Programmer en Python
Licence 2, Mathématiques
V. Monbet
TD 2.
Corrigé

1 Boucles et affichages

1. Que dessinent les programmes suivant ?

```
def mystere1(n):
   for k in range(n, 0,-1):
      print('*'*k)
mystere1(5)
```

```
*****
****
***
**
*
```

La première ilgne du programme ci-dessous permet d'utiliser la syntaxe de la fonction `print` de Python 3 en Python 2.

```
from __future__ import print_function
def mystere2(n):
    for k in range(1, n+1):
       print(' '*(n-k), '* '*k, sep='')
mystere2(5)
```

```
    *
   * *
  * * *
 * * * *
* * * * *
```

2. Le Talkhys est un traité d'arithmétique d'Ibn Albanna, mathématicien marocain de la première moitié du XIIIe siècle. On y trouve un certain nombre d'identités remarquables.
Quelles sont les identités reproduites par la fonction ci-dessous

```
def talkhys2():
    n = 0
    for k in range(1, 10):
        n = 10 * n + k
        print('8 x {:<9} + {} = {:<9}'.format(n, k, 8*n+k))
```

8 x 1	+ 1 = 9
8 x 12	+ 2 = 98
8 x 123	+ 3 = 987
8 x 1234	+ 4 = 9876
8 x 12345	+ 5 = 98765
8 x 123456	+ 6 = 987654
8 x 1234567	+ 7 = 9876543
8 x 12345678	+ 8 = 98765432
8 x 123456789	+ 9 = 987654321

Ecrire une fonction `talkhys` qui permette de reproduire la table

9 x 1	+ 2 = 11
9 x 12	+ 3 = 111
9 x 123	+ 4 = 1111
9 x 1234	+ 5 = 11111
9 x 12345	+ 6 = 111111
9 x 123456	+ 7 = 1111111
9 x 1234567	+ 8 = 11111111
9 x 12345678	+ 9 = 111111111
9 x 123456789	+ 10 = 1111111111

2 Représentation des nombres et égalité

La représentation des nombres réels dans un ordinateur conduit à des approximations. Par exemple

```
In [1]: 0.1 + 0.2
Out[1]: 0.30000000000000004

In [2]:(0.1+0.2)-0.3
Out[2]: 5.551115123125783e-17
```

Ce résultat est dû à la représentation dyadique des réels.
Définition **Nombre décimal**
Un nombre décimal s'écrit $\frac{x}{10^n}$ où $x \in \mathbb{Z}$.
Définition **Nombre dyadique**
Un nombre dyadique s'écrit $\frac{x}{2^n}$ où $x \in \mathbb{Z}$.
Problème : un nombre décimal n'est pas forcément dyadique.
Conséquence :
- la conversion décimal (humain) vers dyadique (machine) peut causer une première approximation ;
- le calcul machine entre nombre dyadiques peut causer une seconde approximation.
Quand on utilise le type float, on utilise jamais l'égalité de valeur `==` ; on fera toujours intervenir une marge d'erreur (absolue ou relative).

```
In [3]: 0.1 + 0.2 == 0.3
Out[3]: False
```

```
In [4]: abs(0.1 + 0.2 - 0.3) <= 1e-10
Out[4]: True
```

Inconvénient : une valeur choisie dans l'absolu parce qu'elle nous parait petite peut se révéler trop grande lorsque les nombres à comparer sont eux-même très petits ou trop petite lorsque les nombres sont très grands.
- La tolérance absolue consiste à fixer $\epsilon > 0$ et à convenir que x et y sont proches dès lors que $|x - y| < \epsilon$.
• La tolérance relative consiste à fixer$\epsilon > 0$ et à convenir que x et y sont proches dès lors que $|x - y| < \epsilon|y|$. L'inconvénient de ce choix est que la relation n'est pas symétrique.
Dans le module `numpy` la fonction `isclose(x,y)` renvoie `true` si $|x - y| < \epsilon_a + \epsilon_r|y|$.
Dans le module `math` la fonction `isclose(x,y)` renvoie `true` si $|x-y| < \epsilon_a+\epsilon_r max(|x|, |y|)$.

Exercice. Equation linéaire du second degré

1. Écrire une fonction `solve(a,b,c)` qui prend en argument les valeurs a,b puis c pour désigner le polynôme $P(x) = ax^2 + bx + c$
   et qui retourne la ou les racines réelles du polynôme ou, le cas échéant, un message indiquant que ce polynôme n'a pas de racine réelle.

   ```
   def solve(a, b, c):
       delta = b * b - 4 * a * c
       if delta < 0:
           print("pas de solution")
       elif delta > 0:
           x, y = (-b-sqrt(delta))/2/a, (-b+sqrt(delta))/2/a
           print("deux racines simples {} et {}".format(x, y))
      else:
          x = -b/2/a
          print("une racine double {}".format(x))
   ```

2. Expliquer les résultats suivants.

   ```
   >>> solve(0.01, 0.2, 1)
   deux racines simples -10.0000001317 et -9.99999986829
   >>> solve(0.011025, 0.21, 1)
   pas de solution
   ```

   Explication

   ```
   >>> .2 * .2 - 4 * .01 * 1
   6.938893903907228e-18
   >>> .21 * .21 - 4 * .011025 * 1
   -6.938893903907228e-18
   ```

3. Lorsque $\Delta = b^2 - 4ac$ est petit devant b^2 les deux racines sont quasiment confondues. Comment peut-on modifier la fonction `solve` pour éviter ces erreurs d'approximation ?

On remplace la condition $delta = 0$ par $|delta| << b^2$.

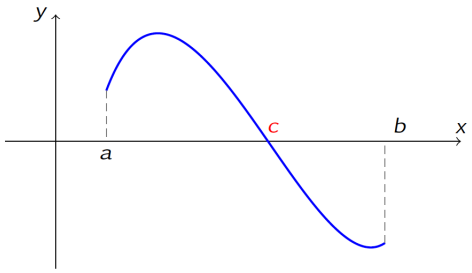
```
def solve2(a, b, c, epsilon= 2**-52)):
    delta = b * b - 4 * a * c
    if delta < - epsilon*b**2:
        print("pas de solution")
    elif delta > epsilon*b**2:
        x, y = (-b-sqrt(delta))/2/a, (-b+sqrt(delta))/2/a
    print("deux racines simples {} et {}".format(x, y))
        else:
        x = -b/2/a
        print("une racine double {}".format(x))
```

Pourquoi 2^{-52} ie 210^{-16} ? Il s'agit de l'epsilon numérique :

```
>>> 1 + 2**-52 == 1
False
>>> 1 + 2**-53 == 1
True
```

3 Recherche d'une racine par dichotomie

Soit $f : [a, b] \to \mathbb{R}$ continue telle que $f(a)f(b) \leq 0$.



1. Prouver que f admet au moins une racine.
2. On considère le cas d'une racine unique. Montrer que suivant le signe de $f\left(\frac{a+b}{2}\right)$ l'une des deux relations est vérifiée
$$f(a)f\left(\frac{a+b}{2}\right) \leq 0 \text{ ou } f\left(\frac{a+b}{2}\right)f(b) \leq 0$$
3. On définie deux suites $(u_n)_{n\in\mathbb{N}}$ et $(v_n)_{n\in\mathbb{N}}$ par $u_0 = a$, $v_0 = b$ et
$$(u_{n+1}, v_{n+1}) = \begin{cases} (u_n, m) & \text{si } f(u_n)f(m) \leq 0 \\ (m, v_n) & sinon \end{cases} \quad \text{avec } m = \frac{u_n + v_n}{2}$$
Montrer que ces deux suites sont convergentes et qu'elles admettent la même limite qui est la solution de l'équation $f(x) = 0$.

On remarque qu'on a à chaque étape, $u_n \leq m_n \leq v_n$. De plus, comme (u_n) est par construction une suite croissante, (v_n) une suite décroissante, et $(u_n - v_n) \to 0$ lorsque $n \to +\infty$, les suites (u_n) et (v_n) sont adjacentes et donc elles admettent une même limite. D'après le théorème des gendarmes, c'est aussi la limite disons ℓ de la suite (m_n). La continuité de f montre que $f(\ell) = \lim_{n\to+\infty} f(m_n) = \lim_{n\to+\infty} 0 = 0$. Donc les suites (u_n) et (v_n) tendent toutes les deux vers ℓ, qui est une solution de l'équation $f(x) = 0$.

4. Proposer une fonction `dicho` pernant comme arguments `f,a,b,epsilon` qui itère les deux suites $(u_n)_{n\in\mathbb{N}}$ et $(v_n)_{n\in\mathbb{N}}$ jusqu'à convergence des suites.

```
def dicho(f, a, b, epsilon=1e-12):
    if f(a) * f(b) > 0:
        return None
    u, v = float(a), float(b)
    while abs(v - u) > 2 * epsilon:
        w = (u + v) / 2
        if f(u) * f(w) <= 0:
            v = w
        else:
            u = w
   return (u + v) / 2
```

On peut aussi écrire une version récursive

```
def dichotomie(a,b,prec):
      if b-a<=prec:
          return a,b
      else:
          c = (a+b)/2
          if f(a)*f(c) <= 0:
              return dichotomie(a,c,prec)
          else:
              return dichotomie(c,b,prec)
```

5. Analyser la terminaison de l'algorithme.
Validité : $\forall n \in \mathbb{N},\ f(u_n)f(v_n) \leq 0$.
Terminaison : lorsque $v_n - u_n \leq 2^n$, il existe une racine c de f vérifiant : $|w - c| < \epsilon$ avec $w_n = \frac{u_n+v_n}{2}$.
Coût : $v_n - u_n = \frac{b-1}{2^n}$ et $\frac{b-a}{2^n} \leq 2\epsilon$ est équivalent à $n \geq \log_2\left(\frac{b-a}{\epsilon}\right) - 1$.
Si $\epsilon = 10^{-p}$, l'algorithme se termine lorsque :

$$n \geq \log_2(b-1) + p\log_2(10) - 1$$

C'est à dire que le coût est linéaire en p.

6. Proposer une adaptation de la fonction `dicho` qui permette d'ajouter un nombre maximal d'itérations avant d'aboutir à un échec.

```
def dicho(f, a, b, epsilon=1e-12):
    if f(a) * f(b) > 0:
```

```
        return None
    n = 0
    u, v = a, b
    while abs(v - u) > 2 * epsilon:
        n += 1
        if n > maxiter:
            chn = 'Échec après {} itérations.'.format(maxiter)
            raise RuntimeError(chn)
        w = (u + v) / 2
        if f(u) * f(w) <= 0:
            v = w
        else:
            u = w
    return (u + v) / 2
```

4 Recherche d'une racine par

L'idée de la méthode de la sécante est très simple : pour une fonction f continue sur un intervalle $[a, b]$, et vérifiant $f(a) < 0$, $f(b) > 0$,on trace le segment $[AB]$ où $A = (a, f(a))$ et $B = (b, f(b))$.Si le segment reste au-dessus du graphe de f alors la fonction s'annule sur l'intervalle $[a', b]$ où $(a', 0)$ est le point d'intersection de la droite (AB) avec l'axe des abscisses. La droite (AB) s'appelle la sécante. On recommence en partant maintenant de l'intervalle $[a', b]$ pour obtenir une valeur a''.

Proposition
Soit $f : [a, b] \to$ une fonction continue, strictement croissante et convexe telle que $f(a) < 0$, $f(b) > 0$. Alors la suite définie par

$$a_0 = a \quad \text{et } a_{n+1} = a_n - \frac{b - a_n}{f(b) - f(a_n)} f(a_n)$$

est croissante et converge vers la solution ℓ de $(f(x) = 0)$.

1. Faire un dessin représentant la méthode de la sécante.
2. Justifier la construction de la suite récurrente.

   L'équation de la droite passant par les deux points $(a, f(a))$ et $(b, f(b))$ est

   $$y = (x - a)\frac{f(b) - f(a)}{b - a} + f(a)$$

   Cette droite intersecte l'axe des abscisses en $(a', 0)$ qui vérifie donc

   $$0 = (a' - a)\frac{f(b) - f(a)}{b - a} + f(a),$$

   donc

   $$a' = a - \frac{b - a}{f(b) - f(a)} f(a)$$

3. a_n croissante
   Montrons par récurrence que $f(a_n) \leq 0$.
   C'est vrai au rang 0 car $f(a_0) = f(a) \leq 0$ par hypothèse.
   Supposons vraie l'hypothèse au rang n.
   Si $a_{n+1} < a_n$ (un cas qui s'avérera a posteriori jamais réalisé), alors comme f est strictement croissante, on a $f(a_{n+1}) < f(a_n)$, et en particulier $f(a_{n+1}) \leq 0$; sinon $a_{n+1} \geq a_n$.
   Comme f est convexe : la sécante entre $(a_n, f(a_n))$ et $(b, f(b))$ est au-dessus du graphe de f . En particulier le point $(a_{n+1}, 0)$ (qui est sur cette sécante par définition a_{n+1}) est au-dessus du point $(a_{n+1}, f(a_{n+1}))$, et donc $f(a_{n+1}) \leq 0$ aussi dans ce cas, ce qui conclut la récurrence.
   Comme $f(a) \leq 0$ et f est croissante, alors par la formule $a_{n+1} = a_n - \frac{b-a_n}{f(b)-f(a_n)} f(a_n)$, on obtient que $a_{n+1} \geq a_n$.
4. Convergence (a_n)
   La suite (a_n) est croissante et majorée par b, donc elle converge. Notons ℓ sa limite.
   Par continuité $f(a_n) \to f(l)$.
   Comme pour tout n, $f(a_n) \leq 0$, on en déduit que $f(l) \leq 0$. En particulier, comme on suppose $f(b) > 0$, on a $\ell < b$.
   Comme $a_n \to \ell$,$a_{n+1} \to \ell$,$f(a_n) \to f(\ell)$, l'égalité

$$a_{n+1} = a_n - \frac{b - a_n}{f(b) - f(a_n)} f(a_n)$$

   devient à la limite (lorsque $n \to +\infty$) :

$$\ell = \ell - \frac{b - \ell}{f(b) - f(\ell)} f(\ell)$$

   ce qui implique $f(\ell) = 0$.

def secante(a,b,n) : for i in range(n) : a = a-f(a)*(b-a)/(f(b)-f(a)) return a