



Python (M1) – Des fonctions et des listes

TD #1

Rédacteur: Stéphane Vialette

1 Fonction itératives et récursives sur les listes

1.1 Longueur

Écrire (en itératif et récursif) une fonction prenant une liste en argument et retournant le nombre d'éléments la composant (sans utiliser la fonction `len` de python) Exemple d'utilisation :

```
>>> l = range(10)
>>> l
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> import tdl.len
>>> tdl.len.len_iterative(l)
10
>>> tdl.len.len_recursive(l)
10
>>>
```

[FIGURE 1 about here.]

1.2 Renversement

Écrire une fonction `reverse` qui prend en argument une liste et retourne la liste des mêmes éléments mais dans l'ordre inverse (encore une fois, il ne saurait être question d'utiliser la méthode `reverse` de la classe `list`). Exemple :

```
>>> import tdl.reverse
>>> tdl.reverse.reverse([])
[]
>>> l = range(10)
>>> l
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> tdl.reverse.reverse(l)
[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
>>>
```

[FIGURE 2 about here.]

1.3 Maximum

Écrire (en itératif et récursif) une fonction prenant une liste non vide en argument et retournant un de ses éléments maximaux (sans utiliser la fonction `max` de python)
Exemple d'utilisation :

```
>>> l = range(10)
>>> l
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> import tdl.max
>>> tdl.max.max_iterative(l)
9
>>> tdl.max.max_recursive(l)
9
>>> import random
>>> random.shuffle(l)
>>> l
[7, 0, 9, 2, 6, 3, 4, 8, 1, 5]
>>> tdl.max.max_iterative(l)
9
>>> tdl.max.max_recursive(l)
9
>>>
```

[FIGURE 3 about here.]

1.4 Zip

la fonction `zip` retourne une liste de tuples, où le *i*-ème tuple contient le *i*-ème élément de chacune des listes passées en argument. La liste retournée est tronquée à la longueur de la plus courte liste passée en argument. Exemple d'utilisation :

```
>>> x = [1, 2, 3]
>>> y = [4, 5, 6]
>>> zipped = zip(x, y)
>>> zipped
[(1, 4), (2, 5), (3, 6)]
>>> x = [1, 2, 3]
>>> y = [4, 5, 6, 7]
>>> z = [8, 9]
>>> zipped = zip(x, y, z)
>>> zipped
```

```
[(1, 4, 8), (2, 5, 9)]
>>>
```

Écrire une implémentation python de la fonction `zip` (vous pourrez dans un premier temps simplifier l'écriture en supposant que votre fonction `zip` ne prend que 2 listes en arguments).

[FIGURE 4 about here.]

Écrire une implémentation python de la fonction `unzip` effectuant l'opération inverse (vous pourrez une nouvelle fois simplifier l'écriture en supposant dans un premier temps que votre fonction `unzip` prend en argument une liste de paires). Exemple :

```
>>> zipped = [(1, 4), (2, 5), (3, 6)]
>>> x, y = unzip(zipped)
>>> x
[1, 2, 3]
>>> y
[4, 5, 6]
>>> zipped = [(1, 4, 6), (2, 5), (3)]
>>> x, y, z = unzip(zipped)
>>> x
[1, 2, 3]
>>> y
[4, 5]
>>> z
[6]
>>>
```

[FIGURE 5 about here.]

2 Fonctionnelles

2.1 Implémentation

Écrire une implémentation en python des fonctions `filter`, `map` et `reduce`. Consultez

- <http://docs.python.org/library/functions.html#filter>,
- <http://docs.python.org/library/functions.html#map>, et
- <http://docs.python.org/library/functions.html#reduce>

pour une description complète de ces fonctions.

[FIGURE 6 about here.]

[FIGURE 7 about here.]

[FIGURE 8 about here.]

2.2 Pliage

Considérons la fonction `fold_left` définie de la façon suivante :

```
def fold_left(seq, reduce_fun, init_val):
    if not seq:
        return init_val
    return fold_left(seq[1:], reduce_fun, reduce_fun(init_val, seq[0]))

# some tests
add_fun = lambda total, item: total+item
assert fold_left([1, 2, 3, 4, 5, 6], add_fun, 0) == 21
assert fold_left([], add_fun, 0) == 0
assert fold_left(None, add_fun, 0) == 0
```

Que fait cette fonction ? Pouvez vous en donner une expression python plus concise ?

3 Composition ...et autre

3.1 $f \circ g$ et $g \circ f$

Écrire la fonction `compose(f, g)` qui retourne la composée de `f` et `g`. Par exemple :

```
>>> import tdl.compose
>>> succ = lambda x: x+1
>>> double = lambda x: x*x
>>> succ_double = tdl.compose.compose(succ, double)
>>> succ_double
<function h at 0x1004d0668>
>>> [succ_double(i) for i in range(5)]
[1, 2, 5, 10, 17]
>>> double_succ = tdl.compose.compose(double, succ)
>>> double_succ
<function h at 0x1004d7668>
>>> [double_succ(i) for i in range(5)]
[1, 4, 9, 16, 25]
>>>
```

[FIGURE 9 about here.]

3.2 fonction itérée

Écrire la fonction `fun_iter(fun, n)` qui retourne la fonction `fun` itérée `n` fois, *i.e.*, `f(f(f(f(f(...)))))`.

[FIGURE 10 about here.]

3.3 Application partielle

Écrire la fonction `partial(f, x)` qui retourne l'application partielle de `x` à `f` (une fonction prenant exactement 2 arguments). Par exemple :

```
>>> import partial
>>> def addition(x, y):
...     return x+y
...
>>> succ = partial.partial(addition, 1)
>>> succ
<function partial_fun at 0x10f27bed8>
>>> [succ(i) for i in range(10)]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> def multiplication(x, y):
...     return x*y
...
>>> triple = partial.partial(multiplication, 3)
>>> [triple(i) for i in range(10)]
[0, 3, 6, 9, 12, 15, 18, 21, 24, 27]
>>>
```

[FIGURE 11 about here.]

Pour aller plus loin, modifiez votre implémentation pour prendre en argument des fonctions `f` d'arité quelconque.

```
>>> import partial
>>> def f(x, y, z):
...     return x + (2*y) + (3*z)
...
>>> g = partial.partial(f, 2)
>>> g(3, 4)
20
>>> h = partial.partial(g, 3)
>>> h(4)
20
>>>
```

4 Crible d'Ératosthène

Le crible d'Ératosthène est un procédé qui permet de trouver tous les nombres premiers inférieurs à un certain entier naturel donné `N`.

L'algorithme procède par élimination : il s'agit de supprimer d'une table des entiers de 2 à `N` tous les multiples d'un entier. En supprimant tous les multiples, à la fin il ne restera

que les entiers qui ne sont multiples d'aucun entier, et qui sont donc les nombres premiers. On commence par rayer les multiples de 2, puis à chaque fois on raye les multiples du plus petit entier restant. On peut s'arrêter lorsque le carré du plus petit entier restant est supérieur au plus grand entier restant, car dans ce cas, tous les non-premiers ont déjà été rayés précédemment. À la fin du processus, tous les entiers qui n'ont pas été rayés sont les nombres premiers inférieurs à N .

Écrire la fonction `erastothene(N)` qui retourne la liste des nombre premiers inférieurs à N en utilisant la méthode du crible d'Ératosthène.

```

def len_recursive(seq):
    """
    Return the length of a list (recursive)
    """
    if len(seq) == 0:
        return 0
    else:
        return 1 + len_recursive(seq[1:])

def len_iterative(seq):
    """
    Return the length of a list (iterative)
    """
    count = 0
    for elt in seq:
        count = count + 1
    return count

```

FIGURE 1 – Correction Question 1

```

# Version recursive
def reverse_recursive2(seq):
    if seq == []:
        return seq
    else:
        return reverse_recursive2(seq[1:]) + seq[0]

# Version recursive avec une fonction auxiliaire
def reverse_recursive1(seq):
    def reversed_aux(seq, reversed_seq):
        if len(seq) == 0:
            return reversed_seq
        else:
            return reversed_aux(seq[1:], [seq[0]] + reversed_aux)
    return reversed_aux(seq, [])

# Version iterative
def reverse_iterative(seq):
    reversed_seq = []
    for i in range(len(seq) + 1):
        reversed_seq.insert(0, seq[i])
    return reversed_seq

```

FIGURE 2 – Correction Question 2

```

# Version recursive avec fonction auxiliaire
def max_recursive(seq):
    def max_recursive_aux(seq):
        if len(seq) == 1:
            return seq[0]
        else:
            max_after = max_recursive_aux(seq[1:])
            if seq[0] > max_after:
                return seq[0]
            else:
                return max_after
    if seq == []:
        raise TypeError('max_recursive expected a non empty list')
    return max_recursive_aux(seq)

# Version iterative
def max_iterative(seq):
    if seq == []:
        raise TypeError('max_recursive expected a non empty list')
    max_so_far = seq[0]
    for elt in seq:
        if elt > max_so_far:
            max_so_far = elt
    return max_so_far

```

FIGURE 3 – Correction Question 3

```

# Une premiere version pour deux sequences
def zip2(seq1, seq2):
    if not seq1 or not seq2:
        raise TypeError('zip2() expects two sequences')
    ret = []
    min_len = min(len(seq1), len(seq2))
    for i in range(min_len):
        ret.append((seq1[i], seq2[i]))
    return ret

# Une seconde version qui accepte un nombre arbitraire mais non nul de sequences
# On utilise une exception pour gerer la sortie de boucle
def zip(*seqs):
    if not seqs:
        raise TypeError('zip() expects one or more sequence arguments')
    ret = []
    i = 0
    try:
        while True:
            item = []
            for s in seqs:
                item.append(s[i])
            ret.append(tuple(item))
            i = i + 1
    except IndexError:
        return ret

```

FIGURE 4 – Correction Question 4

```

# Une premiere version pour une sequence de paires
def unzip2(seq):
    seq1, seq2 = [], []
    for p in seq:
        x1, x2 = p
        seq1.append(x1)
        seq2.append(x2)
        # ou encore
        # seq1.append(p[0])
        # seq2.append(p[1])
    return seq1, seq2

# Une seconde version qui accepte des paires de taille arbitraire
# mais toutes les paires doivent avoir meme taille ... cette version
# n'est cependant pas parfaite !!! pourquoi ?
def unzip(seq):
    def all_same(items):
        return all(x == items[0] for x in items)
    if and not all_same([len(t) for t in seq]):
        raise TypeError('All tuples must have the same length')
    ret = [[] for _ in range(len(seq))]
    for t in seq:
        print t
        i = 0
        for item in t:
            ret[i].append(item)
            i = i + 1
    return tuple(ret)

```

FIGURE 5 – Correction Question 4 (suite)

```

def filter(fun, seq):
    if fun is None:
        fun = lambda x: x is True
    filtered_seq = []
    for elt in seq:
        if fun(elt):
            filtered_seq.append(elt)
    return filtered_seq

# On triche un peu ... une variante utilisant une comprehension de liste
def filter2(fun, seq):
    if fun is None:
        fun = lambda x: x is True
    return [fun(elt) for elt in seq]

```

FIGURE 6 – Correction Question filter

```
def map(fun, seq):  
    mapped_seq = []  
    for elt in seq:  
        mapped_seq.append(fun(elt))  
    return mapped_seq
```

FIGURE 7 – Correction Question map

```

def reduce(fun, seq, init=None):
    lseq = list(seq)
    if init is None:
        res = lseq.pop(0)
    else:
        res = init
    for elt in lseq:
        res = fun(res, elt)
    return res

```

initialize?
no

yes
reduce sequence
apply function
done

FIGURE 8 – Correction Question reduce

```
def compose(f, g):  
    def h(x):  
        return f(g(x))  
    return h
```

FIGURE 9 – Correction Question compose

```
from compose import compose

def itere(func, n):
    h = lambda x: x
    for _ in range(n):
        h = compose(h, func)
    return h
```

FIGURE 10 – Correction Question fonction itérée

```
def partial(f, x):  
    def partial_fun(*y):  
        return f(x, *y)  
    return partial_fun
```

FIGURE 11 – Correction Question application partielle