Correction du DS 2

Informatique pour tous, première année

Julien Reichert

Exercice 1

Il n'y a pas de raison de travailler sur $\frac{1}{60}$ autrement qu'en tant que fraction, ce qui facilite les calculs sans entraver les comparaisons à 1. On sait qu'en binaire notre nombre commence par $0,\dots$ puisqu'il s'agit de la partie entière en décimal aussi. Finalement, on va préférer travailler sur $\frac{1}{15}$, sachant que la fraction $\frac{1}{60}$ en est le quart, d'où un exposant valant 2 de moins.

Procédons donc aux multiplications par 2 successives.

$$\begin{aligned}
2\times \tfrac{1}{15} &= \mathbf{0} + \tfrac{2}{15}\\
2\times \tfrac{2}{15} &= \mathbf{0} + \tfrac{4}{15}\\
2\times \tfrac{4}{15} &= \mathbf{0} + \tfrac{8}{15}\\
2\times \tfrac{8}{15} &= \mathbf{1} + \tfrac{1}{15}
\end{aligned}$$

À ce stade, on a trouvé un cycle et on arrête le calcul.

Ainsi, $\frac{1}{15}$ s'écrit en écriture scientifique binaire $1,00010001\cdots\times 2^{-4}$. C'est normal, puisqu'on peut aussi voir ceci comme la limite de la somme pour i démarrant à 1 des $\frac{1}{16^i}$. Comme on a dit, $\frac{1}{60}$ s'écrit alors en écriture scientifique binaire $1,00010001\cdots\times 2^{-6}$.

L'exposant -6 se représente sur 8 bits comme $-6+2^{8-1}-1$, soit $\overline{01111001}^2$. L'écriture de la mantisse en virgule flottante omettant le 1 avant la virgule, on a donc les bits 00010001000100010001001 car, le bit suivant étant un 1 suivi ultérieurement de quelques 1 (seulement une infinité…), l'arrondi se fait par excès.

La représentation finale est alors 0 01111001 00010001000100010001001.

Exercice 2

Le script (a) provoque une erreur lors de l'appel de la fonction car la variable `i` n'est pas définie (elle pourrait être globale, mais le script se limitant aux lignes écrites ici, l'erreur est inévitable).

Le script (b) provoque une erreur à deux titres lors de l'appel de la fonction : en tant qu'entier, `42` ne supporte pas la fonction `len`[1], et quand bien même l'entrée serait par exemple une liste, l'élément `a[len(a)]` auquel on tente d'accéder au premier tour dans la boucle n'existe pas et cause un dépassement.

1. On dit qu'il n'est pas *sized*.

Le script (c) provoque une erreur lors de l'exécution du script. En effet, la ligne `return 1`, n'étant pas indentée, n'est pas dans la définition de la fonction, et un `return` en-dehors d'une fonction cause une erreur.

Le script (d) est correct. Ceci étant, le fait de faire un `return` dans la portée d'un `for` sans mettre de condition fait s'interrompre la boucle dès le premier tour, ce qui est en général une maladresse. En tout cas, là n'était pas la question.

Le script (e) provoque une erreur lors de l'exécution du script. Là aussi, la ligne `print(n)` est hors de la définition de la fonction, et `n` n'est pas définie (ce qui aurait sauvé les meubles).

Exercice 3

```
def somme(l):
    s = 0
    for element in l: # ou for i in range(len(l)) puis element = l[i]
        s += element
    return s
```

Exercice 4

La possibilité la plus simple est de faire des tests à la chaîne pour savoir dans lequel des six cas possibles (pour autant de permutations) on se trouve.

```
def arrange1(a, b, c):
    if a <= b:
        if c <= a:
            return [c, a, b]
        elif c <= b:
            return [a, c, b]
        else:
            return [a, b, c]
    else:
        if c <= b:
            return [c, b, a]
        elif c <= a:
            return [b, c, a]
        else:
            return [b, a, c]
```

Il est également possible d'échanger les variables jusqu'à ce qu'on ait forcément `a <= b <= c`. Il s'agit d'un tri à bulles pour une petite instance.

```
def arrange2(a, b, c):
    if a > b:
        a, b = b, a # Maintenant b est le plus grand des deux
    if b > c:
        b, c = c, b # Maintenant c est le plus grand des trois
    if a > b:
        a, b = b, a # Maintenant a est le plus petit des trois
    return [a, b, c]
```

Exercice 5

```
def harmonie(n):
    s = 0
    for i in range(1, n+1): # Attention aux bornes !
        s += 1 / i
    return s
```

Exercice 6

La réponse est directement donnée pour le logarithme en base quelconque, avec un paramètre par défaut fixé à `2` (fonctionnalité très utile de Python). Attention, l'ordre des arguments n'est pas celui dans lequel ils apparaissent dans l'énoncé (ce n'est pas un problème en soi quand il n'est pas fermement imposé, mais mieux vaut prendre garde dans l'absolu).

Attention, le logarithme peut être négatif ; l'oubli de ce fait n'entraîne pas une grosse pénalité, mais il faut savoir le faire une fois qu'on y a pensé, et aussi penser que la base elle-même peut être inférieure à `1`.

```
def logarithme(n, base=2):
    assert n > 0, "Logarithme d'un nombre négatif !"
    reponse = 0
    if base < 1:
        base = 1 / base
        n = 1 / n
    while n >= base: # Une seule de ces deux boucles
        n /= base
        reponse += 1
    while n < 1: # ... peut être visitée
        n *= base
        reponse -= 1
    return reponse
```