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

2013/2014

Corrigé du TP n° 2 (première partie)

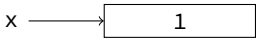
La correction des exercices 1 et 2 figure dans le corrigé précédent.

Exercice 3.

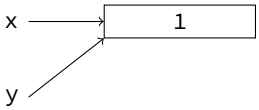
```
def collatz():
    u = eval(input('Choisissez une valeur de départ : '))
    n, m = 0, u
    while u != 1:
        if u % 2 == 0:
            u /= 2
        else:
            u = 3 * u + 1
        n += 1
        m = max(u, m)
    print('rang = {}, maximum = {}'.format(n, m))
```

Exercice 6.

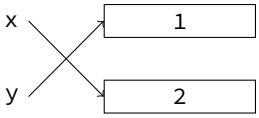
Cet exercice était l'occasion de comprendre la différence entre un type de données immuable ou mutable.
Rappelons que lorsqu'on écrit dans l'interprète `x = 1`, on réalise une liaison entre le nom choisi (ici `x`) et un emplacement en mémoire contenant la représentation binaire de la valeur liée (ici `1`) :



L'instruction `y = x` réalise une deuxième liaison entre `y` et le même emplacement en mémoire :

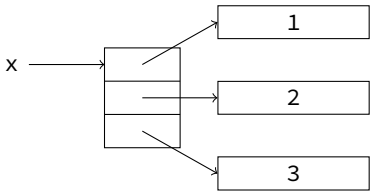


Enfin, l'instruction `x += 1` supprime la liaison précédente pour créer une nouvelle liaison entre `x` et un nouvel emplacement en mémoire contenant la représentation binaire de `2` :

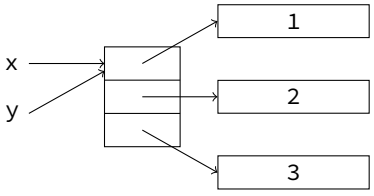


Dans le cas d'une donnée immuable, les modifications de la variable `x` ne sont pas répercutées sur la variable `y`.

Il en va tout autrement dans le cas d'une liste, qui est un type de données mutable.
À une liste est allouée un certain nombre de cases mémoires consécutives, qui chacune contient l'*adresse* d'une valeur. Par exemple, l'instruction `x = [1, 2, 3]` peut être représentée par le schéma suivant :



L'instruction `y = x` réalise une liaison entre `y` et le même bloc mémoire que celui pointé par `x` :



On comprend que dès lors, toute modification du contenu de la liste `x` se répercutera sur `y` puisqu'il s'agit du même bloc d'adresses. La preuve :

```
>>> x = [1, 2, 3]
>>> y = x
>>> x[1] = 4
>>> y
[1, 4, 3]
```

Si on veut créer une véritable copie d'une liste, il faudra donc utiliser le slicing (`x[:]` crée une copie de la liste `x`) ou la méthode `copy()` :

```
>>> x = [1, 2, 3]
>>> y = x.copy()
>>> x[1] = 4
>>> y
[1, 2, 3]
```

Exercice 7.

Notez que malgré la similitude des noms, la fonction et le module qu'on vous demande d'utiliser ici ne sont pas les mêmes que ceux de l'exercice 4. La fonction `randint` que nous allons utiliser possède un troisième paramètre optionnel indiquant la taille d'un tableau qui sera rempli aléatoirement. Ceci nous conduit à définir :

```
from numpy.random import randint

def alea(n=100):
    return randint(n, size=n)
```

La fonction `alea` possède un paramètre optionnel qui par défaut est égal à 100, mais que l'on peut modifier si nécessaire :

```
>>> alea(n=10)
array([5, 2, 0, 6, 4, 4, 6, 6, 4, 9])
>>> alea()             # par défaut la valeur prise par n est 100
array([31, 19, 50, 63, 11, 54, 21, 85, 46, 61, 40, 41,  9, 28, 91, 36, 68,
       12, 82,  1, 45, 99, 36, 10, 40, 57, 76, 46, 66, 46, 25, 31, 95, 56,
       61, 46, 70,  4,  3,  9, 26, 56, 91, 82, 90, 70, 45, 39, 34, 77, 71,
       32, 84, 93, 30, 58, 48, 89, 98, 43, 13, 72, 47, 90, 78, 83, 46, 53,
       54, 61, 98, 13, 71, 80, 59,  2, 76, 57, 95, 59, 13, 12, 65, 18, 35,
       10,  0, 65,  6, 12, 41, 81, 16, 60, 80, 56,  9, 15, 57, 72])
```

On parcourt ensuite la liste des entiers de 0 à n en comptant ceux qui ne sont pas présents dans la liste :

```
def non_present(t):
    s = 0
    for k in range(len(t)):
        if k not in t:
            s += 1
    return s
```

Il nous reste à définir une fonction pour réaliser un nombre x d'expériences :

```
def experience(x, n=100):
    s = 0
    for k in range(x):
        s += non_present(alea(n))
    return s / x
```

Essayons avec un millier d'expériences, et comparons avec la valeur théorique $n\left(\frac{n-1}{n}\right)^n$:

```
>>> experience(1000)
36.838
>>> 100*(99/100)**100
36.60323412732292
```

Avec une autre valeur de n :

```
>>> experience(1000, n = 256)
94.068
>>> 256*(255/256)**256
93.99289725223328
```

Exercice 8. In and Out shuffle

Si `lst` est une liste de longueur n, alors :
– `lst[:n//2]` calcule la première moitié de la liste ;
– `lst[n//2:]` calcule la seconde moitié de la liste ;
– `lst[::2]` calcule la liste des éléments de rangs pairs ;
– `lst[1::2]` calcule la liste des éléments de rangs impairs.
Une fois ceci rappelé, la définition des fonctions `out_shuffle` et `in_shuffle` devient évidente :

```
def out_shuffle(lst):
    n = len(lst)
    lst[::2], lst[1::2] = lst[:n//2], lst[n//2:]

def in_shuffle(lst):
    n = len(lst)
    lst[::2], lst[1::2] = lst[n//2:], lst[:n//2]
```

Notons que ces deux fonctions modifient la liste passée en paramètre et retournent la valeur `None`. Pour calculer la longueur du cycle il va falloir garder copie de la liste initiale :

```
def cycle(n):
    init = list(range(n))
    lst = init.copy()
    out_shuffle(lst)
    s = 1
    while lst != init:
        out_shuffle(lst)
        s += 1
    return s
```

```
>>> cycle(52)
8
```

Pour le *In Shuffle* il faut 52 mélanges pour retrouver la liste initiale.