenir le même comportement d'utiliser ou bien l'instruction `break` ou bien un retour prématuré (`return True` placé dans la boucle si on trouve `x`).

5 Terminaison

Lorsqu'on utilise des boucles `while`, il est possible que le programme ne termine pas, comme on l'a vu avec `while True`. Il est donc important de bien faire attention que la condition de la boucle finisse toujours par être fausse ou éventuellement qu'un `break` soit rencontré au bout d'un certain nombre de passage de boucle.

Les occasions de générer des **boucles infinies** sont nombreuses, et plus le programme est complexe, plus il est difficile de s'assurer qu'elles ne le soient pas. Tester le programme suivant (`ctrl+c`) pour arrêter l'exécution d'un programme sur IDLE :

```
a = 0
while a>0:
    a = 2*a-1
    print(a)
```