Examen Python – M1

Stéphane Vialette

7 novembre 2012

La durée de l'examen est de 2 heures. Tous les document sont autorisés (l'utilisation des téléphones et ordinateurs portables n'est pas autorisée). Lire le sujet en entier avant de commencer ! Les questions jugées les plus difficiles sont annotées **(*)**.

1 Listes

QUESTION 1 Écrire la fonction `cumulative_sums(l)` qui retourne la liste de toutes les sommes des listes préfixes non vides de `l` (sauf si `l` est la liste vide, auquel cas la fonction retourne la liste vide), *i.e.*,

$$\texttt{cumulative\_sums}([x_1, x_2, \ldots, x_k]) = [x_1, x_1 + x_2, \ldots, x_1 + x_2 + \ldots + x_k].$$

Un exemple d'utilisation :

```
# cumulative_sums([]): []
print 'cumulative_sums([]):', cumulative_sums(range(0))

# cumulative_sums([0]): [0]
print 'cumulative_sums([0]):', cumulative_sums(range(1))

# cumulative_sums([0, 1]): [0, 1]
print 'cumulative_sums([0, 1]):', cumulative_sums(range(2))

# cumulative_sums([0, 1, 2]): [0, 1, 3]
print 'cumulative_sums([0, 1, 2]):', cumulative_sums(range(3))

# cumulative_sums([0, 1, 2, 3]): [0, 1, 3, 6]
print 'cumulative_sums([0, 1, 2, 3]):', cumulative_sums(range(4))
```

QUESTION 2 Écrire la fonction `cumulative(fun, l)` qui retourne la liste de toutes les applications de la fonction `f` aux éléments des listes préfixes non vides de `l` (sauf si `l` est la liste vide, auquel cas la fonction retourne la liste vide), *i.e.*,

$$\texttt{cumulative}(\texttt{fun}, [x_1, x_2, \ldots, x_k]) = [\texttt{fun}(x_1), \texttt{fun}(x_1, x_2), \ldots, \texttt{fun}(x_1, x_2, \ldots, x_k)].$$

La fonction `fun` doit donc accepter un nombre arbitraire d'arguments (y compris 0) et non pas une liste.

Pour illustrer notre propos, voici une implémentation alternative de la fonction `cumulative_sums(l)` de la question précédente :

```
def my_add(*args):
    return sum(args)

# cumulative(my_add, []): []
print 'cumulative(my_add, []):', cumulative(my_add, range(0))

# cumulative(my_add, [0]): [0]
print 'cumulative(my_add, [0]):', cumulative(my_add, range(1))

# cumulative(my_add, [0, 1]): [0, 1]
print 'cumulative(my_add, [0, 1]):', cumulative(my_add, range(2))

# cumulative(my_add, [0, 1, 2]): [0, 1, 3]
print 'cumulative(my_add, [0, 1, 2]):', cumulative(my_add, range(3))

# cumulative(my_add, [0, 1, 2, 3]): [0, 1, 3, 6]
print 'cumulative(my_add, [0, 1, 2, 3]):', cumulative(my_add, range(4))
```

QUESTION 3 Une liste $l = [x_1, x_2, x_3, x_4, \ldots, x_n]$ est dite *alternée* si (1) chaque élément est du signe contraire de celui qui le précède, (2) les éléments de rang (d'indice) pair sont égaux à eux-mêmes, et (3) les éléments de rang (d'indice) impair sont égaux à leur opposé. Par exemple, la liste $l = [0, -1, 2, -3, 4, -5]$ est une liste alternée. Écrire la fonction `is_alternate(l)` qui retourne `True` si la liste `l` passée en argument est alternée, et `False` sinon.

QUESTION 4 Une *sous-liste* de la liste `l` $= [x_1, x_2, \ldots, x_n]$ est une liste obtenue en supprimant certains éléments de `l` (et en conservant l'ordre), c'est-à-dire une liste (éventuellement vide) de la forme $[x_{\sigma(1)}, x_{\sigma(2)}, \ldots, x_{\sigma(k)}]$ avec $1 \leq \sigma(1) < \sigma(2) < \ldots < \sigma(k) \leq n$. Écrire la fonction `all_subsets(l)` qui retourne la liste de toutes les sous-listes (y compris la liste vide) de la liste `l` passée en argument. Une implémentation récursive vient de l'observation suivante : Si L est une liste non vide et $x \in L$ est un élément quelconque de L, alors l'ensemble de toutes les sous-listes de L, noté simplement $\mathcal{P}(L)$, peut être obtenu par

$$\mathcal{P}(L) = \mathcal{P}(L - x) \cup \{L' + x : L' \in \mathcal{P}(L - x)\}$$

où nous avons noté $L - x$ la liste obtenue en supprimant l'élément x de la liste L, et $L + x$ la liste obtenu en ajoutant l'élément x à la liste L. L'unique sous-liste de la liste vide est la liste contenant la liste vide, et nous rappelons que la liste vide est l'élément neutre pour la concaténation de listes.

Un exemple d'utilisation :

```
# all_sublists([]): [[]]
print 'all_sublists([]):', all_sublists([])

# all_sublists([0, 1, 2]): [[], [2], [1], [1, 2], [0],
#                           [0, 2], [0, 1], [0, 1, 2]]
print 'all_sublists([0, 1, 2]):', all_sublists(range(3))
```

2 Générateurs et programmation fonctionnelle

2.1 `izip_with`

Le module `itertools` fournit des fonctions rapides pour générer des itérateurs (sous forme de générateurs), et remplacer directement certaines primitives comme `map()`, `filter()`, `reduce()` et `zip()`. Donnons dans un premier temps quelques rappels relatifs à la fonction `itertools.izip(*iterables)` :

```
>>> import itertools as it
>>> iter = it.izip(['a', 'b', 'c'], [1, 2, 3], ['A', 'B', 'C'])
>>> list(iter)
[('a', 1, 'A'), ('b', 2, 'B'), ('c', 3, 'C')]
```

Lorsque les itérateurs sont de longueurs différentes, `itertools.izip(*iterables)` s'arrête dès que l'itérateur le plus petit est consommé. Une implémentation possible de `itertools.izip(*iterables)` :

```
def izip(*iterables):
    # izip('ABCD', 'xy') --> Ax By
    iterators = map(iter, iterables)
    while iterators:
        yield tuple(map(next, iterators))
```

QUESTION 5 Écrire le générateur `izip_with(fun, *iterables)` qui applique la fonction `fun` aux i-èmes éléments de `iterables` et retourne son évaluation. Il est donc nécessaire que la fonction `fun` passée en argument accepte un nombre arbitraire (y compris 0) d'arguements. Par exemple le programme de test suivant :

```
import operator

def mult_add(*args):
    return reduce(operator.add, args, 0)
g = zip_with(mult_add, xrange(0, 10), xrange(10, 20), xrange(20, 30))
print 'zip_with(mult_add, ...):', g
print ','.join(str(x) for x in g)
```

```
def mult_prod(*args):
    return reduce(operator.mul, args, 1)
g = zip_with(mult_prod, xrange(0, 10), xrange(10, 20), xrange(20, 30))
print 'zip_with(mult_mul, ...):', g
print ','.join(str(x) for x in g)
```

produit la sortie

```
zip_with(mult_add, ...): <generator object zip_with at 0x109f205a0>
30,33,36,39,42,45,48,51,54,57
zip_with(mult_mul, ...): <generator object zip_with at 0x109f205f0>
0,231,528,897,1344,1875,2496,3213,4032,4959
```

En effet,

- $0+10+20=30, 1+11+21=33, \ldots, 9+19+29=57$, et
- $0\times 10\times 20=0, 1\times 11\times 21=231, \ldots 9, \times 19\times 29=4959$.

QUESTION 6 Écrire une implémentation de la fonction `izip(*iterables)` du module `itertools` qui utilise `izip_with(fun, *iterables)`. Rappelons ici que la fonction `tuple` prend **au plus** un argument, *i.e.*,

```
In [1]: ?tuple
Type:       type
Base Class: <type 'type'>
String Form:<type 'tuple'>
Namespace:  Python builtin
Docstring:
tuple() -> empty tuple
tuple(iterable) -> tuple initialized from iterable items

If the argument is a tuple, the return value is the same object.
```

2.2 Suite de Fibonacci

Nous allons utiliser le générateur `izip_with(fun, *iterables)` de l'exercice précédant pour programmer un générateur de la suite de Fibonacci. Rappelons que la suite de Fibonacci est définie par

$$\text{fib}(n) = \begin{cases} 1 & \text{pour } n = 0 \text{ ou } n = 1, \\ \text{fib}(n-1) + \text{fib}(n-2) & \text{pour } n \geq 2. \end{cases}$$

Pour nous guider, écrivons dans un premier temps sur deux lignes les premiers termes de la suite de Fibonacci et ceux de la suite de Fibonacci décalés d'une position vers la gauche :

suite de Fibonacci	1	1	2	3	5	8	13	21
suite de Fibonacci décalée	1	2	3	5	8	13	21	34

Nous remarquons (par définition en fait !) que

suite de Fibonacci	1	1	2	3	5	8	13	21
	+	+	+	+	+	+	+	+
suite de Fibonacci décalée	1	2	3	5	8	13	21	34
	2	3	5	8	13	21	34	55

QUESTION 7 (*) Écrire un générateur (récursif bien sûr !) de la suite de Fibonacci utilisant `izip_with(fun, *iterables)` de l'exercice précédant. Nous supposerons disposer de la fonction `consume` qui "consomme" un élément de l'itérable passé en argument et retourne ce dernier :

```
def consume(iterator):
    next(iterator)
    return iterator
```

2.3 Programmation trampoline

Considérons le fragment de code suivant.

```
def tail_rec(fun):
    def tail(fun):
        a = fun
        while callable(a):
            a = a()
        return a
    return (lambda x: tail(fun(x)))

def tail_doohickey(x):
    print 'tail_doohickey(%d)' % (x,)
    if x == 0:
        return True
    else:
        return (lambda: tail_gizmo(x - 1))

def tail_gizmo(x):
    print 'tail_gizmo(%d)' % (x)
    if x == 0:
        return False
    else:
        return (lambda: tail_doohickey(x - 1))

doohickey = tail_rec(tail_doohickey)
gizmo = tail_rec(tail_gizmo)
```

```
if __name__ == '__main__':
    x = 5
    print 'doohickey(%d): %s' % (x, doohickey(x))
```

QUESTION 8 (*) Donner la sortie de ce programme.

QUESTION 9 Que font les fonctions `tail_doohickey` et `tail_gizmo` ?