Lycée Louis-Le-Grand, Paris
MPSI 4 – Informatique pour tous
A. Troesch, J.-P. Becirspahic

Corrigé du TP n° 1

Exercices 1 et 2.

Des deux premiers exercices, retenons que tout objet python possède un *type* dont dépend la manière dont cet objet sera représenté (sous forme binaire) en mémoire. Vous avez rencontré les types :
– `int`, qui modélise les entiers relatifs ;
– `float`, qui modélise les nombres décimaux ;
– `str`, constitué de chaînes de caractères ;
– et enfin le type `tuple`, qui permet de manipuler des n-uplets.
Toute opération ou fonction agit sur un type particulier ; il faudra donc prendre soin de toujours bien distinguer l'entier `1` du nombre flottant `1.0`. En effet, les opérations sur les objets de type `int` sont toujours exactes, mais ce n'est pas le cas des opérations sur le type `float` :

```
>>> 0.1 + 0.2 - 0.3
5.551115123125783e-17
```

Certaines opérations peuvent conduire à des conversions automatiques de type. C'est notamment le cas lorsqu'on effectue une opération élémentaire entre un flottant et un entier : ce dernier est d'abord converti avant de réaliser l'opération entre nombres flottants. Notons en particulier que la division `/` retourne forcément un flottant, même lorsque le quotient est entier :

```
>>> 4 / 2
2.0
```

Pour calculer le quotient et le reste d'une division euclidienne entre nombres entiers, on utilisera les opérateurs `//` et `%`.

```
>>> 5 // 2
2
>>> 5 % 2
1
```

Outre son type, tout objet python est caractérisé par son *identifiant* correspondant à l'emplacement en mémoire où ce dernier est stocké :

```
>>> id(1)
1824320
>>> id(2)
1824336
```

Affecter une valeur à une variable, c'est réaliser une liaison entre le nom choisi pour cette variable et l'identifiant de sa valeur. Lorsque la valeur affectée à cette variable est modifiée, la liaison est elle aussi modifiée :

```
>>> x = 1
>>> id(x)
1824320
>>> x += 1
>>> id(x)
1824336
```

Une conséquence de ce mécanisme est que l'instruction `y = x` réalise une liaison non pas entre les variables `y` et `x` mais entre la variable `y` et la *valeur* de `x` au moment où cette instruction est réalisée. Toute modification ultérieure de `x` n'aura aucune incidence sur `y`.

Retenons enfin que le type `tuple` permet l'affectation simultanée de deux variables, et en particulier l'échange de leur contenu :

```
>>> x, y = 1, 2
>>> x, y = y, x
>>> x, y
(2, 1)
```

Exercices 3 et 4.

Les chaînes de caractères, encadrées par des guillemets simples ou doubles, constituent le type `str` (*string* en anglais).

La fonction **print** convertit en chaînes de caractères les valeurs de ses différents paramètres avant de les afficher à l'écran, séparés par la chaîne de caractères sep, l'affichage se terminant par la chaîne end. Par défaut, sep est un espace et end un passage à la ligne (représenté par le caractère \n), mais il est possible de modifier ces paramètres :

```
>>> print('un', 'deux', 'trois', sep=' et ', end='.\n')
un et deux et trois.
```

La fonction input quant à elle, affiche à l'écran son unique paramètre (une chaîne de caractères) et met l'interprète de commande en pause tant que l'utilisateur n'a pas écrit un texte terminé par un retour chariot. Ce texte est alors renvoyé par la fonction input sous la forme de chaîne de caractères.
Enfin, la fonction eval tente de convertir une chaîne de caractère en valeur numérique, int ou float, si cela est possible.

Exercices 6 et 7.

Le mot-clé **def** permet de définir une fonction. L'entête de la définition précise l'ensemble (éventuellement vide) des paramètres, et le résultat est renvoyé par le mot-clé **return**. En l'absence de ce dernier, la fonction retournera la valeur particulière None. C'est le cas par exemple de la fonction **print** :

```
>>> (print(1), 2)
1
(None, 2)
```

L'exemple ci-dessus montre que la fonction **print** réalise un effet sur l'environnement (l'affichage de la chaîne '1\n') puis renvoie la valeur None comme première composante du tuple.
Considérons les trois définitions suivantes :

```
def f(x):
    return 1/(1+x**2)

def f1(x):
    print('f(', x, ') = ', f(x), sep='')

def f2():
    x = eval(input('valeur de x ? '))
    f1(x)
```

la fonction f exige un paramètre numérique et renvoie un nombre flottant. La fonction f1 attend elle aussi un paramètre numérique, mais renvoie la valeur None après avoir produit un affichage. Enfin, la fonction f2 ne prend aucun paramètre et retourne la valeur None.

```
>>> f(1)
0.5
>>> f1(1)
f(1) = 0.5
>>> f2()
valeur de x ?    # ici l'utilisateur a rentré la valeur 2 au clavier
f(2) = 0.2
```

Exercice 8.

La syntaxe générale des structures conditionnelles est la suivante :

```
if (test 1):
    bloc..............
    d'instructions 1..
elif (test 2):
    bloc..............
    d'instructions 2..
else:
    bloc..............
    d'instructions 3..
```

– Si le test 1 est positif, le bloc d'instructions 1 est réalisé ;
– si le test 1 est négatif et le test 2 positif, le bloc d'instructions 2 est réalisé ;
– si les deux tests sont négatifs, le bloc d'instructions 3 est réalisé.
Évidemment, plusieurs **elif** à la suite peuvent être utilisés pour multiplier les cas possibles, mais n'oubliez pas que les tests sont effectués *séquentiellement*, c'est à dire les uns après les autres jusqu'à trouver un test positif (ou arriver au cas final **else**).

Par exemple :

```
def bissextile(n):
    if n < 1582:
        if n % 4 == 0:
            return True
        else:
            return False
    else:
        if n % 4 == 0:
            if n % 400 == 0:
                return True
            elif n % 100 == 0:
                return False
            else:
                return True
        else:
            return False
```

Notons que les expressions booléennes peuvent être évaluées et qu'un test n'est jamais qu'un *calcul* sur les éléments du type `bool`. Il est donc tout à fait possible d'écrire la fonction précédente comme suit :

```
def bissextile(n):
    return n % 4 == 0 and (n < 1582 or n % 400 == 0 or n % 100 != 0)
```

Exercice 9.

Il est facile de montrer que la suite $(S_n)_{n\in\mathbb{N}^*}$ diverge vers $+\infty$, ce qui assure la *terminaison* (c'est à dire le fait de retourner une valeur en un temps fini) de la fonction suivante :

```
def rang(a):
    n, s = 1, 1
    while s <= a:
        n += 1
        s += 1 / n
    return n
```

Il est aussi possible de prouver que pour de grandes valeurs de n, $S_n \sim \ln(n)$. On peut donc s'attendre à ce que le rang calculé par cette fonction soit de l'ordre de e^a ; il est donc préférable de ne tenter l'expérience qu'avec de faibles valeurs de a, faute de quoi l'attente risque d'être longue.

Le calcul du pgcd de deux entiers utilise l'algorithme d'Euclide :

```
def pgcd(a, b):
    u, v = a, b
    while v > 0:
        u, v = v, u % v
    return u
```

On peut calculer le ppcm de deux entiers à l'aide de la formule : $ab = \text{pgcd}(a,b) \times \text{ppcm}(a,b)$ (en n'oubliant pas d'utiliser la division entière pour ne pas obtenir un résultat de type `float`).

```
def ppcm(a, b):
    return a * b // pgcd(a, b)
```

Exercice 10.

Cet exercice permet de rencontrer une première *méthode* appliquée à une classe d'objets : la méthode `format` pour les objets de type `str`. Quelques exemples :

```
>>> x = 2/7
>>> print('mon nombre vaut {}'.format(x))
mon nombre vaut 0.2857142857142857
>>> print('mon nombre vaut {:g}'.format(x))
mon nombre vaut 0.285714
>>> print('mon nombre vaut {:.3f}'.format(x))
mon nombre vaut 0.286
>>> print('mon nombre vaut {:.2e}'.format(x))
mon nombre vaut 2.86e-01
```

Il est facile de prouver que la suite $(u_n)_{n\in\mathbb{N}}$ définie par les relations $u_0 = 1$ et $u_{n+1} = \sin(u_n)$ converge vers 0 en décroissant, ce qui assure la terminaison de la fonction suivante. Néanmoins, compte tenu de la relative lenteur de sa convergence, nous n'afficherons que les valeurs dépassant 10^{-2} :

```
from math import sin

def suite():
    n, u = 0, 1.
    while u > 1.e-2:
        print('u_{} = {:g}'.format(n, u))
        n, u = n + 1, sin(u)
```

Voici ce que donne cette fonction :

```
>>> suite()
u_0 =  1
u_1 = 0.841471
u_2 = 0.745624
u_3 = 0.67843
u_4 = 0.627572
.................
u_29988 = 0.0100006
u_29989 = 0.0100004
u_29990 = 0.0100002
u_29991 = 0.0100001
```

Les valeurs peuvent être numérotées (en partant de 0) comme suit : `{0:}`, `{1:}`, etc. de façon à pouvoir afficher les valeurs plusieurs fois ou dans un ordre différent. Par exemple :

```
def suite():
    n, u, v = 0, 5., 15.
    while abs(v-u) > 1.e-10:
        print('u_{0:} = {1:.4f}, v_{0:} = {2:.4f}'.format(n, u, v))
        n += 1
        u, v = (2*u+v)/3, (u+2*v)/3
```

```
>>> suite()
u_0 = 5.0000, v_0 = 15.0000
u_1 = 8.3333, v_1 = 11.6667
........................
u_22 = 10.0000, v_22 = 10.0000
u_23 = 10.0000, v_23 = 10.0000
```

Dernier exemple, pour afficher les puissances de 2 en trois colonnes :

```
def puissances():
    n, p = 1, 2
    while p < 1e20:
        print('{:>20}'.format(p), end = ' ')
        n, p = n + 1, 2 * p
        if n % 3 == 1:
            print()      # pour passer à la ligne au bout de trois colonnes
```

```
>>> puissances()
                   2                   4                   8
                  16                  32                  64
                 128                 256                 512
                1024                2048                4096
                8192               16384               32768
               65536              131072              262144
              524288             1048576             2097152
             4194304             8388608            16777216
            33554432            67108864           134217728
           268435456           536870912          1073741824
          ..........          ..........          ..........
```

Exercice 11.

Considérons la suite s définie par : $\forall n \geqslant 2$, $s_n = \sum_{k=2}^{n} \frac{(-1)^k}{k\ln(k)}$. Il est possible de prouver sa convergence en observant que les suites $(s_{2n})_{n\geqslant 1}$ et $(s_{2n+1})_{n\geqslant 1}$ sont adjacentes.
Cette observation a un second avantage : nous pouvons affirmer que la limite S est toujours comprise entre deux termes consécutifs de la suite s, et donc que : $|S - s_n| \leqslant |s_{n+1}| = \frac{1}{(n+1)\ln(n+1)}$.

Il nous reste à trouver une valeur de n vérifiant $\frac{1}{(n+1)\ln(n+1)} \leqslant 10^{-8}$ pour connaître un rang n à partir duquel s_n approche S à la précision souhaitée. Ceci conduit au calcul suivant :

```
from math import log
n, s = 2, 0
while 1/n/log(n) > 1e-8:
    s += (-1)**n/n/log(n)
    n += 1
print('S ~ {:.8f}'.format(s))
```

qui produit l'affichage :

```
S ~ 0.52641224
```

Exercice 12.

Cet exercice est l'occasion de rencontrer le troisième type numérique disponible : le type `complex`.
La fonction `help` nous renseigne au sujet de la syntaxe de la fonction permettant de les définir :

```
complex(real[,imag])

-> complex number

Create a complex number from a real part and an optional imaginary part.
This is equivalent to (real + imag*1j) where imag defaults to 0.
```

Profitons-en pour observer la façon dont l'aide nous renseigne sur les paramètres indispensables et facultatifs (présentés entre crochets). En l'absence d'une partie imaginaire, celle-ci est prise égale à 0.
Ceci nous amène à définir la méthode de résolution des équations su second degré de la façon suivante :

```
from math import sqrt

def quadratic(a, b, c):
    """ résolution de l'équation ax**2+bx+c lorsque a, b et c sont réels """
    delta = b**2-4*a*c
    if delta > 0:
        x1, x2 = (-b-sqrt(delta))/2/a, (-b+sqrt(delta))/2/a
        print('deux racines réelles {:g} et {:g}'.format(x1, x2))
    elif delta == 0:
        x = -b/2/a
        print('une racine double {:g}'.format(x))
    else:
        x1, x2 = complex(-b/2/a,-sqrt(-delta)/2/a), complex(-b/2/a, sqrt(-delta)/2/a)
        print('deux racines complexes {:g} et {:g}'.format(x1, x2))
```

Illustration :

```
>>> solutions(1, 2, -3)
deux racines réelles -3 et 1
>>> solutions(1, 2, 1)
une racine double -1
>>> solutions(1, 1, 1)
deux racines complexes -0.5-0.866025j et -0.5+0.866025j
```

La chaîne de caractères délimitée par les caractères `"""` est celle qui apparaîtra si vous sollicitez de l'aide sur la fonction que vous venez de créer :

```
>>> help(quadratic)
Help on function quadratic in module __main__:

quadratic(a, b, c)
    résolution de l'équation ax**2+bx+c lorsque a, b et c sont réels
```

Il convient enfin de préciser que l'algorithme ainsi écrit peut conduire à des résultats de précision médiocre dans certains cas ; ce problème sera évoqué (et résolu) plus tard dans l'année.